HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Hyperspectral Image Classification of Satellite Images Using Compressed Neural Networks

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

eingereicht von: Daniel Rychlewski

Gutachter: Prof. Dr. Björn Scheuermann

Prof. Dr. Ralf Reulke

eingereicht am: 21.11.2019 verteidigt am: 19.12.2019

Table of Contents

Abbreviations 4
1. Introduction 5
1.1 Motivation 5
1.2 Thesis Structure 6
2. Hyperspectral Image Classification 8
2.1 Hyperspectral Imaging 8
2.2 Convolutional Neural Networks 11
2.2.1 Architecture 11
2.2.2 Learning Process 17
2.2.3 Visualization 19
2.2.4 Alternatives 21
2.3 Related Work 23
2.3.1 Hyperspectral Classification 24
2.3.2 Visualization 25
3. Dimensionality Reduction 27
3.1 Image Compression 27
3.1.1 Feature Extraction 28
3.1.2 Feature Selection 33
3.2 Model Compression 37
3.2.1 Parameter Pruning 38
3.2.2 Post-Training Quantization 41
3.3 Related Work 45
3.3.1 Band Selection 45
3.3.2 Pruning 47
3.3.3 Quantization 48
4. Experiments 50
4.1 Tools and Expansions 50
4.1.1 DeepHyperX 50
4.1.2 Iterative Pruning 53
4.1.3 Intel Distiller 54
4.1.4 M2-DeepLearning 56
4.2 Compressions 57
4.2.1 No Compression 57
4.2.2 Image Channel Compression 63
4.2.3 Neural Network Compression 67

4.2.4 Combinations of Compressions 78
4.3 Visualizations 84
4.3.1 Gradient-Based Saliency Maps 84
4.3.2 Activation Maps and Guided Backpropagation 86
5. Outlook 88
6. Conclusion 90
7. Bibliography 94
8. List of Figures 109
9. Appendix 116
9.1 Further Related Work 116
9.1.1 Pruning 116
9.1.2 Quantization 122
9.2 Model Structures 127
9.3 Visualizations 129
9.4 Architecture Considerations 133
9.5 Additional Measurements 134

Abbreviations

AA	Average Accuracy
ANN	Artificial Neural Network
APoZ	Average Percentage of Zeros
AVIRIS	Airborne Visible/Infrared Imaging Spectrometer
CI	Confidence Interval
CNN	Convolutional Neural Network
HSI	Hyperspectral Imaging
Kappa	Kappa Coefficient
k-NN	k-Nearest Neighbor Algorithm
OA	Overall Accuracy
RNN	Recurrent Neural Network
ROSIS	Reflective Optics System Imaging Spectrometer
SVM	Support Vector Machine

1. Introduction

1.1 Motivation

Convolutional neural networks (CNNs) have proven to be a successful instrument for hyperspectral image classification tasks in recent years due to the high accuracies they are able to reach [1]. In their applications, hyperspectral images hold important advantages over RGB images [2-4], especially for the example of satellite images considered in this thesis [5]. For example, the spectral range and precision required to profile materials and organisms can only be gathered by hyperspectral sensors [6]. In doing so, they capture energy spectra of incoming light for all spatially distributed image components (called hyperspectral pixels) for every hyperspectral band, which is typically located 5-20nm next to another one. Since more dots are contained in the reflection profile of the wavelengths, objects on Earth are easier recognizable and distinguishable from one another in the case of satellite images.

As Burger and Gowen have found out, the vast number of image dimensions poses a significant challenge to performant image classification [7]. Ma et al. describe the problem of redundant, strongly correlated bands as a difficulty of using hyperspectral data, which is often "subject to *Hughes phenomenon* where classification accuracy increases gradually in the beginning as the number of spectral bands or dimensions increases, but decreases dramatically when the band number reaches some value" [8]. Furthermore, for a high number of image bands, there is often no sufficiently large training dataset available [9] due to the task of acquiring ground truth data being expensive and complex [10]. Dimensionality reduction techniques should help mitigate these problems linked to the so-called *curse of dimensionality*. This dimensionality reduction can be done on various levels.

On the one hand, one could decide to remove a few of the usually hundreds of image channels [11] which are not meaningful enough for the image classification task. These *dimensionality reduction techniques* can be divided into the two categories of *feature selection*, which removes image bands so that only few of the original ones remain, and *feature extraction*, which combines multiple image bands to form new ones so that fewer of these newly created channels are necessary to represent the most important information. The impact of representatives of both techniques shall be analyzed in this master thesis (cf. chapters 3.1.1 and 3.1.2).

On the other hand, it is possible to compress the neural network itself instead of the images it operates upon. While there are lots of *model compression strategies*, in this thesis, we will focus on the approaches of *parameter pruning*, i.e. the removal of unimportant connections between neurons from the neural network, and *post-training quantization*, which is the attempt to use fewer bits to represent mainly weights and biases of the network by using a code book after the model has been trained. Both techniques have been successfully used before to, e.g., significantly reduce the model size, speedup the inference of the network and improve energy efficiency, all without any loss of accuracy [12].

In this thesis, we will explore the trade-off between compressed image channels and a compressed neural network regarding appropriate parameters (depending on the circumstances, e.g., memory and VRAM usage, time consumption for inference and, most importantly, classification accuracies) to find out the best way for performing dimensionality reduction. For that purpose, the effect of the architecture shall also be analyzed by comparing multiple well-known candidates for hyperspectral image classification in regard to the parameters mentioned above to realize the cost at which a presumably high classification accuracy is reached. To try to understand the implications of image dimensionality reduction and parameter pruning on the classification made by the CNN, intermediate layer *visualizations* such as saliency and activation maps are supposed to give us an insight. Overall, this master thesis will seek to

strike a balance between using the potential of hyperspectral image recognition by CNNs regarding additional spectral information, which is getting increasingly important for image classification tasks [2-4], and addressing the concern of too many dimensions by analyzing the aspects at which dimensionality reduction can be reasonably applied (image channels, CNN architecture, CNN compression), ending with an analysis attempt of why the image recognition of a particular CNN works the way it does through the visualization of the intermediate layers' calculations. It will explore the way compression works at multiple levels.

1.2 Thesis Structure

Overall, the thesis is split into a theoretical (chapters 2 and 3) and a practical (chapter 4) part. Section 2.1 outlines the theoretical foundations regarding hyperspectral imaging (HSI), explaining the motivations, applications and suitability of HSI for deep learning methods like CNNs. Next, CNNs are introduced in chapter 2.2 with regard to core components (2.2.1). Their way of being prepared in a learning process to perform the desired task is outlined in 2.2.2 from the theoretical standpoint. On another note, the question of CNN interpretability through visualization methods is addressed in 2.2.3 and alternatives to CNNs presented in 2.2.4. An insight is given into current HSI classification developments (2.3.1) and important visualization techniques applied so far (2.3.2).

Moving on to the focus of compression methods for dimensionality reduction in 3, we deal with ways of image band selection (3.1), which we categorize into feature extraction (3.1.1) and feature selection approaches (3.1.2), as well as two selected categories of model compression techniques (3.2), parameter pruning (3.2.1) and post-training quantization (3.2.2). Both levels of reducing dimensionality are backed up in 3.3 by explaining appropriate research, i.e. for band selection (3.3.1), pruning (3.3.2) and quantization (3.3.3).

The second half of the thesis is dedicated to the experiments we have conducted (4). First, the tools we used are introduced in 4.1, including our custom expansions to make them work for our hyperspectral scenario. This includes the hyperspectral PyTorch research framework DeepHyperX, forming the foundation with its models and datasets and our implemented band selection techniques (4.1.1), the fine-grained pruning centerpiece Iterative Pruning (4.1.2), the deep learning compression framework Intel Distiller (4.1.3), which we used for quantization and coarse-grained pruning, and the visualization preset M2-DeepLearning (4.1.4). After that, the results of the experiments are presented (4.2), starting with the reference measurement of noteworthy parameters like accuracy metrics, RAM / VRAM usages, inference / training times and model sizes for different deep learning techniques, but mainly numerous deep neural networks (4.2.1). The reduced set of networks and datasets we receive is then used for our compression-related experiments. We examine the feasibility of different feature extraction and selection techniques in 4.2.2.1 and 4.2.2.2 respectively. This is followed by an evaluation of the network compression techniques (4.2.3) of fine-grained pruning (4.2.3.1), coarse-grained pruning (4.2.3.2) and post-training quantization (4.2.3.3), each time describing the setup of the experiment, necessary conversions of model files, framework suitabilities and other noteworthy practical challenges we needed to overcome, accompanied by our observations and explanations of the results. Furthermore, band selection and model compression are combined in 4.2.4 to witness possible improvements over separately performed image and model compressions. This is done in two distinct compression pipelines, both starting with applying band selection, where the first one merely concerns pruning as the model compression technique used (4.2.4.1), whereas the second one adds quantization on top for additional dimensionality reduction (4.2.4.2). Lastly, we apply visualization techniques in 4.3 as an attempt to understand the calculations of a chosen CNN for enhanced interpretability, for which we use gradient-based saliency maps (4.3.1) as well as activation maps and backpropagation (4.3.2).

Rounding off the thesis is an outlook of experiments and analyses we have and have not conducted, together with desirable developments for simplifying further research (5). A conclusion summarizes our findings (6). The appendix starts with additional research to contextualize our findings with, which is

categorized and located in 9.1 and its subchapters. To better recognize the features of the models we selected following the baseline measurement, architecture visualizations are presented in 9.2. More detailed visualizations as a side note of the visualization experiment are proposed in 9.3, together with additional insight into a chosen hyperspectral dataset and a model architecture after quantization. Our architecture considerations in 9.4 allow the reader to quickly get an understanding of the components used for the experiments and to what extent they work together for which purpose. The additional measurements in 9.5 highlight the suitability or lack thereof of hyperspectral datasets for our experiment, but they also illustrate our parameter measurements for the Keras library on one of our models for the outlook chapter to show an example of how our work could be built upon if desired.

Our main contributions are:

- baseline comparison of hyperspectral deep learning solutions and datasets considering suitability with regard to relevant parameters: accuracy metrics, VRAM / RAM usage, model size, training / inference time
- contrasting feature extraction and feature selection techniques with regard to their suitability as band selection techniques for the hyperspectral classification scenario with deep convolutional neural networks
- identifying the impact of two pruning techniques at different granularities and post-training quantization as selected model compression techniques on relevant parameters:
    - fine-grained threshold-based pruning by layer on accuracy metrics and model size, contrasting it with l_1-norm-based coarse-grained pruning with special focus on current practical implementation-related challenges
    - post-training quantization to explore the components' respective importance by varying the numbers of bits used, as well as the tradeoff between accuracy retained and model size savings reached, exploring the current state of deep framework interoperability
- combining image compression and model compression approaches in compression pipelines to evaluate the suitability, especially given the results of non-cascaded compressions:
    - chosen band selection techniques (based on previously obtained results) and fine-grained pruning are combined, with post-training quantization optionally on top, focusing on possible accuracy improvements
- visualization attempts for semantic interpretability are made using saliency (based on gradients as well as guided backpropagation) and activation maps

2. Hyperspectral Image Classification

Convolutional neural networks (CNNs) are of great interest when it comes to hyperspectral image classification tasks [13, 14]. This chapter contains everything needed to understand the characteristics of hyperspectral imaging (2.1) and the theory behind a convolutional neural network (2.2). In the related work section, we will delve into current developments (2.3).

2.1 Hyperspectral Imaging

After the world's first artificial satellite Sputnik's launch into space in 1957 had marked the beginning of the space age, as Landgrebe describes, the advances in spacecraft, pattern recognition technology and the digital computer stimulated the debate around space-based land remote sensing in the early 1960s [15]. Motivated by the purpose of making "observations from space to obtain information to better manage the Earth's […] resources", researchers were met with the task of figuring out which areas should be measured and how to make this task as economical as possible [15]. However, to identify, e.g., corn, not only would a "spatial resolution of the order of centimeters" be required to detect the shape of a corn leaf, but the fact that too much data would be acquired by the sensor made the approach infeasible [15]. This has led to the *multispectral approach*: the pixel sizes were enlarged so that the captured patches included areas, which were big enough so that the response to a function of wavelength could be unique enough to the object. For instance, to capture corn, one could capture a small agricultural area, and the area's response as a function of wavelength would allow to identify corn. This response is called a *reflectance profile* or *spectral profile*, which is illustrated in Figure 1. As Wong describes, a spectral profile is unique for each "type of natural or synthetic terrain", so it can serve as an HSI fingerprint for discrimination and identification [16]. Acquiring spectral reflectances is a lengthy process that involves eliminating sources of distortion such as the smear of the capturing equipment and accounting for spectral stray light, which would otherwise limit the dynamic range and signal-to-noise ratio of the system [17]. This is confirmed by Bioucas-Dias et al., who consider the difficulty of spectral mixing as a challenge of HSI capturing and touch upon the uncertainties related "to the measurement process such as noise and atmospheric effects" [18].

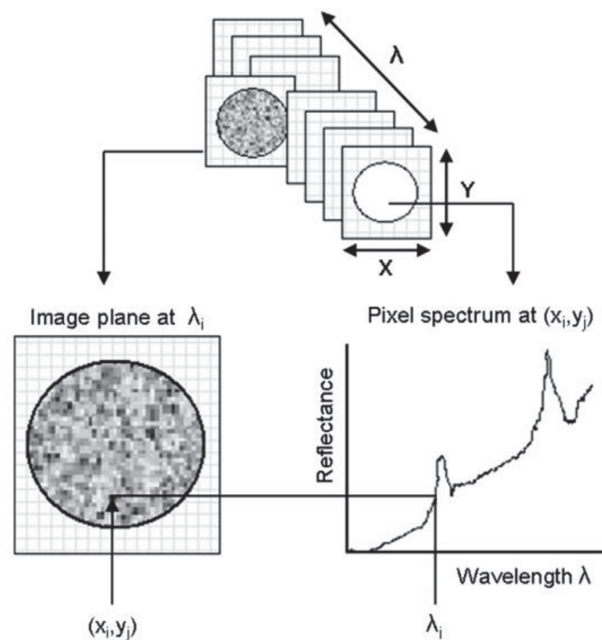


Figure 1: Hyperspectral reflectance profile illustration of Burger and Gowen [7]. Among the bands captured by the sensor at different wavelengths λ, we choose the band with the wavelength λ_i and observe the reflectance profile for the pixel at the position (x_i, y_i). The image plane at λ_i translates into exactly one dot in the reflectance profile. The hyperspectral image cube is comprised of two spatial dimensions (X and Y) and one spectral dimension (λ) [7].

Rather than seeing an object with its spatial dimensions (length, width, height) like the human eye does (e.g., a house), the goal was to pick up the wavelength response of the composite (e.g., multiple houses, including the materials in between) so that the differences in these responses could be used to distinguish between classes, assuming different objects have different spectral responses. This measurement fact that the spectral responses were used to identify a pixel meant that spatially adjacent pixels did not need to be measured, which mitigated the problem of too much data: data volume scales quadratically with the spatial dimension, but only linearly with the number of spectral bands, so as Landgrebe continues to explain, due to "reducing the spatial resolution while increasing the number of spectral bands, the data volume is greatly reduced" [15]. Over time, these multispectral sensors have advanced, eventually becoming suitable for hyperspectral measurements: as Zhang et al. assessed, the "increase of the spectral resolution" has shaped the development of remote sensing [19]. The difference between these methods is that hyperspectral imaging "uses **continuous and contiguous ranges** of wavelengths (e.g., 400 - 1100nm in steps of 0.1nm)", while "multispectral imaging (MSI) uses a **subset of targeted wavelengths** at chosen locations (e.g., 400 - 1100nm in steps of 20nm)" [20], although Hagen and Kudenov make the critical comment that an "inconsistent use of terminology" does exist in the field of spectral imaging, including hyperspectral and multispectral imaging being used interchangeably or distinguishing by the number of spectral bands captured [21]. Due to these fine-grained steps for the hyperspectral case, reflectance profiles can be approximated in the form of a graph as we saw in Figure 1, whereas we would have seen gaps for the multispectral technique. On top of that, the wavelength range captured by the hyperspectral or multispectral system can be varied beyond the visible spectrum of roughly 380-760nm [22], which would have been captured in three channels for RGB images at appropriate wavelengths for the colors red, green and blue. For example, spectral measurements for the near-infrared range (750-2500nm [22]) have proven to be helpful for face recognition as the subsurface tissue detected varies significantly from person to person [23]. As for ultraviolet remote sensing (10-400nm [22]), "[m]any important discoveries of the composition, energetic and dynamics of the thermosphere and ionosphere have been made using UV observations" [24].

Hyperspectral imaging (HSI), also known as chemical or spectroscopic imaging, is a technique which combines imaging and spectroscopy to gather both spatial and spectral information from an object [7]. Initially developed for remote sensing applications, HSI application areas range from agriculture monitoring, eye care and food processing to mineralogy, surveillance and the environment [20]. As pointed out in 1.1, HSI does have advantages over RGB imagery, which makes researchers choose HSI for the areas above. For example, Coulter et al. praised HSI as being ready to become "one of the most significant new technologies to see broad acceptance by the exploration industry" due to its usefulness in the "identification and mapping of minerals directly associated with alteration and mineralization over exposed ground", which they claimed no other airborne method provided, and the increased affordability and availability of HSI [25]. HSI also provides benefits for cancer detection and diagnosis because infected cells have a characteristic spectral profile, which can be used to distinguish them from normal cells [26]. For agriculture and vegetation, Thenkabail et al. attribute the "improved and targeted modelling and mapping of specific agricultural characteristics" to HSI [3]. Continuing with HSI's advantages, Park and Lu state the limitations of conventional imaging as generally "not suitable for detection or assessment of intrinsic properties and characteristics of products, whether they are physical and/or chemical (e.g., moisture, protein, sugar, acid, firmness or hardness, etc.)", positively referring to HSI as an improved imaging technology in food and agriculture [4]. For this thesis, though, we are going to stick with HSI for remote sensing since the datasets we will deal with (see Figure 22) have been captured by hyperspectral airborne sensors like AVIRIS and ROSIS. As we speak, the DESIS sensor (abbreviation for *DLR Earth Sensing Imaging Spectrometer*) is being used for hyperspectral environmental and resource monitoring from space, following "the robotic integration of the DESIS instrument to the [ISS-]MUSES platform in August 2018", a commissioning and validation phase as well as the start of the operational phase in early 2019 [27], capturing 235 spectral channels with a spectral resolution of 2.55nm "in a spectral range of 400nm up to 1000nm" [28], which reinforces the recency of the topic of hyperspectral remote sensing, on which our compression-related experiments are based. Several alternatives of sensors for hyperspectral remote

sensing like “Hyperion, TianGong-1, PRISMA, HISUI, EnMAP, Shalom, HyspIRI, HypXIM”, CHRIS, HSI, HSA, GISAT, HYSI and FLORIS exist [29].

The capturing process by hyperspectral equipment can be performed based on different ideas. Qin et al. depict the limitations of the conventional optical sensing techniques of imaging and spectroscopy since the former “cannot acquire spectral information” and the latter “cannot cover large sample areas”, whereas HSI as the combination of both components is described as “well suited for safety and quality evaluation of food and agricultural products”, illustrating the methods of *point-scan*, *line-scan* and *area-scan* and their respective suitability as HSI acquisition methods to arrive at a hyperspectral cube [2]. As an alternative for distinguishing between scanning methods, Lu and Fei describe the categories of *spatial scanning*, which means “acquiring a complete spectrum for each pixel [...] (point-scanning) or line of pixels [...] (line-scanning)”, followed by a spatial scan “through the scene”, and *spectral scanning* (area-scanning), which captures the whole scene first and only then completes the data cube through wavelength traversal [30]. Grusche combines these two methods into a method for “diagonal sections of a hyperspectral datacube”, which he calls *spatiospectral scanning* [31]. Alternatively, one could use snapshot techniques instead of scanning as outlined by Hagen and Kudenov, who also point out how both scanning and snapshot imaging have their up- and downsides [21]. For example, they denote the higher light throughput of snapshot imaging as the *snapshot advantage*, but computational complexity and high equipment cost are the downsides [21]. On the other hand, from our experience, line-scanning cameras, which concatenate spatial lines until eventually, a hyperspectral cube is formed, are very common for hyperspectral data acquisition in remote sensing, and also beside remote sensing, as Park and Lu claim, “line-scan based hyperspectral imaging systems have been used for several applications” [4]. Cao et al. describe and illustrate the sampling processes for different multispectral cameras (of which we have just presented four types), which involve shearing, modulation and/or projection, pointing out their reliance on the *Nyquist-Shannon sampling theorem* [32]: as Sun et al. describe based on Oppenheim et al.’s book [33], signals “cannot be recovered exactly unless the sampling frequency [is] twice or more than the maximum frequency in the original ones” [34]. Due to the inefficiency of the Nyquist acquisition method for spectrum processing, progress has been made in the field of *compressed sensing* so that “signals can be recovered from a small number of samples or measurements if they are sparse” [34, 35].

There are several purposes for which HSI can be used. According to Burger and Gowen, “[c]lassification and quantification of the material present in a scene are usually the main goals of HSI analysis” [7]. They go on to describe that “[c]lassification of hyperspectral images aims to identify objects of similar characteristics using the spectral and spatial information contained in the hypercube” [7]. We are indeed going to use HSI for classification in this thesis, but to know how this relates to other HSI goals, we will present the functional taxonomy defined by Chang shown in Figure 2 [36]. It depicts the five goals discrimination, detection, identification, quantification and classification in a Venn diagram, thereby describing the subset relations that apply.

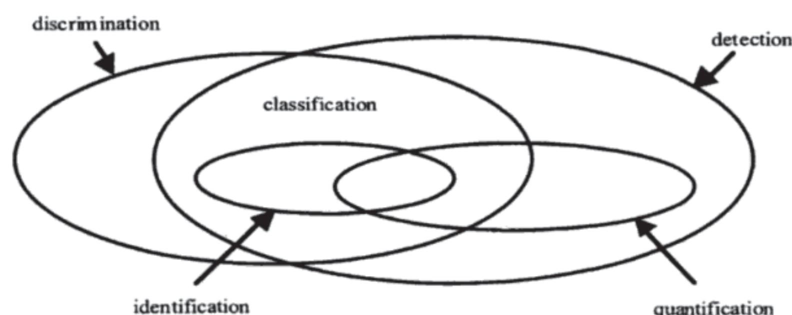


Figure 2: Functional taxonomy of HSI analysis goals defined by Chang [36]. Classification is portrayed as the intersection of discrimination and detection. Together with quantification, it is a common aim of HSI analysis [7]. Target classification is more specific than detection because a detector “may not be able to classify the targets it detected” (e.g., an anomaly detector). On the same note, a target classifier which classifies vehicles as wheeled or tracked is not necessarily able to identify a jeep and a truck, which shows the difference in granularity between classification and identification [36].

A challenge of HSI apart from the cost factor for acquiring the data and the large data size is the **abundance of data** gathered through airborne imaging spectrometers. The "high volume and complexity of data acquired with hyperspectral instruments" [19] poses a challenge to a performant classification with a low estimation error [15]. To make matters worse, the *Hughes phenomenon* tells us that "recognition accuracy can first increase as the number of measurements made on a pattern increases, but decay with measurement complexity higher than some optimum value" [37]. This is a specialization of the *curse of dimensionality* for machine learning, which means that data distribution becomes sparse with increasing space dimensionality [13], and especially true for remote sensing, where the "low spatial resolution limits the number of training samples" [38], just like the *ground truth data*, which Kondermann describes as "the results the vision system should yield" [39], is limited and expensive to gather [15]. Fortunately, the data of a hyperspectral image cube often "tends to be highly correlated in both the spectral and spatial domains" [7], which means chances are we do not need all bands, so we can try to reduce the dimensionality and with that, the band redundancy and correlation. In fact, it is common practice that after the preprocessing of a hyperspectral dataset, a dimensionality reduction operation is applied [6], which we will dive into in chapter 3. For our experiments, we will simply take the fact that hyperspectral datasets exist, which have been previously captured amid various challenges, for granted.

Several methods exist for HSI classification. *Supervised methods* such as neural networks and SVMs deal with labeled data, i.e. the hyperspectral dataset with its ground truth map, while *unsupervised methods* like k-NN do not the data to be labeled [7]. We will mainly deal with CNNs, a subclass of neural networks with at least one convolutional layer, but also include SVMs, k-NN and an RNN in our measurements to find out how well these classifiers perform in the experiments. Therefore, we will now present the basics needed to understand CNNs for our experiment and how they work on hyperspectral datasets.

2.2 Convolutional Neural Networks

Since we will mainly use CNNs for pixel-wise HSI classification in the experiments, we are now going to introduce the concept of a CNN, i.e. what it consists of (2.2.1), how the training process works (2.2.2) and which techniques exist to try to gain a semantic insight into how the network functions (2.2.3). We will also present other neural network variations like RNN and alternative methods for classification like SVM, which are relevant for our case, since a CNN is just one example of an instrument that can be used for (HSI) classification purposes (2.2.4). Of course, there are also CNNs that are successful at RGB classification tasks [40], and any classification strategy for an RGB dataset will also work for an HSI dataset assuming that the wavelengths of RGB are among the ones captured in the hyperspectral spectrum, because in that case, HSI accounts for at least as much information: if all else fails, a conversion of the dataset from HSI to RGB using the appropriate wavelengths of RGB can be used to only work with the data at the three needed wavelengths and apply the RGB strategy. In case the RGB wavelengths have been skipped by the hyperspectral sensor, one might estimate the values for RGB based on the two closest surrounding wavelengths captured where one lies above and one below the targeted wavelength, possibly coming to the conclusion that the claim also holds, depending on the quality of the estimate. The statement is not necessarily applicable the other way around because strategies that use "the inherent properties of HSI data" [41], which are not necessarily present in RGB data, to say the least, exist (cf. 2.3.1, Figure 20 and Figure 21).

2.2.1 Architecture

A *convolutional neural network* (CNN) is an *artificial neural network* (ANN) with at least one convolutional layer. Therefore, we are going to explain the more general term ANN first, although for our HSI classification task, we will primarily deal with CNNs because of their better suitability for this task [42], which we are going to elaborate on later.

An ANN is a computing system "inspired by the architecture of biological neurons such as the human brain", which is typically not programmed, but can be used for tasks such as image recognition by **learning** to identify images [43]. In fact, learning is the "key characteristic of a neural network", which is especially

useful when no known mathematical model exists that describes the underlying dataset, since "a neural network may discover interesting relationships as it rambles through the database" [43]. Mathematical processes based on numerous parameters enable this learning process (see 2.2.2). Structurally, the building blocks of an ANN are *neurons* organized in *layers*. These layers, of which we can distinguish between one *input layer*, one or multiple *hidden layers* and one *output layer*, are connected to one another. This means that all neurons from one layer are connected to all neurons in both the previous and the subsequent layer. Associated with each such connection is a *weight*. The intuitive explanation is that the higher the magnitude of the weight, the more important the connection [44].

When we want an ANN to perform, e.g., an image classification task, the input layer of the ANN receives the image in its input layer, and the result is produced in the output layer. Receiving the image means that different (hyperspectral) patches of the image of a user-defined input shape with the distance of a user-defined stride to one another are connected to different neurons in the input layer. The neural network needs to perform calculations and propagate them to the output layer for the result to be visible. For the calculations, the weight of a connection is multiplied with the input it receives, which is done for all connections from the input layer to form a *weighted sum*, which includes an optional neuron-specific *bias* that may be added [45]. At this point, to be able to **approximate different functions** through the learning process, we need to introduce *non-linearity* to the ANN. If we did not do that, no matter how many layers we used, the ANN would have the severe restriction of only being able to approximate linear functions. This is why we use an *activation function* σ, of which common choices include the sigmoid function $\sigma(x) = \frac{1}{1+e^x}$, the hyperbolic tangent function $\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ and the rectified linear unit function $\sigma(x) = \max(0, x)$ abbreviated as ReLU. If we consider the neuron k, name the weights from the incoming connections w_{ki} for the i-th connection of m total incoming connections and the neuron's bias b_k, the weighted sum shown in Figure 3 and fed into the activation function will be $v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k$, while the output y_k for the neuron at the other end of the connection, which will act as its input signal, will be $y_k = \sigma(v_k)$.

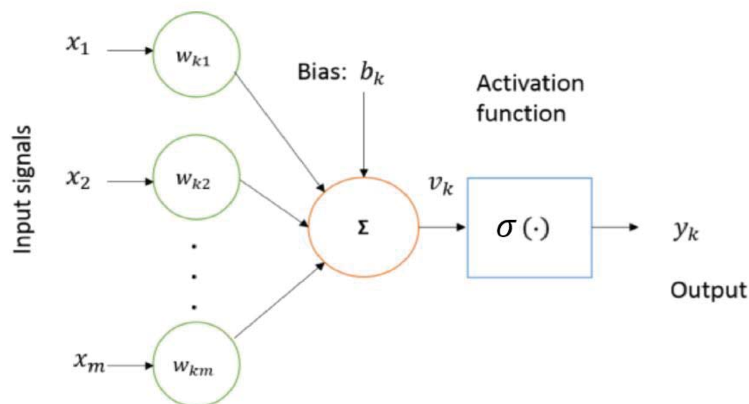


Figure 3: Mathematical illustration of how the output of a neuron propagated to the connected neuron in the subsequent layer is calculated. For the k-th neuron and $i = 1, \ldots, m$, this involves calculating the weighted sum v_k of weights w_{ki} from input signals x_i, adding the bias b_k and using the activation function σ to obtain the output $y_k = \sigma(v_k)$. Figure adjusted from [42].

The example layer we took for explanation purposes of these basic calculations is called a *fully connected layer*, also often called a *linear layer*, where every input neuron is connected to every output neuron. Consequently, if we have m input and n output neurons, there are going to be $m * n$ connections, so adding such a fully connected layer in this circumstance would increase the number of parameters of the network (and thereby the network's complexity) by $m * n$ weights, in addition to the $m + n$ biases we would have already had for the $m + n$ neurons without connections in between. A neural network having many parameters generally means a slower training process and often enough, we do not actually need the parameters to obtain good results (cf. 3.3.2 and 9.1.1), so it is desirable to reduce their number through model compression methods (see 3.2.1).

A layer used to reduce the input dimensionality is the *pooling layer*. Depending on the type of pooling used, its purpose is to compress or even discard redundant information. Max pooling is a technique where within a given filter size (e.g., 3×3, which means we chose a two-dimensional filter), only the largest value is selected as the one output number for the considered part of the input to the layer. This filter then moves depending on the *stride* specified so that the next number can be calculated like that, which is repeated until the input is fully traversed by the filter, with additional values enlarging the input as a *padding* if necessary, for which several options like padding with zeros or with the same values are choosable. Springenberg et al. claim that a max pooling layer can be replaced by a convolutional layer with increased stride without loss in accuracy [46]. Average pooling averages all numbers within the filter of the user-defined size on the layer input, takes this number as the output and the filter moves on according to the stride until all parts of the input patch are covered. It has proven to be less efficient and is more rarely used than max pooling [47].

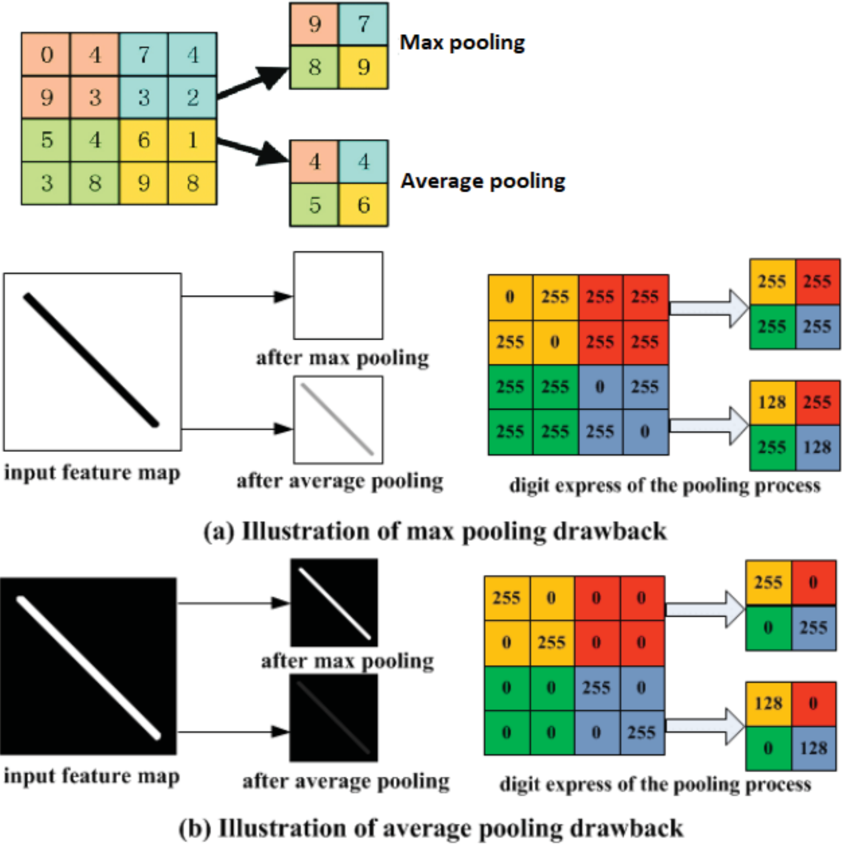


Figure 4: Illustration of max and average pooling with their challenges. If we choose the wrong pooling type, the line, whose grayscale values we see in the matrices, ceases to be visible. Yu et al. propose a mixed pooling in their paper as an alternative to max and average pooling and one of many pooling implementation types, which is a "stochastic procedure which randomly employs the local max pooling and average pooling methods when training CNNs" [48, 49]. Figure combined from [48] and [49].

There is one more important type of layer we need to present, which is the *convolutional layer*. In a convolutional layer, "feature maps from previous layers are convolved with learnable kernels", producing a kernel output that may be activated with an activation function to form the output feature maps [42]. In practice, we can think of a convolutional layer as a dot product operation because the values of the filter, whose size and dimension we decide, are multiplied with the input patch at the respective positions and summed up to arrive at one output value of many, for which the process is repeated by moving the filter according to the stride and performing the calculation for the new positions once again. Mathematically speaking, we can calculate the outputs of a convolutional layer with the formula

$$x_j^l = \sigma\left(\sum_{i \in M_j} x_i^{l-1} * k_{ij}^l + b_j^l\right)$$

where x_j^l is the output of the current layer we call l, x_i^{l-1} is the previous layer's output, k_{ij}^l is the kernel for the present layer (i.e. elements of the convolution matrix), b_j^l are biases for the current layer and M_j represents a selection of input maps [42]. Variants such as tiled convolution ("learn rotational and scale invariant features"), dilated convolution ("increase the network's receptive field size") and deconvolution (single input activation affects multiple output activations) exist, as Gu et al. describe in their paper about recent CNN developments [50]. CNNs with only convolutional layers are called *fully convolutional networks* (FCN).

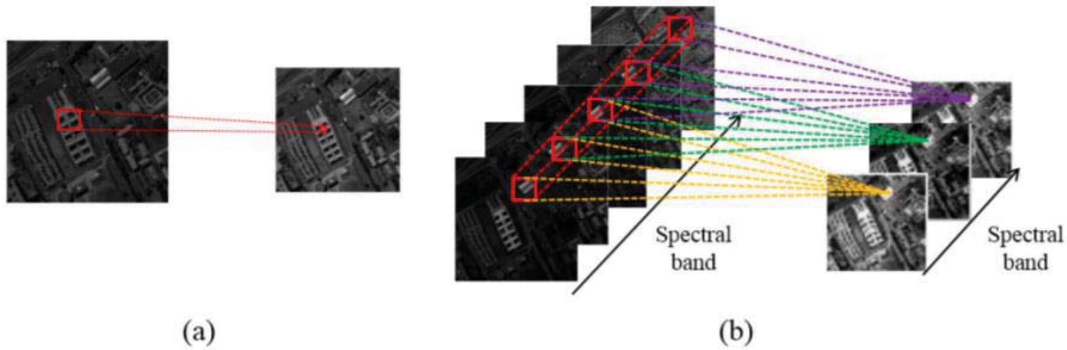


Figure 5: Li et al.'s illustration of 2D (a) and 3D (b) convolution operations (left-hand side: before convolution; right-hand side: after convolution). The 2D convolutional kernel convolves the input so that spatial features are captured. In contrast, 3D cubes are used for 3D convolutional operations in an effort to extract spatiospectral features [51].

CNNs have many advantages over ANNs, especially for our task of hyperspectral image classification. They are more similar to the human visual processing system, structurally highly optimized for image processing and effective at learning and extracting feature abstractions [42]. Shape variations are accounted for by max pooling layers [42]. Moreover, **HSI features are nonlinear** (due to physical laws of "radiative transfer and sensor properties and calibration" [18, 52, 53]), **discriminant and invariant**, and according to Chen et al., **a combination of convolutional and max pooling layers can extract** these useful features for image classification and target detection [54]. It is also true that a CNN has much fewer parameters than an ANN of the same size due to the "sparse connections with tied weights" [42], which is the reason why Li et al. even claim that for a hyperspectral image as the input (with hundreds of image bands instead of just three if we jump to Figure 22), "[t]he full connection of neurons between two adjacent layers is infeasible" [55], so **CNNs are of special importance for our HSI classification task**, much more so than ANNs. Hu et al. claim that the "hierarchical architecture of CNNs is gradually proved to be the most efficient and successful way to learn visual representations" [56]. If we wanted to confirm the higher structural complexity of an ANN compared to a CNN, we would calculate the number of parameters, for which there are formulas available [42]. They are a good metric to use for model compression techniques such as parameter pruning to determine the current pruning percentage, which we did. For example, for the layer l, the number of output feature maps FM_l, the number of input feature maps FM_{l-1} and the filter size F, the formula Alom et al. used to calculate the number of parameters P_l (weights and biases) is

$$P_l = (F \times (F + 1) \times FM_{l-1}) \times FM_l$$

As an example, for 32 input feature maps, 64 output feature maps and the filter size 5, we will have $P_l = (5 \times (5 + 1) \times 32) \times 64 = 61{,}440$ parameters for this layer [42].

Taking a look back in time shows us how CNNs have developed to become such a popular choice for both HSI and RGB image classification. Khan et al. divide the rise of CNNs into five phases [40]. First come the

CNN origins in from the 1980s until 1999, which date back to Fukushima in 1988, who first proposed a CNN network structure [57], but LeCun et al.'s ConvNet from 1989 applied on the handwritten digit classification problem and relying on backpropagation as the training algorithm (see 2.2.2) must also be mentioned as the first multilayered CNN [58]. In early 2000, Support Vector Machines surpassed CNNs in popularity due to their good performance [59], but CNNs were already being rolled out for customer tracking [60] and tested for medical image segmentation, anomaly detection and robot vision [61-63]. In 2007, NVIDIA's launch of the CUDA platform [64, 65] enabled GPU usage for neural network training, which greatly boosted CNN research as it alleviated the previously very present and serious problem of long training times of weeks to months [66, 67]. With the performance breakthrough brought by AlexNet in 2012 [1] suggesting that the lack of enough training data and appropriate computational resources were the main reasons for a stagnant CNN performance before 2006 [42], CNN architectures have become deeper, and especially since 2015, numerous new architectures and applications of CNNs have been proposed (as presented already) [40].

Finally, whereas for the RGB case, numerous CNNs exploiting *spatial* features such as LeNet, AlexNet, ZefNet, VGG and GoogleNet exist [40], the CNN application to HSI allows for the exploitation of *spectral* features on top of that. In fact, this potential to use *spatiospectral* (combination of spatial and spectral) features of the image to the advantage of the image classification process is frequently used by 3D CNNs [14, 51, 68-71], where the "3D" stands for the highest dimensionality of a convolutional layer encountered in the respective CNN. By contrast, 1D CNNs are known to only consider spectral features and 2D CNNs are often used for spatial features [72], as the following figure illustrates. As a consequence of using a hyperspectral dataset for pixel-wise classification, the objects of interest called pixels are not just points with red, green and blue components as in the RGB case, but a hyperspectral pixel spans all spectral bands captured, which makes it considerably larger due to usually hundreds of components [11].

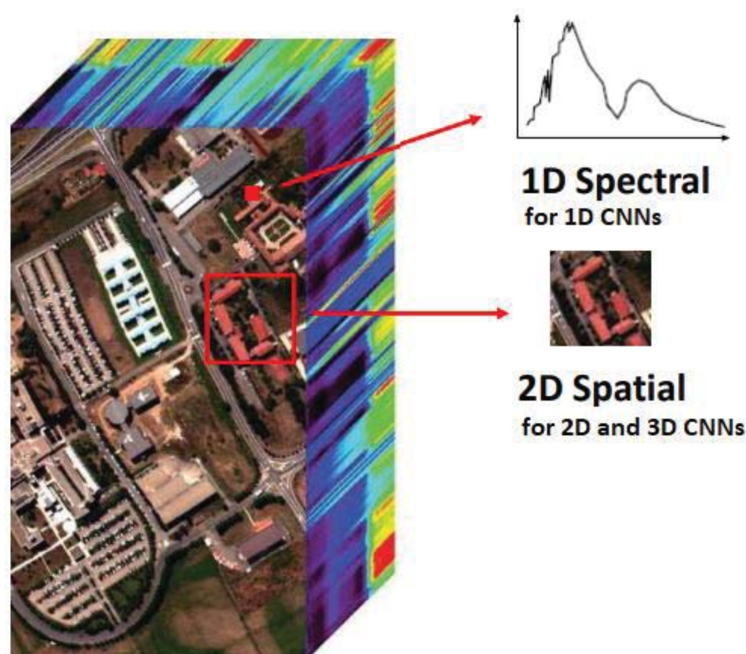


Figure 6: Hyperspectral image cube for the Pavia University scene to illustrate the hyperspectral input patch for a CNN, depending on the architecture used, adjusted from [38]. 1D CNNs focus on spectral features with their 1D convolutional layers, so they are only given a one-dimensional hyperspectral pixel of a depth as an input, the size of which is equal to the $number_of_bands$ *(this will translate to a two-dimensional tensor in PyTorch). Conversely, 2D spatial input patches for all bands of the combined size of* $width \times height \times number_of_bands$ *are used by 2D CNNs (for spatial feature recognition) and 3D CNNs (with the potential of spatiospectral exploration of the input patch with 3D convolutional kernels), which will both mean five-dimensional tensors in PyTorch as we have seen in DeepHyperX [38].*

Now that we have taken care of the topic of CNN layers, one would wonder how to arrive at a well-performing configuration of multiple layers, i.e. how to design a neural network. As of now, this problem

of designing one accepted neural network from the scratch given a particular problem is a research question to which there is no clear answer, but there are attempts to **systematically search** for a well-performing neural network based on parameters we would like to vary (number of nodes, kernel size, activation function etc. – or, on a more coarse-grained level, how many layers to choose, which ones and in which order) according to the criteria we choose (e.g., if a structurally similar CNN performed well on a dataset, the currently considered slightly modified variant might be worth a try, otherwise we could consider skipping it). Smithson et al. attempted to programmatically generate a working network configuration using a design space exploration algorithm on the MNIST and CIFAR-10 datasets, where the design space is comprised of the numbers of the numbers of linear and convolutional layers, the numbers of nodes in each layer, the convolution kernel sizes, the max pooling sizes, the activation function and the network training rate in 2016 [73]. Last year, at least two frameworks for neural network architecture search were published, which were DPP-Net, which considers the parameters we want to optimize for neural network search (device-related: inference time, memory usage; device-agnostic: accuracy, model size) [74], and MONAS, which uses reinforcement learning to search with the goal of a high prediction accuracy "and other important objectives (e.g., power consumption)" [75]. Moreover, in May 2019, Wistuba et al. at IBM Research AI published an in-depth review of the current proceedings regarding neural architecture search, distinguishing between the four categories "reinforcement learning, evolutionary algorithms, surrogate model-based optimization, and one-shot models", without directly comparing the results to avoid uneven comparisons as the papers followed different objectives and approaches [76]. This is too much to explain here, but schematically, Figure 7 shows the first three concepts which we now know can be used for neural network generation. Since the search space can be big, Li et al. **pruned** "**the architecture search space** with a partial order assumption to automatically search for the architectures with the best speed and accuracy trade-off" [77]. We expect more progress to be made in this rather novel research field of architecture search. In the meantime, the decision whether to add or remove a layer comes down to trial-and-error in practice [78], apart from advice that comes from experience, e.g., that one typically chooses neural network architectures that have worked for a problem already, takes the model parameters from a checkpoint file (this is called *transfer learning*) and starts to vary the architecture from there (including retraining to confirm or refute the allegedly better performing newly formed architecture), that deeper models usually perform better than wider models, that 3×3 and 1×1 "usually work the best", where 1×1 convolutions are well-suited for dimensionality reduction purposes and a lot more information about which values are well-suited for *hyperparameters* according to Isikdogan's neural network design experience [79]. These are parameters set prior to the start of the learning process [43].

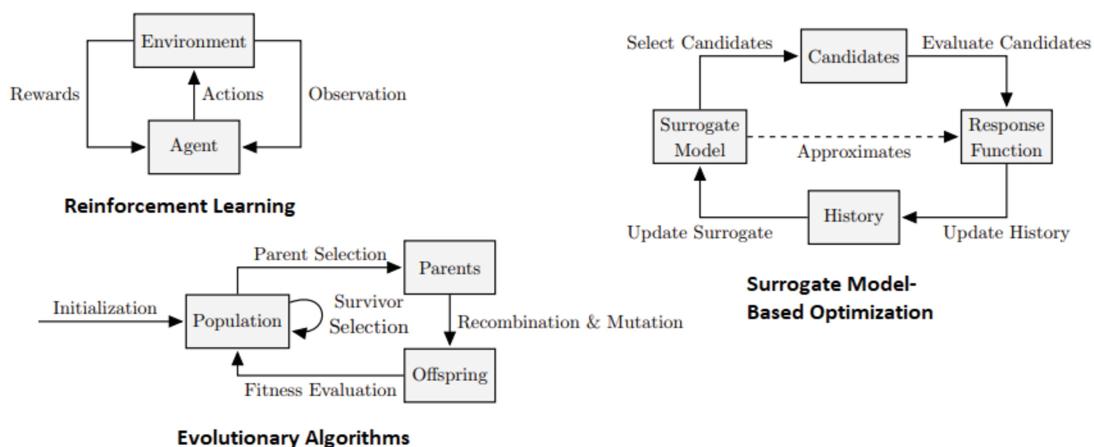


Figure 7: Three processes used for the task of exploring a neural network architecture are shown in Wistuba et al.'s illustrations [76]. For reinforcement learning, the agent aims to maximize its reward, i.e. a high accuracy, by choosing a neural network, making an observation from the environment and adjusting accordingly. Surrogate models act to minimize a cost function (often least squares) so that a promising candidate is chosen, evaluated, and the surrogate updated, which is repeated until a convergence criterion is reached. In the evolutionary algorithm context, parents from a neural network population are selected, mutated and evaluated or dropped, depending on the performance reached [76].

2.2.2 Learning Process

After randomly initializing the weights and biases of the neural network, one can feed an image to the neural network and receive an output in the output layer. Expectedly, these activations of the neurons in the output layer will not be identical to what one would expect based on the labels for the data. Consequently, we need to find a way to tell the neural network **how to improve** for the classification task. For that, we can define a cost function ε, better known as a *loss function*: for a set of inputs and corresponding outputs (x_t, y_t) and the output of the network $\hat{y}_t \coloneqq f(x_t)$, the loss function $\varepsilon(y_t, \hat{y}_t)$ may be calculated, for example, using the sum of squares. If we name the components of the vectors $y_t = \begin{pmatrix} y_t^1 \\ \vdots \\ y_t^n \end{pmatrix}$ and $\hat{y}_t = \begin{pmatrix} \hat{y}_t^1 \\ \vdots \\ \hat{y}_t^n \end{pmatrix}$ for n output layer neurons, this means that $\varepsilon(y_t, \hat{y}_t) = \sum_{i=1}^{n} \left(y_t^i - \hat{y}_t^i\right)^2$. This loss function allows to give the neural network a form of feedback for the feedforward pass of x_t.

Training a neural network is the process of adjusting its weights and biases with the goal of minimizing that cost function. As Gu et al. put it, "[t]raining CNN is a problem of global optimization. By minimizing the loss function, we can find the best fitting set of parameters" [50]. Finding local minima of such a cost function is what the *gradient descent* algorithm does. It works by repeatedly finding the direction of the steepest descent for the cost function, for which its gradient regarding a variable θ is computed, which stands for the neural network parameters (weights and biases). Due to the long training time, s*tochastic gradient descent* (SGD) is commonly used for training neural networks instead [80, 81], where the dataset (X, y) is split into batches (x_i, y_i). For both algorithms, a convergence criterion is used when θ does not improve enough anymore to justify the further iterations according to the criterion to stop the iterative process. Variants of SGD such as parallel SGD [82], lock-free parallel SGD [83] and SGD with predictive variance reduction [84] exist, but overall, plain SGD is still very common to use [81].

Inputs: loss function ε, learning rate η, dataset X, y and the model $\mathcal{F}(\theta, x)$

Outputs: Optimum θ which minimizes ε

Gradient descent (training data size N)	**Stochastic Gradient Descent (SGD)** (batch size N)
Repeat until converge: $\tilde{y} = \mathcal{F}(\theta, x)$ $\theta = \theta - \eta * \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \varepsilon(y, \tilde{y})}{\partial \theta}$ **End**	**Repeat** until coverage: Shuffle X, y; **For** each batch of (x_i, y_i) in X, y **do** $\tilde{y}_i = \mathcal{F}(\theta, x_i)$ $\theta = \theta - \eta * \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \varepsilon(y_i, \tilde{y}_i)}{\partial \theta}$ **End**

Figure 8: Side-by-side comparison of the gradient descent and SGD algorithms, adjusted from [42]. The fact that SGD operates on batches instead of the entire training dataset significantly improves the runtime, which is why it is more common to use [81]. Because of these different algorithm philosophies, the meaning of the N parameter varies.

Looking at Figure 8, we see the variable η as the *learning rate*. As we can already infer from the formula, the learning rate is a step size. As Alom et al. state, it needs to be chosen with caution because on the one hand, a large η might make the network diverge as opposed to converge, on the other hand, a small η makes the process take longer and the network prone to be stuck in local minima [42]. To avoid being stuck in local minima, the learning rate is often lowered during training [80]. The common variants of doing this are the constant variant, where a manually defined step function is subtracted from η, the

factored variant, where η is multiplied with a decay factor in the $(0;1)$ range, and an exponential decay [42].

Neither gradient descent nor SGD propagate the changes through the hidden layers from the output layer to the input layer. This is what *backpropagation* does, relying on SGD for adjusting the gradients for each layer. It works because it applies the chain rule for calculating derivatives for both weights W_l and biases b_l: if we consider $\boldsymbol{h_l} = \sigma_l(W_l^T h_{l-1} + b_l)$ the output of the hidden layer l (where the superscript T denotes the transposition of the weight matrix, with b_l being a bias vector), we can calculate h_{l+1} like that:

$$h_{l+1} = \sigma_{l+1}(W_{l+1}^T \boldsymbol{h_l} + b_{l+1}) = \sigma_{l+1}(W_{l+1}^T \sigma_l(W_l^T h_{l-1} + b_l) + b_{l+1}) = f(g(h_{l-1}))$$

As a result, we now see that the chain rule $\frac{\partial y}{\partial x} = \frac{\partial f(x)}{\partial x} = f'\big(g(x)\big) * g'(x)$ applies to neural network backpropagation, whose specifics are presented in the following algorithm [42].

Backpropagation algorithm

Input: A network with l layers, the activation function σ_l, the loss function ε, the outputs of hidden layer $h_l = \sigma_l(W_l^T h_{l-1} + b_l)$ and the network output $\tilde{y} = h_l$

Compute the gradient: $\delta \leftarrow \frac{\partial \varepsilon(y_i, \tilde{y}_i)}{\partial y}$

For $i \leftarrow l$ to 0 **do**

Calculate gradient for present layer: $\frac{\partial \varepsilon(y,\tilde{y})}{\partial W_l} = \frac{\partial \varepsilon(y,\tilde{y})}{\partial h_l} \frac{\partial h_l}{\partial W_l} = \delta \frac{\partial h_l}{\partial W_l}$ and $\frac{\partial \varepsilon(y,\tilde{y})}{\partial b_l} = \frac{\partial \varepsilon(y,\tilde{y})}{\partial h_l} \frac{\partial h_l}{\partial b_l} = \delta \frac{\partial h_l}{\partial b_l}$

Apply gradient descent using $\frac{\partial \varepsilon(y,\tilde{y})}{\partial W_l}$ and $\frac{\partial \varepsilon(y,\tilde{y})}{\partial b_l}$

Backpropagate gradient to the lower layer: $\delta \leftarrow \frac{\partial \varepsilon(y,\tilde{y})}{\partial h_l} \frac{\partial h_l}{\partial h_{l-1}} = \delta \frac{\partial h_l}{\partial h_{l-1}}$

End

There are variants of backpropagation like Bayesian backpropagation based on heuristic techniques [85], stochastic backpropagation to define "rules for gradient backpropagation through stochastic variables" [86] and RPROP, whose specialty is that the "adaptation process is not blurred by the unforeseeable influence on the size of the derivative but only dependent on the temporal behavior of its sign" [87]. Vora et al. have compiled a survey of the backpropagation variations which have been tried in research so far in a chronologically ordered table [88], where we witness a rising importance of integrating genetic algorithms into backpropagation in the most recent works, i.e. of Yeremia et al. and Rajasekaran et al. [89, 90].

As the backpropagation algorithm relies on adjusting weights and biases based on partial derivatives of the loss function, challenges arise when the gradients become vanishingly small as this leads to a very slow training process for the first layer, if it is even being updated at all. This is because the earlier the layer is placed in the network, the more tiny gradients are multiplied (exponentially so, because subtracting one layer means one additional gradient factor) and the more severe the effect of this *vanishing gradient problem* becomes. It is especially prevalent for $ReLU(x) = \max(0, x)$ as the gradient is always one or zero. Once the latter case occurs during backpropagation, previous layers do not learn at all (so-called *dead ReLU*), which is why ReLU variants such as *leaky ReLU* exist, which return a small positive gradient for $x < 0$, e.g., $0.01x$ [91]. Alternatively, the *batch normalization* proposed by Ioffe et al. [92] can accelerate the learning process "by reducing internal covariance [through linear transformation of] input samples" so that they have "zero mean and unit variance" [42] (many other solutions, e.g., using long short-term memory or residual networks exist, which we are not going to elaborate on).

Another challenge that arises with backpropagation is that because the SGD it relies on is an iterative process, we need multiple epochs for the training to arrive at weights and biases that result in a good accuracy. *Epochs* denote the number of times where the entire dataset is passed through the network, which is not to be confused with batches as these just refer to chunks of the data passed through the network. Too many epochs may lead to *overfitting*, i.e. the problem that the network does not learn to perform its task like image classification, but intuitively speaking, learns the training samples by heart, so that it is unsuitable to classify data it has never seen before. In mathematical terms, an overfitted model does not generalize well to new data because **regarding the bias-variance tradeoff, reducing the bias error has led to a high variance, i.e. low adaptability for new samples**, e.g., for the unknown samples which the model encounters in productive use. On the other hand, a high bias error is also not desirable since it means that no accurate estimate of the function to be approximated is produced (*underfitting*), so we aim for a happy medium and do not want either case to occur [93]. To lower the chance of overfitting, a dataset should be split into a *training, validation and testing* part (e.g., 60%-20%-20% or 80%-10%-10%). The difference between validation and testing is that validation is an intermediate result, which allows to see the impact of the training that has occurred so far, including early stopping if overfitting is detected, e.g., by an increasing error on the validation dataset [94], whereas testing is the final stage of evaluation after all training has ended [94], so it is the inference time on the testing dataset that can be chosen as a meaningful metric for a neural network. Moreover, Srivastava et al. introduced *dropout layers* to reduce overfitting, where a user-defined fraction of the neurons, which are randomly selected, are not considered by the neural network during training [95]. Max pooling layers also help against overfitting because due to their nature of pooling multiple patterns to fewer patterns, it is less likely that the network learns a pattern which it should not (see 2.2.1).

At the stage of testing, different metrics are available to compare neural networks. Apart from the inference time, RAM and VRAM usage, the accuracy metrics are probably most important because if the accuracy is bad, the network is likely not worth investigating further regardless of the other parameters, neither from the scientific nor from the industry standpoint. Different accuracy metrics exist for image classification. Su et al. used three commonly used [96] accuracy metrics, which we deemed appropriate for our experiment, i.e. *overall accuracy* (OA), *average accuracy* (AA) and the *Kappa coefficient* (Kappa): OA is the “percentage of correctly classified pixels” (the most general metric, which we referred to when just saying “accuracy” before introducing OA), AA the “mean of the percentages of correctly classified pixels for each class” (good for detecting difficult datasets, i.e. where many classes that are challenging to be detected for the classification task exist) and Kappa “the percentage of correctly classified pixels corrected by the number of agreements that would be expected purely by chance” (perceived as more robust than OA [97], but not as common to encounter as OA in research from our experience) [98].

2.2.3 Visualization

Once a CNN has been trained and reaches good accuracy, it is desirable to get an understanding of why it performs well. In particular, the question why after applying a network compression technique such as parameter pruning, the CNN performs the way it does and how that relates to its original state is interesting. One of the ways to reason about the feature detection processes in the different layers for the purpose of image classification is through visualization of intermediate layers. We will proceed to present three variants of generating *saliency maps* as well as an approach based on *maximally activating patches* for a chosen layer and briefly touch upon *guided backpropagation*.

The idea of saliency maps is that we want to tell which pixels of the input patch matter for the classification obtained. This is why we compute the gradient of the class score with respect to the image pixels and take the absolute maximum value over all image bands [99, 100]. It is the most basic type of saliency maps, but as we shall see in the experiments, it can give a good intuition for the explainability of the neural network as to how its behavior changes due to parameter pruning. More formally, Simonyan et al. start with a linear score model for a class c, an image I_0 in vectorized form, a weight vector w_c, a bias vector b_c and a score $S_c(I) = w_c^T I + b_c$ for a vectorized image I. Whereas for this linear model, the

magnitude of the weight elements obviously constitutes the importance of the image pixels for the class, the case of neural networks is more complicated than that due to the intentionally involved non-linearities, but we can linearly approximate the score function as $S_c(I) \approx w^T I + b$ with $w = \frac{\partial S_c}{\partial I}(I_0)$. Thus, we arrive at a class score we can visualize, whose magnitude indicates the pixels that "need to be changed the least to affect the class score the most" [100].

Specializations of this basic gradient-based saliency map algorithm exist, of which we are going to focus on integrated gradients and guided backpropagation (other methods like DeepLIFT [101], layer-wise relevance propagation (LRP) [102] and deconvolutional networks [103] exist). As a result of their exploration of important attributes for gradient-based visualization methods, Sundararajan et al. defined the sensitivity and implementation invariance axioms, which they used as a guidance to design their integrated gradients approach (which they formally defined at length in their paper, but including this would be disproportionate for this part of the thesis) so that it satisfies both criteria:

Sensitivity: "An attribution method satisfies [the sensitivity axiom] if for every input and baseline that differ in one feature but have different predictions, then the differing feature should be given a non-zero attribution" and "if the function implemented by the deep network does not depend (mathematically) on some variable, then the attribution to that variable is always zero." [104]

Implementation invariance: "Two networks are functionally equivalent if their outputs are equal for all inputs, despite having very different implementations. Attribution methods should satisfy [i]mplementation [i]nvariance, i.e., the attributions are always identical for two functionally equivalent networks." [104]

The difference of guided backpropagation compared to backpropagation is that **negative gradients do not flow backwards**. The idea is that neurons that decreased the higher layer unit activations do not do so anymore, as Figure 9 explains [46].

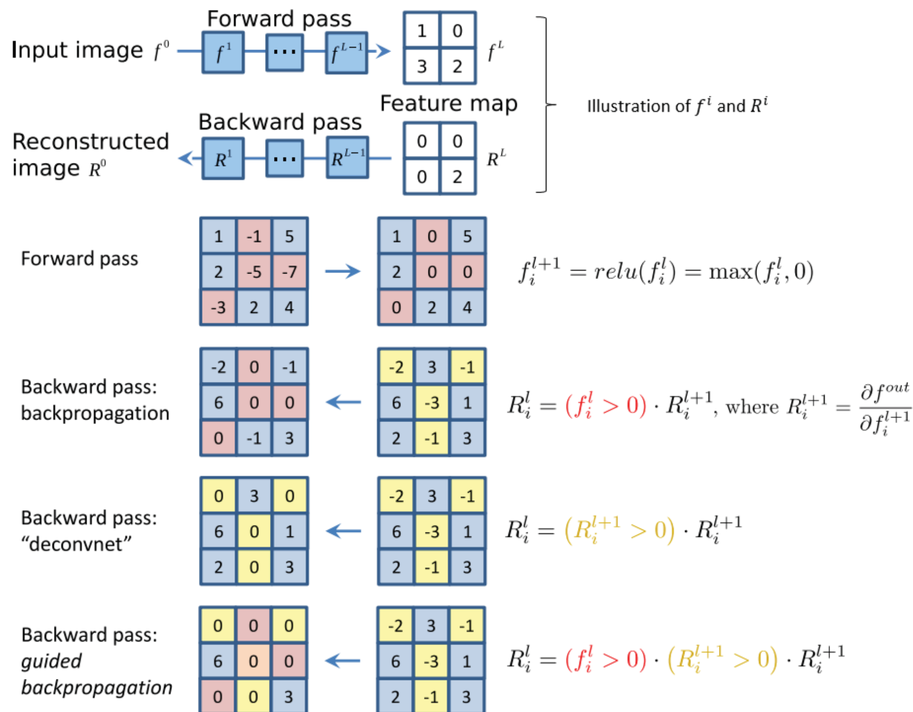


Figure 9: Illustration of different visualization methods for activations heavily inspired by, but arranged differently than [46]. Having defined what we mean by f^i and R^i, the forward pass and backpropagation should seem familiar (concept explained in 2.2.2). As new approaches, deconvnet only passes through positive values during the backward pass, while the backpropagation variant "guided backpropagation" additionally respects ReLU's zeros from the forward pass in the sense that they remain during the guided backpropagation process. For Springenberg et al., "guided backpropagation produces cleaner visualizations than the 'deconvnet' approach" [46].

Finally, we will move on to activation maps, i.e. the visualization of the **input patches that maximize the activations** of filters in a chosen layer. Erhan et al. rightfully formulate this problem as an optimization problem because to find out which input patch activates a chosen layer the most, an idea would be to iteratively modify the input patch to solve the problem of

$$x^* = \arg\max_x h_{ij}(\theta, x)$$

where a fixed θ denotes the weights and biases of the CNN (e.g., obtained through training the network) and h_{ij} is an activation function of the filter i in the layer j of both θ and the input sample x [105]. Nguyen et al. formulate the problem in a similar way, but they use a penalty function which ought to be minimized [106]. This iteration-based approach of modifying an image is highly reminiscent of the gradient descent during backpropagation, and indeed, we can apply *gradient ascent* in the input space to move x in the direction of the gradient $h_{ij}(\theta, x)$ to find a local minimum to this non-convex optimization problem [105]. In contrast to the gradient descent of backpropagation for minimization purposes (of the loss function), we are interested in maximizing a value this time around (which happen to be the activations), so we use gradient ascent. These patches x^* can be visualized, which is exactly what we did (cf. chapter 4.3.2).

Obviously, the presented visualization methods are not the only ones published so far. Bojarski et al. employ an approach they call visual backpropagation, which backpropagates entire sets of pixels as opposed to single pixels, and exploits the assumption that "feature maps contain less and less irrelevant information to the prediction decision when moving deeper into the network" [107]. Shrikumar et al.'s DeepLIFT relies on backpropagating activation differences between a reference and a contribution score of each neuron, which they claim fares better than gradient-based methods [101]. Many more works can be found in the visual analytics survey of Hohman et al. [108] and Alsallakh et al.'s paper concerned with neural networks learning class hierarchy, in which they point out that "[u]nderstanding the training behavior of CNNs [through visualization] helps in introducing targeted design improvements to large-class CNN classifiers" [109], granted that one has the necessary experience for designing neural networks. We also point out key visualization work in 2.3.2.

2.2.4 Alternatives

One of the neural networks in the experiment is a *recurrent neural network* (RNN) because it contains a recurrent GRU unit. RNNs are a subtype of ANNs that allows "operation over a sequence of vectors over time" as opposed to "a fixed number of computational steps" [42]. Unlike a *feedforward neural network*, they are capable of considering the outcome of previous computations of the same unit in their current computation, which is realized by using outputs – either from hidden layers (Elman architecture [110]), or from the output layer (Jordan architecture [111]) – as additional inputs to the network. In research, RNNs have been used for HSI classification tasks because **if one considers spectra as sequences, RNNs can take advantage of this sequential data** [112], unlike CNNs, which would merely consider the sequences as orderless vectors, ignoring the continuity of spectra [113]. As an outcome of their experiment, Yang et al. have come to the conclusion that a "recurrent network structure […] can effectively exploit both the spectral and the spatial contexts", which helps with faster convergence compared to other models like 2D or 3D CNNs [72]. Beyond HSI classification, RNNs are particularly well-suited for language understanding tasks [114], but can also be applied to generating video frames based on previous ones [115].

Motivated by the vanishing gradient problem, Felix A. et al. created a less susceptible RNN model called *long short-term memory* (LSTM) in 2000 [116, 117]. The RNN cell we use once in the experiments is called *gated recurrent unit* (GRU), which was created as a lightweight LSTM variation in terms of "topology, computation cost and complexity" [42, 118], whose functionality is shown in Figure 10. GRUs can be applied for HSI classification as Mou et al. have shown [113]. Hang et al. went further than just inputting entire spectral bands into RNNs without preprocessing by cascading GRUs "to explore the redundant and complementary information of HSIs", where the two layers have different purposes (eliminate

redundancies in adjacent spectral bands vs. find complementary information from non-adjacent spectral bands) [119], and many other research papers exist on the topic of RNNs for HSI classification [120-122].

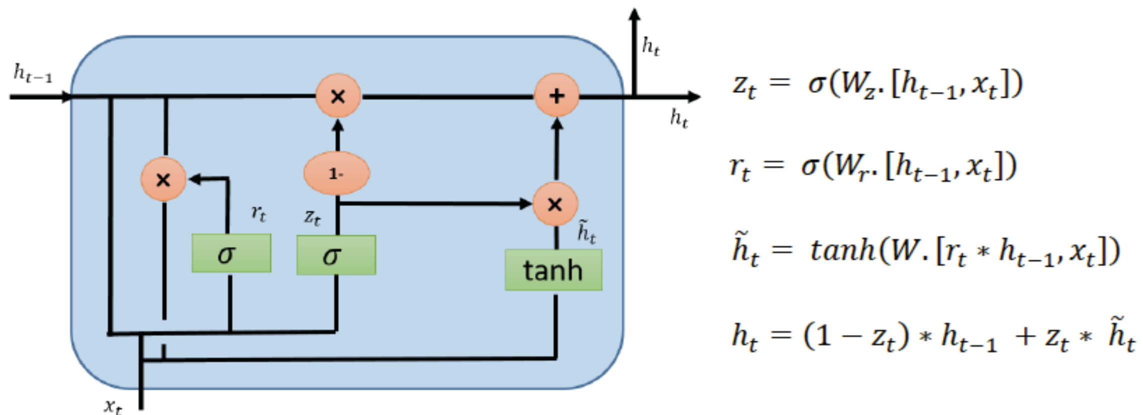


Figure 10: Structure of a GRU cell and mathematical expression, as depicted by Alom et al. [42]. The state of the GRU at the time t is represented by h_t, which is calculated using the previous state h_{t-1} and an auxiliary input x_t. The other variables show the intermediate calculations, where σ stands for the activation function used [42].

Now that we are about to present traditional machine learning methods like *support vector machines* and the *k-nearest-neighbor algorithm* (k-NN) as we used these in the experiment to see if neural networks outperformed them, it is time to discuss how ANNs and their specializations relate to SVM and k-NN. What we have presented so far concerns the field of *deep learning*. Looking at the literature, deep learning is a subset of neural networks where the ANN consists of more than one hidden layer [123-125]. As told already in 2.2.1, neural networks are a technique inspired by the brain, but they are not the only one. *Spiking computing*, for instance, is a brain-inspired paradigm that takes into consideration not just a spike's amplitude, but also "the width of pulse and the timing relationship between different pulses" for calculations, viewing the communication on the dendrites and axons as "spike-like pulses", much like the spiking of the brain [126]. IBM TrueNorth is a project inspired by spiking computing [127]. *Machine learning* is a broader term "defined in 1959 by Arthur Samuel as the field of study that gives computers the ability to learn without being explicitly programmed", which entails SVM and k-NN [126]. Lastly, John McCarthy coined the phrase *artificial intelligence* as "the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do" in the 1950s [126]. Examples of AI that do not fall under machine learning include *automated theorem proving* [128, 129] and *knowledge-based engineering* [130].

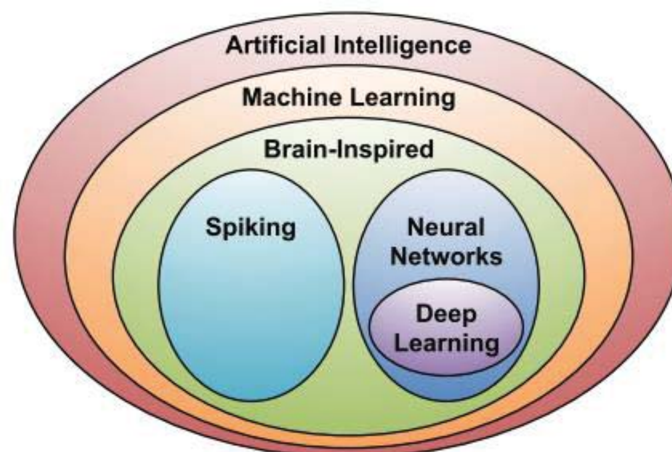


Figure 11: Disambiguation of AI-related terms by Sze et al. [126]. Any ANN (CNN, RNN) falls under deep learning, whereas SVM and k-NN are machine learning methods. Spiking computing shows that not all brain-inspired methods are neural networks, just like not all AI-related terms automatically fall under machine learning [126].

Support vector machines (SVMs) are a supervised method which dates back to the work of Vapnik und Chervonenkis in 1974 [131] that we use for both linear and non-linear classification. Linear classification means that given many data points as p-dimensional vectors, a linear classifier would strive to construct an optimal $(p-1)$-dimensional hyperplane to separate the data with, whereby phenomena like the *bias-variance tradeoff* are considered (e.g., by allowing to misclassify outliers in exchange for a model with a smaller variance, which pays off in the long term). A criterion for an optimal hyperplane can be the maximum distance to the nearest data sample, also known as *margin*. Accordingly, the linear classifier is known as the *maximum-margin classifier* if the classes are linearly perfectly separable, and in cases where misclassification has been allowed, as a *soft-margin classifier*, also known as *support vector classifier*. The name "support vector" stems from the fact that data samples on the edge and within the soft margin are called support vectors [93]. This is because "the maximal margin hyperplane depends directly on the support vectors", or, in other words, it is supported by them [93].

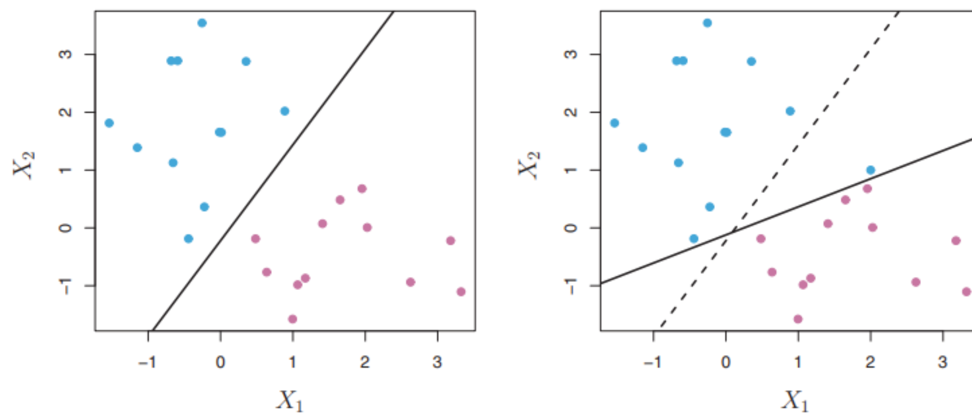


Figure 12: Separation variants of data points with X_1 and X_2 components illustrated by James et al. [93]. Maximum-margin classifiers are sensitive to outliers like the blue dot added in the right picture. If we allow the outlier to be misclassified so that we arrive at a support vector classifier, the $(2-1)$-dimensional hyperplane (a line) would have a desirable larger margin, which shows the practical implications of the bias-variance tradeoff [93].

SVM classification can also be done with a non-linear kernel. *Kernels* or kernel functions systematically find support vector classifiers in higher dimensions, which is handy when the data is not clearly separable in the current low dimensionality. In doing so, they compare every pair of measurements to figure out if a transformation would be suitable for the data, without actually transforming it (to save the SVM unnecessary computations), which is an analysis known as the *kernel trick* [93]. Common kernels include the polynomial kernel and the Gaussian radial basis function, both of which we use in the experiments, alongside the linear kernel as outlined in Figure 20 (the kernel with the best OA prevails) [132]. We will include SVM_grid as an SVM variation which employs *grid search* to exhaustively search for model candidates from a grid of parameters we specify. We will also try the k-nearest neighbor algorithm for classification, which works by identifying the closest k data points to a given point and "estimates the conditional probability" of the point for a class as a fraction of the neighboring points belonging to the class [93]. In other words, the most common class of the neighbors is assigned to the point, which can be done using a plurality vote. These machine learning methods will put into perspective the results we get for the different neural network architectures, which are our main focus.

2.3 Related Work

We will now outline important related work by stating the **problem** tackled, the **action** as a response to the problem and the **result** respectively. These PAR descriptions of a paper are formulated based on the content of the respective paper cited after the name of the paper above – we did this to avoid citing every paper at least three times if not before every semicolon, which would have unnecessarily bloated the descriptions to the point of impacting the readability. This applies to all related work sections of the thesis, i.e. this section, 3.3 and 9.1.

2.3.1 Hyperspectral Classification

Just like deep learning methods are popular for image classification tasks, CNNs have been frequently used for HSI classification tasks [13]. All hyperspectral CNNs we use for the experiments are associated with papers that could be considered related work and we did outline their varying ideas in Figure 20 and Figure 21, but the scenario is always the same (HSI-CNNs for remote sensing classification) and the fact that 3D CNNs can jointly use spectral and spatial information, as emphasized in many descriptions of these CNNs, is something we know and have presented already (see Figure 6). Therefore, we will include interesting use cases and current developments for the combination of deep learning and HSI instead. A survey of this topic has been conducted by Signoroni et al., who shine a light on HSI's challenges, e.g., data handling and interpretability of hyperspectral data, high-cost capturing equipment and few publicly available datasets, and opportunities, e.g., the improved results by jointly using "both spectral and spatial features", regardless of whether the domain is remote sensing, biomedical applications, food and agriculture or none of that [13]. They also included keywords to categorize current CNN developments, which we will happily use [13].

Training Sample Restrictions

R-VCANet: A New Deep-Learning-Based Hyperspectral Image Classification Method [41]

- *Problem:* HSI deep learning methods require many training samples, but for HSI data, the number of samples is usually limited
- *Action:* propose a **rolling guidance filter** (RGF) and **vertex component analysis** network R-VCANet, which uses inherent spatiospectral HSI characteristics to construct the network: first combine via RGF to explore contextual structure features and remove small details; then the vertex component analysis network extracts deep features from smoothed HSI
- *Result:* R-VCANet performs better than state-of-the-art (February 2017) methods

MugNet: Deep Learning for Hyperspectral Image Classification Using Limited Samples [133]

- *Problem:* deep learning methods may perform poorly for only few training samples available
- *Action:* propose a **multi-grained network** (MugNet) as a small-scale data based simplified deep learning model with only few hyperparameters that need to be tuned; first, multi-grained scanning explores spectral relationships between bands and spatial correlations for neighboring pixels, and can compare different grains' spatiospectral relationship; second, generate convolution kernels in semi-supervised way to take advantage of unlabeled hyperspectral pixels
- *Result:* evaluation experiments on grss_dfc_2013 and grss_dfc_2014 datasets show that MugNet outperforms state-of-the-art (October 2017) HSI classification methods

A Deep Network Architecture for Super-Resolution-Aided Hyperspectral Image Classification with Classwise Loss [134]

- *Problem:* few labeled samples available for HSI; want to enhance discriminative ability of a network
- *Action:* propose deep network for super-resolution (SR)-aided HSI classification with class-wise loss (SRCL); it consists of a three-layer SR CNN to **reconstruct a high-resolution image** from a low-resolution image, an unsupervised tripled-pipeline CNN (TCNN) with an improved class-wise loss and a classification module
- *Result:* SRCL outperforms state-of-the-art (August 2018) classification methods

Capsule Networks

Hyperspectral Image Classification Based on Capsule Network [135]

- *Problem:* CNNs suffer from the problem of limited training samples
- *Action:* use **affine transformation matrix** in a capsule network: first perform HSI classification based on spectral information, then integrate spatial and spectral information for better accuracy
- *Result:* effectiveness of the framework is demonstrated by appropriate experiments

Classification-Related Tradeoffs

Low–High-Power Consumption Architectures for Deep-Learning Models Applied to Hyperspectral Image Classification [136]

- *Problem:* high computational complexity and energy usage of HSI-CNNs
- *Action:* explore the use of **low-power consumption architectures** and deep learning algorithms for HSI classification
- *Result:* NVIDIA Jetson Tegra TX2 performs well in terms of performance and low-energy consumption

Feature Extraction with Multiscale Covariance Maps for Hyperspectral Image Classification [137]

- *Problem:* overfitting problems for HSI-CNNs
- *Action:* use multiscale **covariance maps** for handcrafted feature extraction so that spectral and spatial information can be jointly exploited: each entry symbolizes the covariance between two spectral bands in a local spatial window; can enhance samples with spatial information using a multiscale strategy
- *Result:* 2D CNN on HSI datasets demonstrate increased robustness of the model; classification performance together with multiscale covariance map is better than the state of the art (February 2019)

2.3.2 Visualization

As outlined in 2.2.3, various parts of a CNN can be visualized in different ways. Olah et al.'s survey on feature visualization includes a table with many related papers, which are categorized based on the amount of regularization they include [138]. A recent visualization trend of neural networks concerns *activation atlases*, i.e. "using feature inversion to visualize millions of activations from an image classification network", as outlined by Carter et al. [139], although this state of the art is not related to our work. Nevertheless, there are numerous papers which deal with saliency, guided backpropagation and/or activation maps. Apart from the visualization survey [108] and Alsallakh et al.'s paper [109], the following are papers we deem to be relevant. The findings of the papers and frameworks developed for RGB datasets are definitely advanced enough for interpretability investigations in our opinion, but HSI visualization techniques are a topic we still have to consider very novel based on the related work we have found, or rather, could not find.

Visualizing Higher-Layer Features of a Deep Network [105]

- *Problem:* need qualitative analyses to find interpretations of high-level features of deep architectures
- *Action:* contrast and compare techniques applied on Stacked Denoising Autoencoders and Deep Belief Networks: **activation maximization** and **linear combination** of previous layers' filters
- *Result:* interpretation is possible, easy and consistent

Saliency Based Visualization of Hyperspectral Satellite Images Using Hierarchical Fusion [140]

- *Problem:* hyperspectral image needs to be reduced to RGB to be visualized: need to retain the maximum information possible in doing this conversion in accordance with salient regions, which are sensed by the human visual system
- *Action:* **weighted fusion** method of saliency maps and hyperspectral bands
- *Result:* efficacy demonstrated on datasets captured by AVIRIS and ROSIS sensors

Explaining Hyperspectral Imaging Based Plant Disease Identification: 3D CNN and Saliency Maps [141]

- *Problem:* want to have an accurate and explainable model for plant disease classification using hyperspectral data
- *Action:* develop **3D CNN** for soybean charcoal rot disease identification; use **saliency map** for inference and visualize most sensitive pixel locations for classification
- *Result:* can be more confident about the model, given that most sensitive wavelength is near-infrared and that this commonly coincides with research results

Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps [100]

- *Problem:* visualize image classification models learned with CNNs
- *Action:* propose two visualization techniques based on **gradient computation** of class score with respect to input image: class model visualization vs. image-specific class saliency visualization
- *Result:* these maps can be used for weakly supervised object segmentation

Visualizing and Understanding Convolutional Networks [103]

- *Problem:* no understanding on good classification performance of large CNNs or how to improve these models
- *Action:* propose new visualization technique (**deconvolutional network**) for examining intermediate feature layers and the role of the classifier, which also allows to find better performing model architectures; discover contribution of model layers to performance
- *Result:* newly found ImageNet model generalizes and performs very well on Caltech-101 and Caltech-256 datasets

Striving for Simplicity: The All Convolutional Net [46]

- *Problem:* not all components of modern CNNs, i.e. alternating convolutional and max pooling layers, followed by few fully connected layers, might be necessary
- *Action:* propose fully convolutional network with competitive accuracy on CIFAR-10, CIFAR-100 and ImageNet datasets; analyze it using the deconvolution approach for feature visualization
- *Result:* max pooling can be **replaced by convolutional layer with increased stride**

Salient Object Detection in Hyperspectral Imagery [142]

- *Problem:* visualization efforts until now have been often pixel-based, aiming to extract spatial-spectral features
- *Action:* combine hyperspectral data with salient object detection: introduce **visual saliency model into spectral domain** to extract salient spectral features (related to material property and spatial object layout); implement and compare three methods to show how the color component in the traditional saliency model can be replaced by spectral information
- *Result:* experiments on three hyperspectral datasets show effectiveness of proposed methods

Target Detection of Hyperspectral Image Based on Spectral Saliency [143]

- *Problem:* traditional unsupervised target detection algorithms are difficult due to lack of prior information about the target, they exploit spectral information
- *Action:* propose **spectral saliency target detection**, a saliency detection technique in the HSI processing domain: utilize both spatial and spectral saliency by combining model with spectral matching to perform well for concealed and small targets
- *Result:* superior and stable performance of proposed algorithm on eight HSI scenes with complex background

Learning Important Features Through Propagating Activation Differences [101]

- *Problem:* want to explore interpretability of neural networks
- *Action:* DeepLIFT decomposes output prediction by **backpropagating neuron contributions** to every input feature; these contribution scores are calculated by comparing neuron activations with “reference activations”; contributions can optionally be set to consider also negative scores
- *Result:* significant advantages of DeepLIFT on MNIST over gradient-based methods

3. Dimensionality Reduction

Now that we have introduced the concepts of HSI and CNNs in sufficient detail, we can talk about strategies to circumvent the curse of dimensionality. We can remove image bands, decide to compress the neural network, or combine both approaches. The necessary theory is the topic of this chapter.

3.1 Image Compression

When dealing with HSI, the challenge we are confronted with is the large number of often hundreds of image bands [11], which slow down computation [7]. Various research, motivated by findings such as the one of Jimenez and Landgrebe that actual data of a high-dimensional space is often located in a low-dimensional subspace (which means the rest is empty, i.e. a waste of space) [144], has shown good results with fewer image bands (see references in 3.1.1, 3.1.2 and 3.3), which are either obtained through transformation and/or combination (*feature extraction*) or elimination (*feature selection*) of image bands. Regarding which category would be preferable over the other, Su et al. mark that there is "still a **lack of research comparing different feature selection and feature extraction algorithms** using a limited number of training samples **in the context of HSI classification**" [98]. But that does not mean we should refrain from applying image compression altogether: Sarhrouni et al. even claim that image bands "can be **not only redundant, but a source of confusion**, and decreasing so the accuracy of classification" [145]. We need to understand the ideas and basic theoretical foundations of the most prominent techniques we will use in the experiment, so the following two subchapters are dedicated to explaining selected techniques. Mainly, we will focus on **PCA, NMF, LLE** and **Autoencoder** in 3.1.1 and present **linear regression**, **logistic regression** and **random forest** in 3.1.2, where similar techniques in terms of idea or mathematical calculation are grouped together, and vary the length of each explanation according to the importance of the band selection technique in the experiment.

As a case in point, let us consider hyperspectral imagery obtained by a satellite. Not only is the bandwidth to Earth typically highly constrained [146], but the volume of hyperspectral data acquired by the sensors also poses a formidable challenge to the satellite [36]. Therefore, the satellite performs an **on-device preprocessing** of the data using an image compression method. Depending on the architecture and performance / capabilities of the satellite used, we can then either decide in favor of on-satellite inference with (preferably, compressed) neural networks so that we only transmit the result to Earth to alleviate the bandwidth concerns the most, at the cost of the computational effort involved in on-satellite preprocessing (e.g., that minerals X have been detected by scanning position Y), or we decide to transmit the images with fewer image bands to Earth so that a more powerful computer can take care of the evaluation. This decision highlights the tradeoff between the satellite's computational capability and the bandwidth available to cast the data to Earth. As a theoretical expansion, one could decide to realize fault-tolerance so that from the prioritized batch of data received on Earth, the first elements suffice for further analysis because they are way more important than following elements (for which there are quantifying metrics, e.g., the explained variance of the component). In any case, dimensionality reduction has helped to reduce the amount of data that needed to be sent to Earth.

Speaking of image compression, the ISO 18381:2013 standard called "Space Data and Information Transfer Systems - Lossless Multispectral and Hyperspectral Image Compression" reviewed and confirmed in 2018 [147] "establishes a data compression algorithm applied to digital three-dimensional image data from payload instruments, such as multispectral and hyperspectral imagers, and specifies the compressed data format" with regard to the data compression goals of reducing the "transmission channel bandwidth", "the buffering and storage requirement" and the "data-transmission time at a given rate" [147]. The corresponding informational report CCSDS 120.2-G-1 from December 2015 explains that one can split an image into smaller images to "limit the impact of data loss" through compression and "reduce implementation memory requirements for some compression approaches", thus distinguishing between full image and segmented image compression [148]. It specifies that the "compressor is intended to be suitable for use [on an] onboard spacecraft; in particular, the algorithm complexity and memory use are

designed to be sufficiently low to make highspeed hardware implementation feasible” [148]. Their results regarding hyperspectral images and compression effectiveness (i.e. data rates) are that “full image compression always performs as well as or better than segmented compression”, as the table below illustrates [148]. Arguably, the precise techniques used are at most tangentially related to the techniques we will focus on as we have just mentioned, but the fact that an official standard (with a plethora of references on lossless on-satellite image compression for further research) points out that for lossless image compression, one could compress hyperspectral images by a big factor according to Figure 13 motivates both our theoretical exploration of band selection techniques in the following two subsections and related experiments in chapter 4.2.2.

	Instrument	CCSDS-123.0-B		JPEG-LS direct		ESA + sample-adaptive coding		LUT + sample-adaptive coding		CCSDS-122.0-B		JPEG2000	
		Segmented	Full Image	Segmented	Full Image	Segmented	Full Image	Segmented	Full Image	Segmented	Full Image	Segmented	Full Image
Hyperspectral Images	IASI	4.77	4.75	6.61	6.60	4.90	4.91	5.72	5.57	7.62	7.55	7.42	7.02
	AIRS	4.32	4.30	6.35	6.35	4.66	4.68	6.13	5.66	6.91	6.86	7.01	6.62
	CRISM-FRT	5.21	5.06	5.74	5.53	8.92	8.92	9.60	9.54	6.93	6.92	6.30	5.97
	CRISM-HRL	4.73	4.56	5.75	5.55	8.38	8.38	9.15	9.06	6.82	6.80	6.32	5.95
	CRISM-MSP	2.90	2.55	3.75	3.42	6.80	6.80	7.32	7.02	5.01	4.91	4.70	3.85
	M3-Global	2.37	2.14	4.93	4.83	6.45	6.45	7.30	7.20	5.74	5.72	5.40	5.10
	M3-Target	3.25	3.09	3.82	3.66	6.60	6.60	7.51	7.44	5.03	5.02	4.29	4.00
	Hyperion	4.37	4.30	5.09	5.02	5.52	5.52	6.16	6.08	5.60	5.58	5.39	5.14
	Hyperion flatfield	3.98	3.97	4.81	4.80	4.11	4.11	4.55	4.54	5.00	4.99	5.08	4.89
	SFSI	4.69	4.67	4.77	4.75	4.91	4.91	5.43	5.43	5.30	5.30	4.79	4.65
	SFSI_rmnoise	3.01	2.96	4.40	4.35	4.07	4.07	4.80	4.70	4.89	4.88	4.56	4.42
	AVIRIS(16-bit raw)	5.99	5.98	8.64	8.61	6.16	6.16	7.42	6.87	8.79	8.78	8.98	8.88
	AVIRIS(12-bit raw)	2.69	2.68	4.56	4.54	3.01	3.01	3.53	3.44	4.73	4.72	4.67	4.59
	AVIRIS(16-bit cal)	3.76	3.74	6.41	6.39	4.32	4.32	4.75	4.54	6.58	6.57	6.65	6.56
	CASI	5.03	5.02	6.79	6.77	5.10	5.10	5.96	5.65	7.04	7.02	7.09	6.97
Multispectral Images	MODIS-night	4.64	4.70	5.39	5.38	7.76	7.76	7.24	7.22	6.20	6.20	5.54	5.51
	MODIS-day	5.63	5.72	4.69	4.69	5.75	5.75	6.54	6.44	5.59	5.59	5.43	5.37
	MODIS-500m	7.19	7.20	7.63	7.62	8.36	8.36	8.88	8.88	8.21	8.22	7.80	7.77
	MODIS-250m	6.43	6.48	6.96	6.96	7.15	7.15	7.73	7.73	7.14	7.14	7.11	7.08
	MSG	3.40	3.39	3.50	3.50	4.11	4.11	5.07	5.06	3.70	3.70	3.53	3.50
	LANDSAT	3.37	3.37	3.70	3.70	3.91	3.91	4.32	4.32	4.03	4.03	3.90	3.87
	PLEIADES	7.32	7.32	7.85	7.84	7.95	7.95	8.74	8.65	8.05	8.05	8.25	8.15
	VEGETATION_level1b	5.26	5.26	5.31	5.30	5.86	5.86	6.97	6.96	5.59	5.59	5.53	5.48
	VEGETATION_level1c	5.04	5.04	5.27	5.26	5.55	5.55	6.77	6.77	5.46	5.46	5.46	5.42
	SPOT5	4.53	4.53	4.71	4.71	5.17	5.17	5.66	5.66	4.96	4.96	4.94	4.92

Figure 13: Side-by-side comparison of average compressed data rates for segmented and full images in bits per sample for several instruments in both hyperspectral and multispectral scenarios using the image compression standards and techniques in the columns, where “[b]etter performance [i.e. fewer bits per sample necessary] is indicated in green” [148]. The authors explain their results in the following way: “For hyperspectral images, full image compression always performs as well as or better than segmented compression. For multispectral images, where segment size is selected to be larger, this performance penalty is less apparent, and in some cases segmentation even provides a small benefit.” [148]

3.1.1 Feature Extraction

Principal component analysis (PCA) is a technique from the field of multivariate statistics that uses orthogonal transformation “to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components” [149]. It can be used for dimensionality reduction as the number of axes constituted by the principal components is less than or equal to the number of original dimensions. These principal components are chosen to **explain as much variance of the data as possible** in decreasing order, while allowing for a reconstruction with minimal squared error [150]. Consequently, the hope for our case is that when we instruct PCA to keep, e.g., 70 components out of 200, where the components equate to the number of hyperspectral image bands used, these will suffice for characteristic features of the hyperspectral dataset to be detected so that a high OA during classification will be reached. Additional motivation is given by the findings of Su et al., who have plotted the correlations quantified by the *Pearson Correlation Coefficient* (PCC), which describes the standardized covariance [151], between image bands for the PaviaU dataset, finding that 56% of the 5253 PCCs had values larger than 0.7, indicating high correlation and with that, large image compression potential (“0 means no linear correlation, 1 (or -1) means totally linear positive (or negative) correlation”)

[98], along with a lot of similar work (see 3.3.1). PCA is by far one of the most popular dimensionality reduction algorithms [152-155].

Before we get into the mathematical reasoning, we need to establish our goal and introduce the variables. We want to find a subspace $\mathbb{R}^k$ of $\mathbb{R}^n, k < n$ to project our (possibly strongly correlated) data from our dataset $\{x^{(i)} \in \mathbb{R}^n : i = 1, \dots, m\}$ into while retaining as much variance as possible with fewer dimensions. Without loss of generality, we assume the mean of the data is zero and that all coordinates have been preprocessed to have a unit variance (for scaling purposes). Now we want to find a unit vector u, i.e. the first axis of the new coordinate system we seek to construct, so that the length of the projection of x onto u is maximized, which we can calculate by $x^T u$. Thus, we arrive at the following problem, where we first just simplify the term:

$$\max_{u:\,\|u\|=1} \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)T}u\right)^2 = \max_{u:\,\|u\|=1} \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)T}u\right)^T\left(x^{(i)T}u\right) = \max_{u:\,\|u\|=1} \frac{1}{m}\sum_{i=1}^{m}\left(u^T x^{(i)}\right)\left(x^{(i)T}u\right)$$

$$= \max_{u:\,\|u\|=1} \frac{1}{m}\sum_{i=1}^{m} u^T x^{(i)} x^{(i)T} u = \max_{u:\,\|u\|=1} u^T\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)} x^{(i)T}\right)u$$

With $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} x^{(i)T}$, the above formula yields $\max_{u:\,\|u\|=1} u^T \Sigma u$. This is a constrained optimization problem because of $u^T u = \|u\| = 1$, and we can use the Lagrange multiplier to continue:

$$\mathcal{L}(u,\lambda) = u^T \Sigma u + \lambda(u^T u - 1) \Rightarrow \frac{\partial \mathcal{L}(u,\lambda)}{\partial u} = \frac{\partial(u^T \Sigma u + \lambda u^T u)}{\partial u} = \frac{\partial(u^T \Sigma u)}{\partial u} + \frac{\partial(\lambda u^T u)}{\partial u}$$

To further solve this derivative, let us consider that $u \in \mathbb{R}^n$ is a column vector, so

$$\frac{\partial(u^T \Sigma u)}{\partial u} + \frac{\partial(\lambda u^T u)}{\partial u} = 2\Sigma u + 2\lambda u = 0 \overset{\lambda := -\lambda}{\Longleftrightarrow} \Sigma u = \lambda u$$

We see that λ is the eigenvalue to the eigenvector u. This is a process we can repeat with more unit vectors perpendicular to one another until we arrive at $u_1, \dots, u_k$ as the top k vectors of Σ, forming an orthonormal basis. More to the point, we want to sort our pairs of eigenvalues λ_i with corresponding eigenvectors u_i descendingly by λ_i ($\lambda_1 \geq \dots \geq \lambda_n$) because the variance we retain is described by $\sum_{i=1}^{k} \lambda_i$ and since $n - k$ dimensions will intentionally be lost, it makes sense to choose the largest eigenvalues. Accordingly, the retained variance of $\mathbb{R}^n$ in our subspace $\mathbb{R}^k$ is $\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{n} \lambda_i}$.

To summarize, with the assistance of Li and Wang's and Ng's notes on PCA, we have just shown that the u_is of $u_1, \dots, u_k$, which are called *principal components*, form an orthogonal basis for the data (hyperspectral image bands) we would like to project from $\mathbb{R}^n$ to $\mathbb{R}^k, k < n$. The new representation for a $x^{(i)} \in \mathbb{R}^n$ with regard to the basis is [149, 156]

$$\hat{x}^{(i)} = \begin{pmatrix} u_1^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{pmatrix} \in \mathbb{R}^k$$

Several generalizations of PCA exist (Kernel PCA, Sparse PCA, Incremental PCA), as well as methods relying on very similar calculations (MDS, SVD, ICA). *Kernel PCA* is a PCA variation that uses a kernel function $\phi : \mathbb{R}^n \to \mathbb{R}^q, q > n$, choosing ϕ well enough to be able to apply standard PCA in the higher-dimensional $\mathbb{R}^q$. This is done because linear separation is not always possible in $\mathbb{R}^n$, but chances increase with the dimensionality. Following the same procedure as above, we obtain $Ku = \lambda u$ for the kernel matrix $K = \phi^T \phi$. This follows from the symmetry and positive semi-definite properties of a kernel [157, 158]. *Sparse PCA* addresses the concern that the principal components are a linear combination of many, if not all input variables (bad interpretability) by sparsity constraints so that many components of the eigenvectors become zero [157]. *Incremental PCA* is used to split large datasets that do not fit in memory into batches

which can be processed using PCA [159]. As for remote sensing, Mather and Koch state that PCA applications "normally result in a first principal component that is a weighted average of all bands in the dataset" [160]. As we will see in the related work section (3.3.1), the first few principal components often already explain the majority of variance [161-163].

Multidimensional scaling (MDS) creates plots based on distances between samples, whereas PCA works with correlations among samples [157]. MDS works numerically just like PCA does when using the Euclidean distance as a distance metric [164], but obviously, there are also other distance metric candidates, e.g., Manhattan / Cosine / Mahalanobis / Hamming distance. It often directly takes a distance matrix as an input [165] or, alternatively, works using a dissimilarity matrix D by trying to find an optimal configuration $X \subseteq \mathbb{R}^k : f(D_{ij}) \approx d_{ij} = (x_i - x_j)^2$, where f is a monotone function and k the number of dimensions in the subspace [166]. This classical / *metric* MDS we have been talking about needs to be distinguished from *non-metric MDS* (nMDS): "Metric MDS transforms the dissimilarity data into a set of Euclidean distances so that the final configuration (i.e. an arrangement of points in a smaller number of dimensions) matches the original dissimilarities as closely as possible. nMDS focuses on the rank order of the dissimilarity matrix and requires the final configuration to contain distances that follow the rank of the original dissimilarity matrix as closely as possible." [164]

PCA's eigenvalue decomposition only works for diagonalizable matrices. In contrast to diagonalization, *singular value decomposition* (SVD) is a technique that factorizes any $X \in \mathbb{R}^{m \times n}$ so that it can be written as $X = WDU$ with $W \in \mathbb{R}^{m \times m}, D \in \mathbb{R}^{m \times n}, U \in \mathbb{R}^{n \times n}$ with D being a diagonal matrix [167]. PCA uses SVD to take the columns of W as the eigenvectors of the covariance matrix, which are principal components that can be ranked in importance according to the descending order of their respective eigenvalue [156, 157].

In contrast to PCA which tries to best explain the variability of the data, *independent component analysis* (ICA) seeks to construct a basis where each basis vector is an independent component [168]. As Comon says, "[t]he concept of ICA may [...] be seen as an extension of [PCA], which can only impose independence up to the second order and, consequently, defines directions that are orthogonal" [169].

To present another technique, *linear discriminant analysis* (LDA) is a supervised learning technique (unlike the unsupervised PCA) in the goal of finding "$N - 1$ basis vectors that maximize the interclass distances while minimizing the intraclass distances" for classifying N classes [170].

The next important method after PCA is *non-negative matrix factorization* (NMF), but even NMF is not substantially different from PCA. As Lee and Seung state, NMF is the problem of decomposing a non-negative matrix V into non-negative matrix factors $W \in \mathbb{R}^{n \times r}$ and $H \in \mathbb{R}^{r \times m}$ so that $V \approx WH$ [171], but in another paper they point out that both PCA and NMF find approximate factorizations for the above formula, but with different constraints: "PCA constrains the columns of W to be orthonormal and the rows of H to be orthogonal to each other", whereas "NMF does not allow negative entries in the matrix factors W and H" [172]. To get the desired matrix factorization, they outline the iteratively applied rules

$$W_{ia} \leftarrow W_{ia} \sum_{\mu} \frac{V_{i\mu}}{(WH)_{i\mu}} H_{a\mu} \qquad H_{a\mu} \leftarrow H_{a\mu} \sum_{i} W_{ia} \frac{V_{i\mu}}{(WH)_{i\mu}} \qquad W_{ia} \leftarrow \frac{W_{ia}}{\sum_j W_{ja}}$$

These rules converge "to a local maximum of the objective function" F which satisfies the nonnegativity constraints we want, and thus, we arrive at the desired decomposition of V into W and H [172].

$$F = \sum_{i=1}^{n} \sum_{\mu=1}^{m} \left[V_{i\mu} \log(WH)_{i\mu} - (WH)_{i\mu} \right]$$

A category of non-linear dimensionality reduction is called *manifold learning*. Associated techniques like *locally linear embedding* (LLE), *uniform manifold approximation and projection* (UMAP), *t-distributed stochastic neighbor embedding* (t-SNE), *isometric mapping* (ISOMAP) and techniques built on random

projection (*sparse random projection*, *Gaussian random projection*) use projection to reduce the dimensionality of datasets, but this bears the risk of losing interesting structure of the data. PCA, ICA and LDA solve this problem by choosing an appropriate linear projection that keeps interesting linear features according to their respective criterion, but this might disregard the data's non-linear structural features. Therefore, manifold learning (typically unsupervised) can be thought of as a generalization of these linear frameworks so that they also consider the non-linear structure of the data [173].

For example, the idea of LLE is to use a neighborhood-preserving mapping for a compact representation of high-dimensional data and inferring the global structure using "local symmetries of linear reconstructions" [174]. If we consider N real-valued vectors $\vec{X}_i \in \mathbb{R}^D, i = 1, \ldots, N$, the algorithm works as follows (as described by Roweis and Saul [174]):

1. Assign neighbors to each $\vec{X}_i$ (e.g., with k-NN)
2. Solve the constrained least-squares problem of minimizing the cost function for reconstruction errors

$$\varepsilon(W) = \sum_i \left| \vec{X}_i - \sum_j W_{ij} \vec{X}_j \right|^2$$

computing W_{ij} as the contribution of the j-th data point to the i-th reconstruction with the constraints

- $W_{ij} = 0$ if $\vec{X}_j$ is not a neighbor of $\vec{X}_i$
- $\sum_j W_{ij} = 1$

3. Compute low-dimensional embedding vectors $\vec{Y}_i$ best reconstructed by W_{ij} by minimizing the embedding cost function

$$\phi(Y) = \sum_i \left| \vec{Y}_i - \sum_j W_{ij} \vec{Y}_j \right|^2$$

"by finding the smallest eigenmodes of the sparse symmetric matrix" [174]

$$M_{ij} = \delta_{ij} - W_{ij} - W_{ji} + \sum_k W_{ki} W_{kj}$$

The three steps of the process are visualized in Figure 14 to get a more intuitive understanding of LLE.

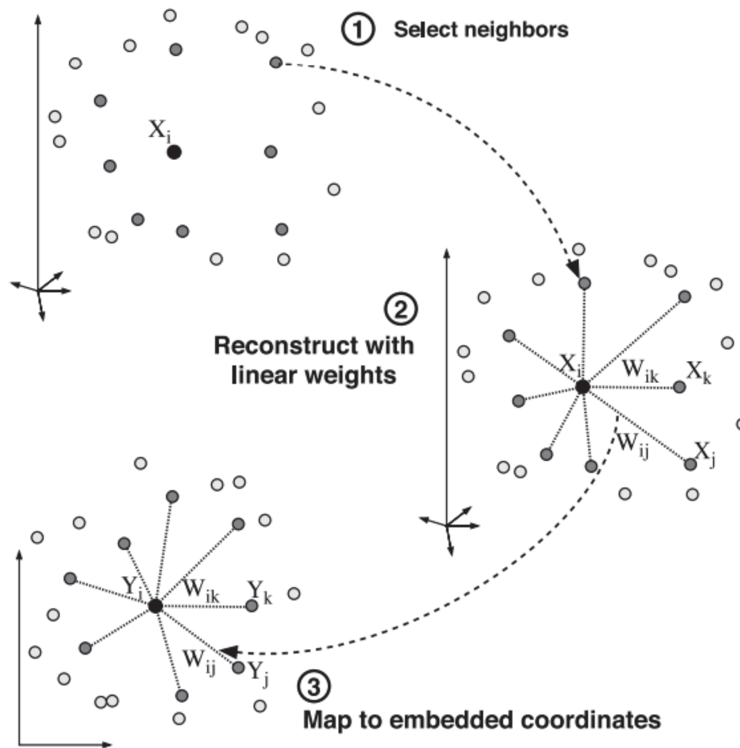


Figure 14. Visualization of LLE calculation steps by Roweis and Saul [174]. LLE does not care about calculating pairwise distances, but instead assumes that each data point X_i and its neighbors are located on a locally linear patch. This is because X_i is linearly reconstructed using its neighbors X_k and X_j and corresponding weights W_{ik} and W_{ij}. With the low-dimensional embedding vectors Y_k and Y_j, X_i can be mapped to Y_i [174].

Martinetz and Schulten as well as Tenenbaum point out that "overlapping local neighborhoods [...] can provide information about global geometry" [174], which is generally followed as a principle in manifold learning [175, 176]. However, while LLE analyzes "local symmetries, linear coefficients and reconstruction errors", ISOMAP's embeddings "are optimized to preserve geodesic distances between general pairs of data points" [174]. This is the one idea that distinguishes ISOMAP from LLE [177]. Other than that, t-SNE is a method that **retains both local and global structure of the data**, making it a popular choice for dimensionality reduction of high-dimensional data [168]. t-SNE "minimizes the sum of *Kullback-Leibler divergences* over all datapoints" both in the low-dimensional embedding and in the high-dimensional data after "converting the high-dimensional Euclidean distances between datapoints into conditional probabilities that represent similarities" [178]. The meaning of the Kullback-Leibler divergence metric here is to measure how one probability distribution is different from another. For the quantification of similarity, the biggest difference between UMAP and t-SNE is that UMAP does not minimize the Kullback-Leibler divergence, but maximizes a likelihood function [179]. Apart from that, there are random projection techniques like Gaussian random projection and sparse random projection. These project the original data into a lower-dimensional subspace using a random matrix with columns of unit length, where the difference lies in the distribution where components of the matrix are taken from [180].

Lastly, the method of *autoencoders* can be used for dimensionality reduction. Autoencoders are neural networks that consist of an encoding and a decoding part, with the goal of **learning a low-dimensional representation** of the input data in the hidden layer between these parts. If we represent the encoder and the decoder part by the functions ϕ and ψ respectively, we can express the transitions in the following way [181]:

$$\phi: \mathcal{X} \rightarrow \mathcal{F} \qquad \psi: \mathcal{F} \rightarrow \mathcal{X} \qquad \phi, \psi = \underset{\phi,\psi}{\arg\min} \|X - (\psi \circ \phi)X\|^2$$

This means that the neural network is supposed to learn the identity function, but forced to use a lower-dimensional hidden layer for the learning process from which we can extract the lower-dimensional

representation of the input (because we know for a fact that the input must have passed through the hidden layers, so whichever way the autoencoder performed the dimensionality reduction, what is important is that it did happen and we can take advantage of it). For just a single hidden layer or only linear activations, an autoencoder is strongly related to PCA [182, 183]. However, an autoencoder's strength lies in its potential to use non-linearity as this allows for a significantly lower information loss [184]. For $x \in \mathcal{X}$ mapped to $h \in \mathcal{F}$ with the weight vector W and biases b of appropriately chosen dimensions and the activation function σ, this yields $h = \sigma(Wx + b)$ for one hidden layer. In case of another hidden layer, the equation can be amended to $h = \sigma'(W'\sigma(Wx + b) + b')$ with the new variables W', b', σ' as we have seen in the forward propagation equations (cf. 2.2.2).

Different kinds of autoencoders exist, e.g., regularized autoencoders and variational autoencoders, from which the latter category is too structurally different from our topic of neural network compression and therefore outside the scope of this thesis (there are in-depth tutorials available [185], along with variants like adversarial autoencoders [186] and importance-weighted autoencoders [187]). Regularized autoencoders, on the other hand, also have many subcategories. First, there are the sparse autoencoders, which became popular following the insight that "when representations are learnt in a way that encourages sparsity, improved performance is obtained on classification tasks" [188] and include rather more than fewer hidden neurons than input neurons, but only a small number of them may be active at once [189] (realized by, e.g., exploiting the Kullback-Leibler divergence [188, 190]). Secondly, there are denoising autoencoders, where a noise is applied on the input image, corrupting the input to be reconstructed, which is supposed to lead to resilience [191]. Thirdly and lastly, Rafai et al. proposed contracting autoencoders, whose penalty term leads to a "localized space contraction", resulting in robust features in the activation layer [192].

The methods factor analysis, feature agglomeration and mini-batch dictionary learning have been included in the experiment in case they performed well, but they did not, so they have turned out to be too irrelevant to explain for our case. The way they work can be looked up in the documentation of the Python machine learning framework *scikit-learn*. Feng et al. summarized lots of other feature extraction methods [193].

3.1.2 Feature Selection

In contrast to feature extraction, which transforms image bands while reducing their total number, *feature selection* methods leave the image bands untouched, but select only the most important ones according to their respective criterion. In this section, we will present *linear regression*, *logistic regression* and *random forest* as three commonly used feature selection techniques [194]. We have selected them to both cover a range of ideas and use well-known techniques. In fact, linear regression is obviously a linear model, logistic regression is known as a generalized linear model (a term which we will come back to) and random forest is non-linear, whereby linear regression in particular was the first technique to be studied and applied extensively [195].

As Yan contends, *regression analysis* is a statistical method for investigating data obtained from scientific experiments to see if there is a "causal relationship between [the] response variable y and regressors $x_1, \dots, x_n$" [195]. Regression also has the purpose of predicting y on a set of values $x_1, \dots, x_n$ as well as identifying the importance of these respective variables. In a regression model, the response variable is written as the sum of a function of regressors and a random error, i.e.

$$y = f(x_1, \dots, x_n) + \varepsilon$$

or, in the case of a *simple linear regression* model with the dependent variable y, the y intercept β_0, the gradient or slope of the regression line β_1, the independent variable x and the random error ε [195],

$$y = \beta_0 + \beta_1 x + \varepsilon$$

By contrast, a *multiple linear regression* model has more than one independent variable in the model ($x_1, \dots, x_n$), so it can be described with [195]

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \varepsilon$$

If we consider multiple such equations, we can replace y with $y_i, i = 1, \dots, n$ and define the variables as vectors and matrices (this time, the indexing means we have n equations with linear combinations of p x_i and β_i, and every $x_i^T = (1 \quad x_{i1} \quad \cdots \quad x_{ip}), i = 1, \dots, n$) [196]:

$$y = X\beta + \varepsilon \text{ where } y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}, \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \varepsilon = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

Note that the parabola $y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i, i = 1, \dots, n$ is also a case of linear regression because what matters is the linear dependency from the β_i, not the squared x_i occurrence.

The main estimation technique for linear regression as to which line is the best to construct usually is least-squares estimation. This means that the sum of the mean squared loss, i.e. the squared distances between the data points to the plotted line, i.e. the squared *residuals*, need(s) to be minimized [196].

Logistic regression is a statistical method that models the probability of an event. A big difference is that while linear regression is well-suited for modeling relations between continuous dependent variables, logistic regression targets categorical independent variables, where the "categorical" descriptor means that variables fall into at least one, and for our purposes, exactly one category. In addition, unlike linear regression, the logistic curve is usually estimated using the maximum likelihood method, which relies on an iterative process such as Newton's method [197, 198].

The standard logistic function $\sigma: \mathbb{R} \to (0,1), \sigma(t) = \frac{e^t}{e^t+1} = \frac{1}{1+e^{-t}}$ outputs a value between zero and one, interpreted as the probability for a dependent variable Y of being true. If we set the linear combination of m explanatory variables x_i to $t = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m = \beta_0 + \sum_{i=1}^m \beta_i x_i$, this yields

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-(\beta_0 + \sum_{i=1}^m \beta_i x_i)}}$$

which describes the *generalized logit model* [199].

To explain how logistic regression relates to being a generalized linear model, we first consider that linear regression forms the basis for this notion, because a *generalized linear model* is a broader term for classifying linear models. This is because it allows to use a *link function* on the response variable as a connection to the linear model [200], which is trivial for linear regression using the identity function as the link function. In the case of logistic regression, we can choose the link function $g: (0,1) \to \mathbb{R}$ to be the logit link function so that $g\big(p(x)\big) = \sigma^{-1}\big(p(x)\big) = \operatorname{logit} p(x) = \ln\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \sum_{i=1}^m \beta_i x_i$, which goes to show that logistic regression, while not per se a linear model, is at least a generalized linear model [199]. With this last equation, we can also see that the "logistic regression model equates the logit transform, the log-odds of the probability of a success, to the linear component" [201]. These log-odds are going to become relevant for determining the importance of logistic regression features so that we can select the most important ones later on in this section.

Random forests are based on *decision trees*. These are tree representations where depending on the truth value of the conditions, which are placed at the nodes of the tree, the decision reached by fully traversing the tree to a leaf node may turn out to be different. Random forests make use of the fact that these conditions can be arranged in different orders, resulting in different decision trees. Specifically, they make use of the general wisdom obtained by the predictions of the traversals of different decision trees: if a plurality of the decision trees predicts Y, chances are that the currently considered combination of

variables $x_1, \ldots, x_n$, which decide the truth values of the conditions of the respective decision tree, is true. To make the predictions accurate, one should choose variables that have a high chance of predicting an occurrence, and it helps if the decision trees, especially their predictions, are different from (not correlated with) one another. Breiman points out the advantages of random forest of being resistant to overfitting due to the strong law of large numbers and that their generalization error converges to a limit with the growing number of decision trees [202].

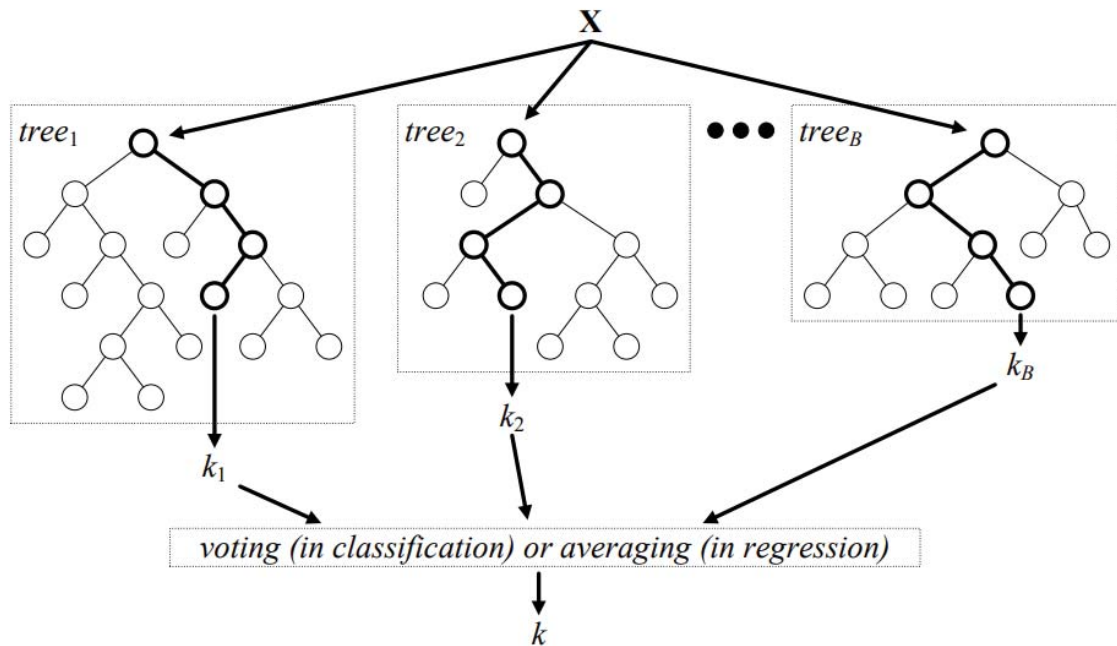


Figure 15: Simplified illustration of random forest by Verikas et al. [203]. Multiple decision trees $tree_i$ structurally possibly different from one another might arrive at different outcomes k_i for the input X, $i = 1, \ldots, B$, but in the end, they are combined to k. This can be done using averaging for regression purposes, or with plurality voting for classification tasks [203].

Random forests belong to the group of ensemble learning methods as they rely on aggregating the results of multiple classifiers, which are decision tree classifiers. For generating these decision trees, a number of *bootstrap samples* are drawn from the original data, whereafter the randomly sampled predictors of one of the bootstrap samples generate “an unpruned classification or regression tree“, so we arrive at multiple decision trees [204]. To evaluate them, one uses *out-of-bag* (OOB) data, i.e. data outside of the bootstrap sample (in neural networks, we would have called this the test set) and aggregates these OOB predictions [204]. Different methods exist to combine decision tree classifiers, with the most prominent ones being boosting and bagging. *Bagging* has been proposed by Breiman and can be summarized in the most important property by saying that each decision tree is constructed independently from one another and a plurality vote / the average determines the result for classification / regression respectively [205]. It is employed by the random forest classifier implementation in the scikit-learn library [206], which we are going to use. By contrast, Shapire et al. explained *boosting* by the fact that while evaluating the decision trees subsequently, each successive tree increases the weight of points which earlier predictors predicted wrongly [207]. Song et al. fault random forest for its difficult interpretability, especially if compared to generalized linear models (e.g., linear and logistic regression), but point out its superior accuracy as an ensemble predictor [208]. This is a confirmation of previous work such as Liaw and Wiener’s, who also praise the high accuracy of random forest in their regression experiment [204]. This combination of academic popularity and promising results justifies our choice of random forest as the third and last feature selection algorithm.

Having presented the necessary theory, how do we use linear regression, logistic regression and random forest to actually reduce the number of image bands for hyperspectral image classification? From the

practical viewpoint, the answer is that scikit-learn provides selectors for *backward threshold-based feature elimination*, *recursive feature elimination* and *forward feature selection*. Any such technique would take the instance of the linear regression, logistic regression or random forest algorithm as a parameter to estimate the importance of the, e.g., 200 image bands with and return the relevant image bands accordingly (see 4.1.1). Theoretically speaking, there are different criteria that can be applied to rank the image bands for their importance.

If we look at scikit-learn's implementation of the methods, we notice which default criteria are used. Linear regression both in our case and by default in scikit-learn uses the R^2 metric to determine the most important dimensions. There are different ways to calculate it, but we will use the explained variance as the basis. If we denote

- the fraction of variance unexplained as FVU,
- the variance of the residuals as VAR_{err},
- the sample variance of the dependent variable as VAR_{tot},
- the sum of squares of the regression, or equivalently, the explained sum of squares as SS_{reg},
- the sum of squares of residuals as SS_{res},
- the total sum of squares by $SS_{tot} = SS_{reg} + SS_{res}$,

$$R^2 = 1 - FVU = 1 - \frac{VAR_{err}}{VAR_{tot}} = 1 - \frac{SS_{res}}{SS_{tot}} \overset{(*)}{=} 1 - \left(1 - \frac{SS_{reg}}{SS_{tot}}\right) = \frac{SS_{reg}}{SS_{tot}} = \frac{SS_{tot} - SS_{res}}{SS_{tot}}$$

where $(*)$ is only true in special cases like linear regression, which is our topic for now. The higher R^2, the more variance an axis explains [209]. Consequently, it ranks higher among the hundreds of image bands in terms of importance, so it will likely be selected by feature selection as a candidate to be kept. There are metrics like the p-value we could use to make sure linear regression model has a sufficient statistical significance, but this is beyond the scope of this explanation.

For logistic regression, we need to alter the above procedure. There are pseudo-variants of R^2, with the most popular ones being McFadden's and Efron's variant [210] and lots of alternatives [211]. These are supposed to emulate, but not approximate R^2, as can be found out in adequate experiments [212]. McFadden's pseudo-R^2 uses log-likelihoods:

$$R^2 = 1 - \frac{\ln\left(LL(M_{full})\right)}{\ln\left(LL(M_{intercept})\right)}$$

where M_{full} is the logit model selected, $M_{intercept}$ the data projected onto the overall probability and LL denotes the log-likelihood [210]. As Shalizi describes, "[b]ecause logistic regression predicts probabilities, rather than just classes, we can fit it using likelihood. For each training data-point, we have a vector of features, x_i, and an observed class, y_i[, which are parameters that both belong to a model M]. The probability of that class was either p, if $y_i = 1$, or $1 - p$, if $y_i = 0$. The likelihood is then" [213]

$$L(M) = \prod_{i=1}^{n} p(x_i)^{y_i} \left(1 - p(x_i)\right)^{1-y_i}$$

which results in a log-likelihood [213, 214]

$$LL(M) = \sum_{i=1}^{n} y_i \ln\left(p(x_i)\right) + (1 - y_i) \ln\left(1 - p(x_i)\right)$$

Using these log-likelihoods, we can calculate McFadden's pseudo-R^2 to determine the improvement attributed to using the M_{full} model instead of the null model $M_{intercept}$, so we can rank the models accordingly. Research addressing the interpretability of McFadden's pseudo-R^2 shows that the values are

lower than real R^2 values we would obtain for, e.g., linear regression because values between 0.2 and 0.4 are considered an excellent fit, but just like in the variance-based R^2 calculations, a higher R^2 value translates into a higher contribution of the model [215, 216].

For random forest, every node of a decision tree can be thought of as a condition of how to split values of a feature so that the end result remains the same for values considered similar (e.g., shall the condition be $x > 90$ or rather $x > 100$? Is there a big semantic difference between the meaning of 95 vs. 105 given the bootstrap samples?). How the condition is determined depends on a measurement of impurity, of which there are many specializations, but for classification, the metric of *Gini impurity* is used, whereas for regression, variance reduction is used by scikit-learn by default (we use the `RandomForestClassifier` class, so Gini impurity is relevant for our case). We need an impurity metric because "the importance of each feature is derived from how 'pure' each of the buckets is" [217]. Gini impurity means that we randomly choose an element from our set, classify it randomly according to the label distribution in the set and calculate how probable a misclassification is. A high Gini impurity simply means that randomly chosen elements would be frequently labeled incorrectly and vice versa, so it is desirable to minimize this metric. The actual number "can be computed by summing the probability [p_i] of each item [with label i] being chosen times the probability [$\sum_{k \neq i} p_k = 1 - p_i$] of a mistake in categorizing that item" [218].

3.2 Model Compression

Instead of or in addition to performing band selection, we can choose to compress the model for reducing its number of parameters and, hopefully, better image classification results. While many methods like parameter pruning, post-training quantization, knowledge distillation, matrix factorization etc. exist, we will exemplarily focus on the first two methods to narrow down the scope of compression combinations for the experiments. Hence, we will be considering the two popular model compression methods of parameter pruning and post-training quantization [219].

Model compression can seem like a vague generalized term at first – after all, the question arises which exact component of the model is compressed and how. *Parameter pruning* focuses on reducing the number of weight and bias parameters, i.e. removing associated connections and neurons from the network, using one of the various criteria presented so far. This means that we aim for a sparser model regarding the number of parameters. By contrast, *post-training quantization* means applying quantization after having trained the neural network, i.e. applying a different representation of weights and biases though a codebook, which saves bits and thus, storage size (if the parameters are physically removed from the network, which is not always possible for all deep learning frameworks), but does not reduce the number of parameters. Depending on the goal, certain compression techniques can be more suitable than others. Common goals include accuracy improvement, model size reduction, a lesser RAM and VRAM usage, greater energy efficiency and a faster inference time / improving latency, which is by no means an exhaustive list. Researchers have combined the two techniques in compression pipelines to take advantage of both methods' compressions [12]. The combination of compressions is something we will also evaluate in our own (different) compression pipelines. Our main objective is to **analyze the accuracy impact** by the techniques on the hyperspectral model because it is unanimously agreed that without a good OA, it does not matter how small and performant a model turns out to be when it is not suitable for the task in question. Where possible with the toolkits developed for HSI so far, we will take a look at model sizes and inference times as well and how they vary with applying model compressions. We also think it is the best decision to not define a mathematical metric as to which OA loss we are willing to accept at most – to say that, it varies too much among the experiments (models, datasets, image and model compressions), it will vary even more for different use cases we cannot all examine in advance (e.g., face detection instead of satellite images with compressed neural networks, motivated by our results), and it is the observer's task to decide which OA loss he is willing to put up with because only he knows the specificities of the platform he wants to run the neural network on regarding model size, RAM available etc. as well as his / the client's objectives.

3.2.1 Parameter Pruning

Parameter pruning in the context of neural networks is the removal of network parameters according to pruning criteria. Typically, these elements are sparse tensors, i.e. tensors containing many exact zeros. Thus, the metric of sparsity is the opposite of density. Whether or not a tensor is sparse is determined by the criterion used in the pruning algorithm. Prunable network parameters are weights (most common focus of pruning), biases (not as frequently used as pruning targets because there are few compared to weights and arguably, they have a large ”contribution to a layer’s output” [220]) and activations (rarely pruned in literature – one related work is Setiono’s in the context of breast cancer diagnosis, who has clustered the activation values of hidden units after an initial iterative weight pruning, finding that the clustering and using the activations’ discretized values did not decrease the accuracy [221]).

The purpose of parameter pruning is twofold. In early work, Le Cun et al. described parameter pruning in their optimal brain damage paper [222], followed by Hassibi et al.’s optimal brain surgeon as an extension to reduce the test error further [223] and preceded by Hanson and Pratt’s network construction minimizing hidden units [224]. Back then, the focus of employing parameter pruning in the majority of the most cited papers on this topic was to **reduce overfitting** [219, 225]. For instance, Hassibi et al. clearly say that they aimed to “improve generalization, simplify networks, reduce hardware or storage requirements [and] increase the speed of further training” [223]. This enumeration does not include accuracy considerations, although both optimal brain damage and surgeon relied on the Hessian of the loss function instead of magnitude-based pruning like weight decay precisely for the sake of a better accuracy, proceeding with a “training from scratch manner” [219]. It is in more recent works that the discovery that pruning did not have a dramatic **effect on accuracy loss** was the focus of network pruning papers; in fact, pruning often retains or improves the accuracy [12], while reducing the complexity of the network (besides, computational limitations certainly can also be attributed for the rise of pruning research since 2015, starting with Han et al.’s paper [226]). The desire to deploy neural networks on mobile devices with their computational and storage limitations also motivates the search for pruned neural networks with acceptable accuracies. The good news is that this wish is not just up in the air, but countless papers (admittedly, with different evaluation datasets, but this makes the argument of general applicability of pruning even stronger) have shown that a significant (up to 99%) chunk of network parameters can be pruned with no loss of accuracy whatsoever (see related work in 3.3.2 and 9.1.1). However, as the variety of parameter pruning types makes a disambiguation of the different types of pruning critically important, we need to get acquainted with the pruning variants.

The first distinction can be made with regard to the frequency and amount of pruning. *One-shot pruning* means that given a trained model, sparse components are pruned all at once. Han et al. point out that while effective, this method does not use the full potential pruning could offer, proposing *iterative pruning*. This is a paradigm where merely a portion of components are pruned in one go, but there are multiple such iterations, between which *retraining* (also called fine-tuning) is applied to let the neural network adjust its parameter values for the new arrangement of components to avoid an accuracy loss. The specifics of iterative pruning, e.g., parameters like the number of iterations and the pruning criteria applied for an iteration (this may vary on an per-iteration basis), are determined by the *pruning schedule* [220].

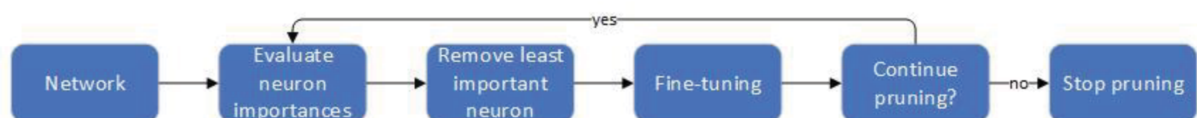


Figure 16: Own visual description of the iterative pruning procedure on a neuron-level basis heavily inspired by Molchanov et al.’s [227]. For each iteration, the pruning criteria determine the least valuable neurons, which are removed, followed by typically a few epochs of retraining [227].

Moreover, we can distinguish among pruning techniques by the philosophy of when to apply pruning. On the one hand, we can train a neural network from scratch or equivalently, apply *transfer learning* to take

an already trained set of network parameters as the starting point, followed by pruning. On the other hand, we can start with the neural network representation that resulted or would result from pruning (to find one possibility of how such a neural network would look like, we can use the first procedure) and train this compact neural network from the scratch using *sparsity constraints* which enforce the least important structures (e.g., neurons, filters) to be zero values. The sparsity constraints are "typically introduced in the optimization problem as l_0 or l_1-norm regularizers" [219]. Comparing the two approaches, multiple papers claim that **training a large network and iteratively pruning it yields better accuracies than training a smaller network with regularizers from scratch** [227, 228], e.g., Zhu et al.'s large-sparse vs. small-dense pruning analysis [229].

Another criterion for differentiating between pruning types is the *granularity* of pruning. We can decide to prune weights and biases (i.e. the connections between neurons associated with the weight parameters and biases belonging to their respective neuron) or to prune *filters* (and biases of the neurons affected), which are *output channels* of a layer, i.e. bundles of connections with respective weight parameters. The former is referred to as *fine-grained pruning*, the latter is by far the most common type of *coarse-grained pruning* alluded to in literature, but coarse-grained pruning can also refer to *structure pruning*, where a structure need not necessarily be a filter, as this term refers to distinctive arrangements of parameters in a tensor (or multiple tensors). In the experiment evaluations, the term coarse-grained pruning will refer to filter pruning. Besides, determining the impact of pruning certain tensors on the neural network is called *sensitivity analysis*. This method tells us how strongly the neural network reacts to removing particular connections or filters, i.e. how sensitive each connection or filter is.

Augasta and Kathirvalavakumar categorize pruning methods differently than we did, i.e. by distinguishing by the pruning criterion as opposed to the procedural approach or granularity used. They arrive at the categories "penalty term methods, cross validation methods, magnitude based methods, mutual information (MI) based methods, evolutionary pruning methods, sensitivity analysis (SA) based methods and significance based pruning methods", introducing examples for every category in their survey [230]. However, Cheng et al. grouped the pruning methods similarly to our categorization, although they do introduce subclassifications of filter-level pruning in their survey such as *vector-level*, *kernel-level* and *group-level pruning* [231].

Researchers have come up with both weight and filter pruning variants, which we are going to touch upon here, along with the most important findings – for more related work, please refer to 3.3.2 and 9.1.1. For weight pruning, Han et al. used sensitivity thresholds for weight pruning by using the standard deviation as a normalizing factor between weight tensors [226]. For example, for a normally distributed tensor, around 68% of elements "have an absolute value less than the standard deviation σ of the tensor", so these could be considered for pruning [232], either as one-shot or as iterative pruning. Han et al. chose the latter and "used the sensitivity results to find each layer's threshold: for example, the smallest threshold was applied to the most sensitive layer, which [was] the first convolutional layer". They found that their method resulted in "reducing 2x the connections without losing accuracy even without retraining". Furthermore, they observed that **fully connected layers were less sensitive to pruning than feature detecting / convolutional layers** and that **deeper layers were less sensitive**, which can save lots of parameters for fully connected layers [226], marking, to the best of our knowledge, the beginning of recent further research on network pruning with focus on exploring the parameter-accuracy tradeoff. Another paper of Han et al. deals with deep compression and stacking parameter pruning, vector quantization and Huffman coding on top of one another to form a pipeline, where the parameter pruning again concerns fine-grained threshold-based pruning because "all connections with weights below a threshold are removed from the network" [12]. Guo et al. propose to not only prune, but also splice connections on-the-fly "to avoid incorrect pruning and make it as a continual network", achieving satisfying results [233]. Zhu et al. vary iterative pruning towards automated gradual pruning so that many weights are pruned at first, when lots of semantically irrelevant weights exist, but fewer weights later on, according to

$$s_t = s_f + (s_i + s_f)\left(1 - \frac{t-t_0}{n\Delta t}\right)^3 \text{ for } t \in \{t_0, t_0 + \Delta t, \dots, t_0 + n\Delta t\}$$

with the initial sparsity value s_i, the final sparsity value s_f and "n pruning steps, starting at training step t_0 and with pruning frequency Δt" [229]. To go further, Narang et al. prove that pruning RNNs is possible and yields good results with a fine-grained gradual pruning schedule [234].

For filter pruning, there is also a variety of possible approaches. This is partly because we can use different criteria to rank tensors based on their importance. For example, Li et al. employ a neuron ranking by the l_1-norm of a filter's weights and prune all neurons of the feature maps of only convolutional layers using a one-shot pruning approach [228]. Yu et al. describe the reason for using one-shot pruning by claiming that a neural network's weights do not change their magnitude much after several epochs, so one could approximate the weights' importance by their initial magnitude [235]. As an alternative to l_1-norm ranking, Hu et al. used the *average percentage of zeros* (APoZ) of the activation channels as the ranking metric for pruning filters [236]. They used it "to measure the percentage of zero activations of a neuron after the ReLU mapping", defining $APoZ_c^{(i)}$ for the c-th neuron in the i-th layer as

$$APoZ_c^{(i)} = APoZ\left(O_c^{(i)}\right) = \frac{\sum_k^N \sum_j^M f\left(O_{c,j}^{(i)}(k) = 0\right)}{N * M}$$

for $O_c^{(i)}$ denoting the output of the c-th channel in the i-th layer, $f(\cdot) = 1$ if true, $f(\cdot) = 0$ if false, M being the dimension of the output feature map of $O_c^{(i)}$ and N being the total number of validation samples [236].

In contrast to ranking structures by their importance, Molchanov et al. have shown that pruning can be understood as a combinatorial optimization problem for the training examples

$$\mathcal{D} = \{\mathcal{X} = \{x_0, \dots, x_N\}, \mathcal{Y} = \{y_0, \dots, y_N\}\}$$

with inputs x_i, outputs y_i, a set of weight-bias pairs $\left(w_i^j, b_i^j\right)$ as network parameters

$$\mathcal{W} = \{(w_1^1, b_1^1), \dots, (w_L^{C_l}, b_L^{C_l})\}$$

"optimized to minimize a cost value $\mathcal{C}(\mathcal{D}|\mathcal{W})$" [227] of a cost function $\mathcal{C}$, solving

$$\min_{\mathcal{W}'} |\mathcal{C}(\mathcal{D}|\mathcal{W}') - \mathcal{C}(\mathcal{D}|\mathcal{W})| : \|\mathcal{W}'\|_0 \le B$$

"where the l_0 norm in $\|\mathcal{W}'\|_0$ bounds the number of non-zero parameters B in $\mathcal{W}'$" to obtain the ideal network parameters $\mathcal{W}'$ [227], with $\mathcal{C}(\mathcal{D}|\mathcal{W})$ denoting the conditional cost for the samples $\mathcal{D}$ given a network with the parameters $\mathcal{W}$. They used the brute force method of pruning VGG16's 4224 convolutional filters [227] to determine $\mathcal{W}'$ (i.e. oracle criterion), but since examining 2^{4224} subsets of filter pruning combinations is computationally impossible, they pruned the massive subset search tree using criteria including minimum weight, activation, mutual information, Taylor expansion, relation to optimal brain damage and APoZ to make this feasible [227]. Obviously, there is a need to consider the filters as subsets instead of pruning one filter at a time (otherwise, we could reduce the complexity from to around $\sum_{i=1}^{4224} i = \frac{4224*4225}{2} \ll 2^{4224}$) because if we wrongly considered the one filter at a time which keeps the cost value as close as possible according to the equation and firmly assumed this to be the correct filter from the beginning, we would possibly miss out on opportunities where a subsequent filter led to a combination of filters that performed better according to the equation than two filters individually would have. Intuitively, one might compare this to the credit assignment problem in reinforcement learning – as an agent, one does not know which combination of actions, i.e. of choosing filters to be pruned, has led to the reward of the greatest neural network performance benefits. Last but not least, just as automated gradual pruning is possible for fine-grained pruning, there are equivalents for coarse-grained pruning. By the way, the Intel Distiller authors provided an overview of pruning algorithms

we used for the explanation, but they also show that weight pruning algorithms can be divided differently than we did terminology-wise, so there is no one perfect classification of all pruning approaches [232].

3.2.2 Post-Training Quantization

Quantization is the process of encoding numbers, often from a large set of numbers or of a continuous set, to a smaller set so that fewer bits can be used to represent the number [237]. With this description, the motivation of **reduced bandwidth and storage** immediately becomes apparent, but as Dally has found, integer computation is both faster than floating-point calculation and more area and energy-efficient [238, 239]. In the context of neural networks, we can use quantization to use **fewer bits to represent weights, activations and accumulators**, but quantization is not restricted to neural networks. For instance, in signal processing, an analog-to-digital converter uses quantization to approximate "a continuous-valued [...] signal [with] a discrete-valued [...] signal" [237]. In the field of neural networks, however, Rastegari et al. have found that they could replace costly multiplications in convolutional and fully connected layers with only additions and subtractions for 2-bit or 1-bit representations of weights, which resulted in increased efficiency through quantization [240].

To understand the details of how quantization helps us get a compact representation of the network parameters, we need to introduce the terms *dynamic range* and *precision*. Dynamic range describes the range of representable numbers, i.e. it gives us an insight into the numerical distance between the lowest and the highest number. For integer formats, it is the interval $[-2^{n-1}; 2^{n-1} - 1]$ for n bits used (these integer formats are often abbreviated as *INT*n for a chosen number n), whereas *FP32* (floating point 32-bit) has a dynamic range of $\pm 3.4 * 10^{38}$ with $4.2 * 10^{9}$ representable values [238]. Precision determines "how many values can be represented within the dynamic range" [238]. To argue why we are interested in using INTn over FP32 at all, we need to take a look at the representation of floating-point numbers. According to the newly revised 2019 IEEE 754 Standard for Floating-Point Arithmetic, the number format for 32-bit width representation of the binary interchange format consists of 1 sign bit, 8 bits for the exponent and 23 bits for the mantissa (i.e. the fraction), while the 64-bit variant uses 1 sign bit, 11 exponent and 52 mantissa bits [241]. Hence, we notice that the mantissa accounts for many bits in both 32- and 64-bit basic formats just to represent a fraction. It is desirable not to waste as many bits for number representation for the sake of a faster calculation. Quantization defines a transformation process where the computationally and storage-wise bothersome floating-point values are mapped to low-bit indices, as registered in a codebook to keep track of the actual value behind the low-bit number, while performing calculations with the low-bit number. Since such a mapping between numbers can be performed in various ways, there is the question of how one should proceed.

Migacz has observed that there is "always a tradeoff between range and precision of the INT8 representation" when he decided to map outliers lower / greater than a threshold $\pm T$ to the minimum / maximum for the sake of a higher precision (this is called *saturation*) [242]. In his slides, he hinted that biases were not necessary to consider for quantization [242]. As for the justification, Jacob et al. explain that "quantization parameters used for biases are inferred from the quantization parameters of the weights and activations" [243]. **The challenge is to take advantage of INT8's higher throughput and lower memory requirements while not letting INT8's lower precision and dynamic range than FP32 result in a significant loss of information** [242]. Choosing the right threshold for saturation can be done on the basis of using the Kullback-Leibler divergence for the loss measurement and obtaining results from a calibration dataset, which is exactly what Migacz has demonstrated [242].

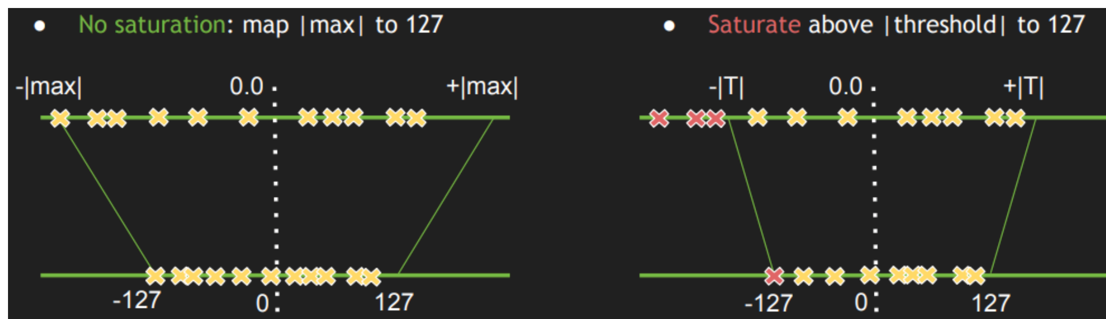


Figure 17: Migacz's illustration of saturation and its advantages in the process of quantization [242]. In this example, values below the negative threshold $-|T|$ are considered outliers and mapped to the lowest value -127 when using saturation. One chooses $|T|$ for a higher precision of the targeted values, i.e. between $-|T|$ and $+|T|$, compared to the precision between $-|max|$ and $+|max|$, which comes at the cost of not expressing the magnitude of values between $-|max|$ and $-|T|$ as well as $+|T|$ and $+|max|$, which is the compromise entailed by saturation [242].

The number distribution may vary among the layers, so it can make sense to quantize the layers individually. In this case, a *scale factor* should be used "to map the dynamic range of the tensor to the integer format range" [238]. Since the scale factor is an FP32 number, we can eliminate floating point computations using only shifts like Courbariaux et al. [244] or approximate the scale factor (often very well) with integer multiplication and a shift operation [238, 245]. We also need to consider the varying difficulties of quantizing different component types. Quantizing weights is a straightforward process as shown in Figure 18. Quantizing activations means we need to get to know their values so that a sensible quantization can be done. Therefore, there is a distinction between *offline quantization* of activations, where calibration batches are used to calculate the scale factors, with the risk of encountering out-of-range values at runtime, and *online quantization*, which operates at runtime and therefore trades the lower overflow risk for a performance overhead [238]. To find the right clipping range, we can use statistical measures like Migacz or analytically compute the clipping value like Banner et al. [238, 242, 246].

The next question is how to group the variety of quantizations available. We can classify the techniques by the point in time of applying quantization, by the number of bits used, whether we quantize all components with the same number of bits or vary them among weights / activations / accumulators or by layer type (e.g., fully connected vs. convolutional layer) and tensor. The first distinction we would like to make is *post-training quantization* and *training-aware quantization*. It is described in detail in Krishnamoorthi's whitepaper, who explains that post-training quantization does not require retraining the model, thus being simple to use [247]. That is because all one needs to do is train the model as one ordinarily would and perform a quantization on top by invoking an appropriate command once.

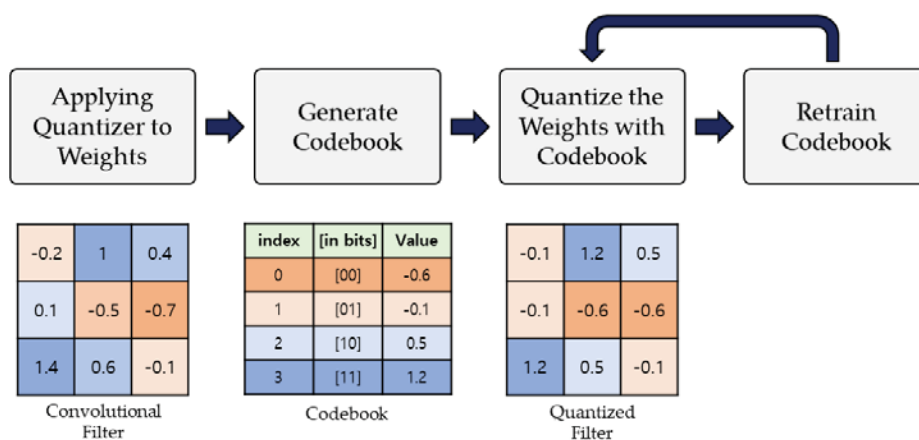


Figure 18: Iterative quantization as performed by Seo and Kim and Choi et al. by retraining the codebook can be applied for convolutional filters in a neural network. Retraining the codebook is often done in papers since it "can provide substantial accuracy improvements at lower bitwidths" [247]. We see that two bits would suffice for the representation of floating-point numbers in this example [248, 249]. Figure from [248].

Training-aware quantization works differently than post-training quantization because it takes part in the training process of the network by “automatically inserting simulated quantization operations in the graph at both training and inference times” [247]. Krishnamoorthi’s illustration (Figure 19) schematically outlines the differences of post-training quantization vs. training-aware quantization. While he has found that quantization-aware training can result in better accuracies to models with small representational capacity like MobileNet [238], post-training quantization is easier to use and works particularly well at the granularity of per-channel quantization (practically shown by Krishnamoorthi on ResNet variations, NASNet-Mobile and Inception-v3 ANNs) [247]. We will apply post-training quantization of our hyperspectral models in 4.2.3.3 with special focus on bit variations for the different components.

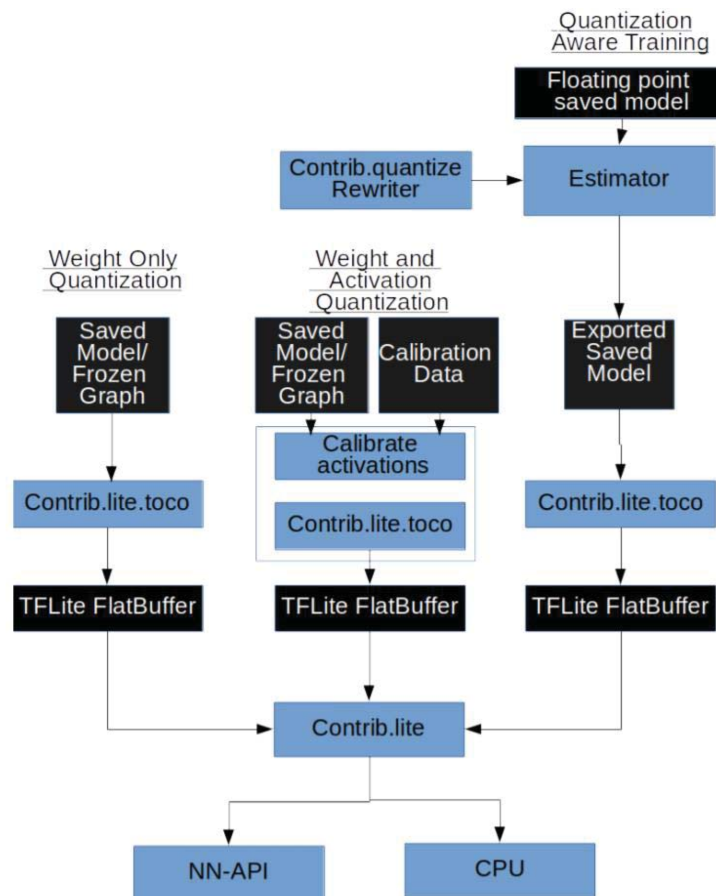


Figure 19: Krishnamoorthi’s comparison of the quantization types post-training quantization and training-aware quantization for the TensorFlow-API [247]. The TOCO converter transforms a `pb` graph file into a usually significantly smaller `tflite` file, which can be deployed on mobile platforms. Activation quantization requires calibration data so the activations can be calculated and analyzed for the upcoming quantization; training-aware quantization takes part in the training process [247].

Just like in pruning, we can ask ourselves the question whether it is better to finetune a checkpoint of a FP32 model with quantization or to train a quantized model from scratch. Just like with pruning and in accordance with the findings of Mishra and Marr [250], Krishnamoorthi also came to the conclusion that **finetuning yields better results** [247]. We can also separate *vector quantization* from *scalar quantization* according to the criterion of how large the quantization elements are. As their names say, vector quantization concerns an array of numbers while scalar quantization targets single numbers. We are only interested in vector quantization. Moreover, while we do not care about the details, we would at least like to mention that a quantization operation encodes values, but at some point, these would have to be decoded, i.e. dequantized. For example, if we perform layer-wise quantization, we need to dequantize the encoded values after each layer because each layer might be quantized differently. Together with PyTorch’s restrictions, which we will describe in the practical part of this thesis, this is why the Intel Distiller toolkit constructs two artificial layers around a layer, where the one’s sole purpose is the

encoding, the other's the decoding of the values. In fact, one could call these surrounding layers *quantization wrappers*, and it is for these reasons that the model size may not physically decrease in PyTorch with Distiller used for quantization. When we refer to quantization, we mean post-training quantization because training-aware quantization is not of interest to us.

An important detail of quantization concerns the backpropagation of gradients. Intuitively, we can imagine a quantization wrapper as a rounding function. Consequently, the gradient of this discrete-valued function is zero "almost anywhere" [238], so **during backpropagation, previous layers would not learn** (see the mathematical explanation of backpropagation in 2.2.2). This problem is circumvented using the *straight-through estimator*, which was proposed by Bengio et al., and allows gradients to be passed as they are (i.e. the strategy is to ignore the zero gradient, which would have otherwise prevented the backpropagation process from taking place at all for previous layers) [251].

Cheng et al.'s model compression survey presents the milestones of quantization [219]. Gong et al. found that "vector quantization methods have a clear gain over existing matrix factorization methods" [225]. Wu et al. quantized filters in both convolutional and linear layers, resulting in "4-6x speed-up and 15-20x compression" for the ILSVRC-12 benchmark [252]. Moving on to distinguishing quantization techniques by probably the most obvious criterion, the number of bits used, Vanhoucke et al. used 8-bit quantization for a speech recognition task with focus on data layout [253] and Gupta et al. used 16-bit quantization using stochastic rounding for training [254]. Both attempts were successful regarding the speedup and memory usage parameters respectively. More related work is outlined in 3.3.3 and 9.1.2.

There are techniques to curb the OA loss entailed by low-bit post-training quantization (e.g., INT4). Intel Distiller's documentation divides the ideas into categories with detailed explanations [238], which we are going to use as a template in the following enumeration as suggestions for further research to try to improve the accuracies:

- **Retraining** / switch from post-training quantization to **quantization-aware training**: allegedly required for INT4 or lower "to obtain reasonable accuracy" [238]
    - Zhou S et al.: start training quantized model with trained FP32 weights instead of training from scratch [255]
    - Zhou A et al.: if we start with a trained FP32 model, we can surpass the FP32-OA baseline with just 5 bits on common RGB models on RGB datasets [256]
- **Replace** the bounded ReLU **activation function** with a bounded one
    - Zhou S et al., Mishra et al.: clipping function with hard coded values [255, 257]
    - Choi et al.: learn clipping value per layer (better results) [258]
- **Change the network structure**:
    - Mishra et al.: use more channels for layers [257]
    - Lin et al.: replace FP32 convolutions with multiple binary convolutions [259]
    - Gysel et al.: dimensionality reduction fine-tuning [260]
- **Ignore first and last layer**
    - just like for weight pruning [226], they are also more sensitive to quantization [255, 258], at least below 8-bits [258], and do not contribute much to overall computation anyway [240]
- **Iterative quantization** (analogous to iterative pruning, by retraining the model after a chunk of it has been quantized) [256]
- **Mixed precisions** (just like we will evaluate in 4.2.3.3)

Other quantization classifications exist, e.g., if the quantizer is symmetric, asymmetric or stochastic [247], but this is too specific for our case as we do not intend to customize this particular behavior of Intel Distiller's quantization, although one could surely find and implement more quantization variations and criteria than the predecessor when being creative enough and present them in related work. SqueezeNet is an example that quantization is a topic worth taking into consideration as it provides "AlexNet-level

accuracy with 50x fewer parameters and <0.5MB model size", which was only possible through the quantization included in Han et al.'s compression pipeline [12, 261]. It remains to be seen if once network compression for hyperspectral classification becomes mainstream and a neural network like AlexNet emerges as a gold standard for HSI classification, similar successes will be had.

3.3 Related Work

As in 2.3, all PAR descriptions are either directly from the respective paper or reformulated by us to better put the paper in context with our work.

3.3.1 Band Selection

Many papers claim that **PCA is the gold standard for hyperspectral band selection tasks** [152-155]. As it will be important in our experiments, we will present papers based on PCA, but also include NMF, LLE and other techniques. Regarding feature selection techniques, linear regression, logistic regression and random forest will be relevant; therefore, appropriate papers will be stated. A survey with comprehensive coverage "of both hyperspectral image analysis tasks and machine learning algorithms" exists [194]. Despite the name of band selection as opposed to band extraction, both feature extraction and feature selection fall under that category in our usage of the word, as is often done [262, 263].

Feature Extraction

Principal Component Analysis for Hyperspectral Image Classification [161]

- *Problem:* size of hyperspectral datasets motivates the need to use preprocessing
- *Action:* use principal component analysis prior to hyperspectral image classification on datasets captured by HYDICE and AVIRIS sensors
- *Result:* only the **first few (e.g., 5) principal components contain significant information**, can yield 70% classification accuracy (compared to 50 / 60 original image bands for HYDICE and AVIRIS datasets respectively with 100% accuracy, but 7x / 16x the classification time)

Principal Component Analysis Applied to Remote Sensing [162]

- *Problem:* show usability of PCA for remote sensing
- *Action:* use PCA to gain information from land cover from satellite images (Gandia and Vallat)
- *Result:* the second principal component allowed detecting the presence of vegetation (Gandia) / forestry area affected by fire (Vallat); therefore, confirmed feasibility of PCA in remote sensing to extract land use information

Assessment of Principal Component Analysis (PCA) for Moderate and High Resolution Satellite Data [163]

- *Problem:* gain an insight into the feasibility of PCA for high-resolution remote sensing data
- *Action:* determine and compare principal components for satellite imagery of agricultural and wetland areas (five datasets: ETM+, IRS, SPOT, IKONOS, CASI)
- *Result:* first principal component contains the most variance, all others contain noise for moderate and high-resolution images; the first three principal components could suffice for classification tasks in agricultural and wetland areas

Dimensionality Reduction of Multidimensional Satellite Imagery [264]

- *Problem:* need to preprocess hyperspectral data due to large size while keeping important properties; there are many methods, so a comparison is necessary
- *Action:* comparison of dimensionality reduction techniques PCA, MNF (minimum-noise fraction) and LLE, e.g., with regard to linear vs. non-linear properties remaining after dimensionality reduction; proposed a variation of LLE for better locality preservation
- *Result:* in general, **PCA performs better** than other methods, but LLE-based techniques produce high-quality dimension-reduced imagery for remote sensing and require fewer computational resources and memory than alternative approaches

[Efficient Hierarchical-PCA Dimension Reduction for Hyperspectral Imagery] [265]

- *Problem:* remove redundant information from large hyperspectral data to combat the curse of dimensionality
- *Action:* use **PCA in a hierarchical algorithm** to reduce hyperspectral data to intrinsic dimensionality: split image into parts, apply PCA on image parts and combine results
- *Result:* promising results for hierarchical PCA compared to no dimensionality reduction

[Band Selection for Dimension Reduction in Hyperspectral Image Using Integrated Information Gain and Principal Components Analysis Technique] [266]

- *Problem:* high computational burden of the widely used hyperspectral data, want to reduce spectral and spatial redundancy without losing valuable details
- *Action:* propose an integrated PCA and information gain method for band selection
- *Result:* robust clustering achieved

[Spectral-Spatial Classification of Hyperspectral Image Using Autoencoders] [267]

- *Problem:* reduce dimensionality of hyperspectral image
- *Action:* **combine PCA** on spectral dimension **and autoencoder** on two spatial dimensions to extract spectral-spatial classification information
- *Result:* high classification accuracies; the method outperforms classifiers like SVM and PCA-based SVM

[Hyperspectral Image Classification and Clutter Detection via Multiple Structural Embeddings and Dimension Reductions] [52]

- *Problem:* high-dimensional attribute space of HSI data, strongly correlated samples regarding spectral signatures, nonlinear structure and uncertainty due to noise
- *Action:* employ LLE with two external structure layers: feature embedding and a layer encoding the ranges of algorithmic parameters for LLE
- *Result:* high accuracy classification results and distinctive maps of detected clutter regions

[Multilayer Structured NMF for Spectral Unmixing of Hyperspectral Images] [268]

- *Problem:* mixed pixels captured by hyperspectral sensors with low spatial resolution: want to decompose them into endmembers and abundance fractions
- *Action:* use multilayer NMF for spectral unmixing
- *Result:* algorithm outperforms previous methods (e.g., plain NMF)

[Randomized ICA and LDA Dimensionality Reduction Methods for Hyperspectral Image Classification] [269]

- *Problem:* need to overcome curse of dimensionality for HSI; ICA & LDA not suitable for non-linear transformations, but kernel methods are computationally expensive; random pixel selection to reduce complexity might miss important features
- *Action:* try randomized ICA and LDA dimensionality reduction methods with random Fourier features
- *Result:* scalable methods; non-linearities handled more efficiently; outperform kernel ICA and kernel IDA in OA, AA and computational time

Feature Selection

[Logistic Regression for Feature Selection and Soft Classification of Remote Sensing Data] [270]

- *Problem:* feature selection is important for preprocessing of complex hyperspectral remote sensing data
- *Action:* use logistic regression to predict class probabilities on basis of input features ranked according to their relative importance; logistic regression is used both for feature selection and classification purposes
- *Result:* **logistic regression can substantially reduce features** with only a small accuracy decrease

[Semi-Supervised Hyperspectral Band Selection via Sparse Linear Regression and Hypergraph Models] [271]

- *Problem:* want to use band selection for effective and efficient band selection
- *Action:* use model based on linear regression using a least absolute shrinkage and selection operator to compute regression coefficients; then compute contribution scores for each band to rank bands, perform pixel-level classification on APHI dataset
- *Result:* advantages of proposed method when comparing it with baseline methods

Random Forest Regression and Spectral Band Selection for Estimating Sugarcane Leaf Nitrogen Concentration Using EO-1 Hyperion Hyperspectral Data [272]

- *Problem:* hyperspectral data is very large and high-dimensional; want to have only the most relevant spectral features
- *Action:* use band selection with a **random forest regression** algorithm
- *Result:* algorithm **has potential** for predicting sugarcane leaf nitrogen concentration with hyperspectral data

3.3.2 Pruning

Model compression techniques have been widely applied for the **overarching theme of reducing a CNN's computational complexity** with regard to the desired computational metric (e.g., model size reduction, energy efficiency, fast inference). The most influential or similar related work regarding pruning will be shown in the following brief overview. Many other papers are categorized and summarized in 9.1.1 with regard to this thesis. As for the results, while it is trivial to prune almost the entire network, the **compression ratios we refer to are taken from models which have a similar accuracy** to the reference network or one the authors describe as "negligibly" below/above. It should be noted that all related work for model compression which we have included targets neural networks for RGB datasets, e.g., AlexNet on ImageNet, but the principle of model compression remains the same – we just need to find out in the experiments how well model compression works for our hyperspectral scenario. The lack of hyperspectral model compression papers is absolutely representative of the state of research because model compression techniques are overwhelmingly applied for the most well-known datasets to be able to compare accuracies, memory usages, energy efficiencies etc., which turn out to be RGB datasets. As far as we know, the combination of the topics "hyperspectral remote sensing" and "model compression" (not to be confused with image band selection, for which research does exist in the hyperspectral case) has not yet gained a comparable popularity.

Specifically for pruning, a curated list of resources exists called "Awesome Pruning" [273]. For both pruning and quantization, collections of papers can also be found [274-276]. Cheng et al. talk about both in their model compression survey, among other techniques like knowledge distillation and low-rank matrix factorization [219].

Weight Pruning

To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression [229]

- *Problem*: assumption of overparameterized model. Therefore, reduce the number of hidden units for energy-efficient inference
- *Action*: **gradual pruning** technique proposed, which can be easily incorporated in training process; compare large-sparse and small-dense models
- *Result*: large-sparse models outperform small-dense models, with 10x reduction of non-zero parameters and minimal accuracy loss

Learning both Weights and Connections for Efficient Neural Networks [226]

- *Problem*: computationally and memory intensive neural networks with fixed architecture
- *Action*: pruning in **three steps**: training of connections, pruning of unimportant connections, retraining the network with remaining connections
- *Result*: AlexNet compressed by 9x, VGG-16 by 13x, both on ImageNet dataset

Network Trimming: A Data-Driven Neuron Pruning Approach Towards Efficient Deep Architectures [236]

- *Problem*: reduce computational and memory costs; but neural network design is labor intensive
- *Action*: use **network trimming** to optimize network: prune unimportant neurons, because many of them are zero regardless of the input received, so they are redundant; use iterative retraining and alternate between pruning and retraining
- *Result*: high compression ratio for LeNet and VGG-16 on ILSVRC-2012 dataset

Filter Pruning

Pruning Convolutional Neural Networks for Resource Efficient Inference [227]

- *Problem*: want to improve inference time with computationally efficient pruning strategy
- *Action*: **interleave** greedy pruning with backpropagation to maintain good generalization; approximate change in cost function by the criterion of Taylor expansion
- *Result*: first order Taylor expansion criterion performing better than norm criterion or feature map activation; >10x reduction in adapted 3D convolutional filters possible (Birds-200 and Flowers-102 datasets)

with focus on the impact of sparsity:

Pruning Filters for Efficient ConvNets [228]

- *Problem*: magnitude-based weight pruning methods are impacted by irregular sparsity in pruned networks so that computation costs might not be reduced in convolutional layers
- *Action*: prune filters with feature maps to reduce computation costs; unlike weight pruning, **filter pruning does not result in sparse connectivity patterns**
- *Result*: reduced inference costs for VGG-16 by up to 34% and ResNet-110 by up to 38% on the CIFAR-10 dataset

Multiple Compression Methods

Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding [12]

- *Problem*: neural networks require lots of computation and memory
- *Action*: introduce **three-stage pipeline** of pruning, quantization and Huffman coding
- *Result*: pruning reduces number of connections by 9x to 13x; quantization represents connections with 5 bits instead of 32; AlexNet compressed from 240MB to 6.9MB on ImageNet, VGG-16 from 552MB to 11.3MB; layer-wise speedup of 3x to 4x and 3x to 7x better energy efficiency, all without loss of accuracy

3.3.3 Quantization

The upcoming links are related work for neural network quantization, with the most important papers being listed here and others listed in 9.1.2. We do not care if training-aware of post-training quantization is used, both have their use cases as outlined before. What is important here is to show off that the theoretical **principle of quantization in general has proven to be successful** in many scenarios outlined in the scientific works below. Moreover, while it is trivial to compress all parameters to 1-bit components at the cost of miserable accuracies, the results we state are always related to either no loss of accuracy or a "negligible" loss or gain in the respective authors' opinion compared to the reference neural network.

A general quantization whitepaper can be found in Krishnamoorthi's "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper" [247].

Compressing Deep Convolutional Networks Using Vector Quantization [225]

- *Problem*: large model size of CNNs due to many parameters
- *Action*: investigate vector quantization methods
- *Result*: especially for linear layers, **vector quantization outperforms matrix factorization methods**; 16-24x compression on ILSVRC-2012 dataset

Quantized Convolutional Neural Networks for Mobile Devices [252]

- *Problem*: high computational complexity of CNNs
- *Action*: speedup computation and reduce storage and memory overhead of CNNs with Quantized CNN framework, which **quantizes both convolutional and fully connected layers**
- *Result*: 4-6x speedup and 16-20x compression on ILSVRC-12 dataset

Improving the Speed of Neural Networks on CPUs [253]

- *Problem*: high number of CNN parameters
- *Action*: tutorial for reducing computational cost with regard to SSE2 instructions and x86 CPUs
- *Result*: 10x speedup of hidden Markov model / neural network over unoptimized baseline

Deep Learning with Limited Numerical Precision [254]

- *Problem*: CNNs are computationally expensive
- *Action*: investigate lower precision and role of rounding scheme
- *Result*: 16-bit fixed-point representation with stochastic rounding suffices

Towards the Limit of Network Quantization [277]

- *Problem*: minimize performance loss of quantization for a given compression ratio
- *Action*: analysis of relation between quantization errors and neural network loss function
- *Result*: Hessian-weighted measure is locally the right objective function for optimization of network quantization; connection between network quantization problem and entropy-constrained scalar quantization in information theory exists; 51.25x, 22.17x, 40.65x compression ratios for LeNet, 32-layer ResNet and AlexNet are achievable

Binarized Neural Networks

Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1 [278]

- *Problem*: reduce memory size and accesses, improve power efficiency of CNNs
- *Action*: propose binary neural networks, i.e. binary weights and activations at runtime; write binary matrix computation GPU kernel
- *Result*: nearly state-of-the-art (February 2016) results on MNIST, CIFAR-10 and SVHN datasets; custom kernel results in 7x speedup over unoptimized GPU kernel

XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks [240]

- *Problem*: want to approximate CNNs without costly multiplications and large memory usage for parameters, especially convolutional operations
- *Action*: propose binary-weight networks, where filters are approximated with binary values, and XNOR networks, with binary filters and input as well as binary operations; **convolution implemented with additions and subtractions only**
- *Result*: 32x memory saving for binary-weight networks, 58x faster convolutional operations and 32x memory savings for XNOR network

BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations [279]

- *Problem*: want faster computation for both training and testing for future deployment of CNNs on mobile devices; multipliers are the most space and power-hungry components of the digital implementation of neural networks; binary weights would replace multiply-accumulate operations
- *Action*: BinaryConnect regularizer trains ANNs with binary weights during forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated
- *Result*: state-of-the-art (November 2015) results on MNIST, CIFAR-10 and SVHN

4. Experiments

Before conducting the experiment, we would like to present the key tools we ended up using. An architectural summary of how we used which tools for our compression-related and visualization experiments can be found in Figure 65 and Figure 66 respectively.

4.1 Tools and Expansions

The machine we use for our experiments is a 64-bit Linux-based computer with an 8-core Intel Xeon Gold 6134 processor and 756GB of RAM which has a Nvidia Tesla V100 GPU with 32GB of VRAM that runs on the CUDA 10.1 drivers. We use PyCharm 2019.1.3 to run the Python programs with an Anaconda Python 3.6 environment and torch 1.1.0 installed. The tools we used are explained below, including how we used them for which purpose in the experiments and how we expanded them, if applicable.

4.1.1 DeepHyperX

Purpose: provides hyperspectral models and datasets for reference measurement including the ones we have added; expanded by us to include band selection techniques, appropriate metrics (AA in particular) as well as VRAM, RAM and time measurements and Excel logging

DeepHyperX is a framework developed by Audebert et al. that incorporates a variety of hyperspectral datasets as well as models [280]. Each of the models has been designed with a different idea in mind, as captured in Figure 20.

Model	Archi-tecture	Ideas	Epochs	Batch Size	Learning Rate
SVM		SVM (linear, RBF and polynomial kernels)			
SVM_ grid		grid search on linear, polynomial and RBF kernels			
SGD		SGD (linear SVM using SGD for fast optimization)			
nearest		k-NN with GridSearchCV			
nn		baseline neural network (4 fully connected layers with dropout)	100	100	0.0001
hamida	3D CNN	offer semantic interpretation of spatiospectral data by simultaneously processing spectral and spatial components. The 3D convolutions involved allegedly make good use "of the few samples available with fewer trainable parameters" [68]	100	100	0.01
lee	3D FCN	"jointly exploit[s] spatial and spectral features" by concurrently applying multiple 3D convolutional layers, whose outputs are combined to a feature-rich spatiospectral map; cope with the shortage of training samples by using residual learning [69]	100	100	0.001
chen	3D CNN	designed for small images, to reduce overfitting by employing l_2 regularization and dropout and for a fast inference time [54]; "a virtual sample enhanced method is proposed to create training samples from the imaging procedure perspective" [54]; reminiscent of a *deep belief network* (DBN), "in which the input is a flattened neighbor region" [70]	20	100	0.003
li	3D CNN	"views HSI cube data altogether", "requires fewer parameters", "takes full spectral bands as inputs" [51]; learns spatiospectral local signal changes using 3D kernels, "exploiting important discrimination information for classification" [51]	100	100	0.01
hu	1D CNN	five layers have been implemented "on each spectral signature to discriminate against others", similar architecture to CNNs for speech recognition; only spectral features considered [56]	100	100	0.01

he	3D CNN	jointly learns 2D spatial and 1D spectral features in an end-to-end approach "to meet the multi-scale targets in [the] spatial domain" [70]	100	40	0.01
luo	3D CNN	"one-dimensional feature maps, obtained by convolution operation on spectral-spatial features, are stacked into a two-dimensional matrix", which is "considered as an image fed to standard CNN" [71]; use XGBoost instead of output layer against overfitting [71]	100	100	0.1
sharma	2D CNN	band selection pipeline with AdaBoostSVM; CNN works on single band as a generic CNN "to improve classification performance" (flat data treatment) [281]	30	60	0.05
liu	2D CNN	semi-supervised CNN that automatically learns features from hyperspectral image data structures, motivated by the "small number of labeled samples available": CNN consists of clean encoder, corrupted encoder and decoder, where the decoder's responsibility "is to estimate the denoised version of the corrupted encoder by minimizing the difference with the clean encoder" [282]	40	100	0.001
boulch	1D CNN	"compress data into a three channel image such that it is possible to reconstruct the original" [283]	100	100	0.001
mou	1D RNN	use the intrinsically sequence-based data structure of hyperspectral pixels using an RNN "and then determine information categories via network reasoning" [113]; use activation function "parametric rectified tanh" [113]	100	100	1.0

Figure 20: Table of models supported by DeepHyperX from the very beginning. The variety of architectures and ideas seen in the table allows us to compare the suitability of the models for the hyperspectral datasets when comparing metrics, especially accuracy metrics, later on, as well as the suitability of model compressions for different architectures.

As experiments with an early stopping implementation have shown, 20 epochs for chen (instead of 400) and 100 for lee (instead of 200) suffice as any further epoch does not lead to a significant accuracy improvement, i.e. one greater than $3.2 * 10^{-3}$ and $5 * 10^{-3}$ percent OA respectively. As for the other neural networks, the values listed in the table are the default ones.

The tool supports training and testing the models on the datasets, visualizing the results in the Visdom visualization server as in Figure 61, Figure 62 and Figure 63. Furthermore, it has been designed to allow for easy model and dataset extensibility, which we have taken advantage of so that DeepHyperX fit our needs. This means that we have included new deep learning models, each with a distinct idea respectively, to find out if they fare better than the original models, which are the following:

Model	**Archi-tecture**	**Ideas**	**Epochs**	**Batch Size**	**Learning Rate**
roy	3D CNN	spectral-spatial 3D CNN facilitates feature representation and is followed by a spatial 2D CNN for reduced complexity and to learn "more abstract level spatial representation" [14]	100	256	0.001
santara	2D CNN	„fewer independent connection weights"; feed bands to parallel CNNs that "are used to extract low and mid-level spectral-spatial features", then concatenate outputs [284]	40	200	0.0005
cao	2D CNN	„patch-wise training strategy to better use the spatial information"; "new supervised HSI classification algorithm in a Bayesian framework based on deep learning and [Markov random fields]" [285]	100	100	0.001

Figure 21: Table of models which we added to DeepHyperX since these are the models that achieve the highest accuracies on the IndianPines dataset [286]. This suffices as a reason to include the models so that we can examine how suitable the ideas, architectures and hyperparameters are in practice when conducting the experiments.

Since the suggested custom dataset *DFC2018_HSI* could not be found anywhere for download at the time of writing this, we have decided to include other custom dataset loaders for *Salinas, SalinasA, Samson, JasperRidge-198, JasperRidge-224, Urban-162, Urban-210, China* and *USA* to make up for that so that we find out how the models behave for bigger datasets both in the spectral and spatial sense and for datasets with very few classes [287]. Including the *Cuprite-188, Cuprite-224* and *Washington* datasets is future work as the first two datasets have an incomplete ground truth and the image file for Washington could not be identified. The following table summarizes the selection of datasets supported by the tool or added to it, along with its respective key data we have taken from [287, 288]:

Dataset	Method and place of gathering the data	Width * height	Number of bands	Number of classes
IndianPines	AVIRIS sensor, North-western Indiana	145*145	200	16
SalinasA	subset of Salinas	86*83	204	6
Salinas	AVIRIS sensor, Salinas Valley, California	512*217	204	16
PaviaC	ROSIS sensor, Pavia, northern Italy (Centre)	1096*715	102	9
PaviaU	ROSIS sensor, Pavia, northern Italy (University)	610*340	103	9
KSC	AVIRIS sensor, Kennedy Space Center, Florida	512*614	176	13
Botswana	EO-1 satellite, Okavango Delta, Botswana	1476*256	145	14
Samson	not stated	95*95	156	3
JasperRidge-198		100*100	198	4
JasperRidge-224		100*100	224	4
Urban-162		307*307	162	6
Urban-210		307*307	210	6
China	EO-1 satellite, "Yuncheng Jiangsu province in China"	420*140	154	6
USA	EO-1 satellite, "irrigated agricultural field of Hermiston city in Umatilla County, Oregon, OR, the USA"	307*241	154	6

Figure 22: Table of datasets supported by DeepHyperX. While IndianPines, PaviaC, PaviaU, KSC and Botswana were already integrated in the tool, we expanded it to support the datasets Salinas, SalinasA, Samson, JasperRidge-198, JasperRidge-224, Urban-162, Urban-210, China and USA as well by implementing adequate data loaders. The AVIRIS sensor captures a center wavelength range from 400 to 2500nm with 10nm width, just like the EO-1 satellite, while ROSIS' center spectral range is 430-860nm with 4nm width [287, 288]. The corrected versions of IndianPines, SalinasA and Salinas are used, which means that water absorption bands have been removed by [288].

Regarding metrics, we have added the measurement of RAM, VRAM and time consumption with an Excel writer for key parts of the software (training, inference, iterative pruning) so that for repeated execution, all the measurements, including their 95% CIs, which we chose because 95% is most often chosen as the confidence level [289], can be found in a file and evaluated easily. Similarly, we backed up the Visdom visualizations and redirected the outputs into a text file in case we needed to look something up. In addition, we implemented the AA metric so that it was calculated and output along with OA and Kappa the moment the confusion matrix is displayed. It is a useful metric to identify cases where classification has gone wrong because all pixels are classified as exactly one class, since it reacts strongly to this phenomenon, which is very well visible alongside OA and Kappa. An alternative metric we could have used just as well is *mean average precision*, as used by Sharma et al. [281] and Gong et al. [225], since it would also have given us strong feedback in this regard.

As for image band selection, we have implemented the methods *PCA, IncrementalPCA, KernelPCA, SparsePCA, LDA, SVD, GRP, SRP, MDS, MiniBatch, LLE, ICA, FactorAnalysis, ISOMAP, t-SNE, UMAP, NMF* and *FAG* in DeepHyperX. It is a mix of feature selection and extraction methods with different ideas, runtime properties and impacts on the accuracies measured. Implementing the feature extraction techniques was straightforward as they selected the exact number of components provided by the user.

For feature selection techniques, whether the exact provided number of components was selected depended on the mode used. While the fastest way, i.e. backward feature elimination based on a threshold (`SelectFromModel` class), only supports a maximum number of remaining components (which means that fewer components may remain), forward feature selection (`SequentialFeatureSelector` class) and iterative backward feature elimination (`RFE` class – recursive feature elimination) both return the exact number of components, but are impractical to use because of how slow they are. We have implemented a mixed mode that uses the backward-threshold method as a basis and adds the remaining components based on recursive feature elimination to reach our goal. For feature selection, we merged the train, validation and test datasets into one dataset just to perform the selection method and split the datasets afterwards, which we called cumulative feature selection. The reason for that is that we want to avoid situations where we wish to have, e.g., 70 components, and 68/61/63 are selected for train/validation/test datasets (validation / test understandably would not work) – if 68 components are chosen for the merged set, we can pad the remaining two with our mixed method to arrive at 70 components and split the set afterwards. It should also be noted that some dimensionality reduction techniques come with restrictions regarding the number of components that make them unsuitable for our purpose, which we will elaborate on as soon as we move on to the results for the techniques.

4.1.2 Iterative Pruning

Purpose: implementation of threshold-based pruning of weights and biases that we connected to DeepHyperX in a way that the alpha parameter can be an input argument of DeepHyperX

The second tool we use is an implementation of the parameter pruning proposed by Han et al. in their deep compression pipeline [226]. The `iterative_pruning.py` file features customizable alpha parameters used to iteratively prune convolutional and fully connected layers step by step [290]. This is done by **removing weights between neurons**, so we will refer to it as fine-grained pruning. **Biases are also considered** because we are also interested in exploring high pruning percentages, which are hard to reach when all biases are kept. In the implementation, the weights and biases to be pruned are set to zero through multiplying the weights and biases with their respective mask where the positions at which weights and biases should be pruned are zero values, otherwise the masks contain one values so that the value remains. The alpha parameters can be set differently for convolutional and linear layers and also depending on the position of the layer, e.g., first/everything but first for convolutional layers and last/everything but last for linear layers, so in this example, we end up with four alpha parameters that can be varied independently from one another. This flexibility and the fact that both convolutional and linear layers are affected by this pruning method is one of the reasons we chose this approach. For instance, Han et al.'s paper shows that the first and the last layer of the model can react more sensitively to weight pruning [226], which explains why the implementation of his paper (which we used) could distinguish between the layers. Chen et al. also stress the importance of being able to "set different pruning rates for each layer", emphasizing that only few channels are required for each category and that consequently, a well-performing pruning strategy should be conditioned on the input images [291] (which we simply did by repeating the experiment until we could come up with sensible alpha value configurations for the models' layers). Another reason why we did not choose, e.g., an Intel Distiller fine-grained iterative pruning implementation, is that this iterative pruning worked out of the box, whereas our custom expansion of Intel Distiller's fine-grained pruning from supporting four-dimensional tensors to five-dimensional ones, which would have been absolutely necessary for our purpose, ended up causing dimensionality problems in many cases. Only for a few models did this Distiller expansion work, so we preferred to use `iterative_pruning.py` for fine-grained pruning. We are convinced that generalizing this expansion to all models would not be a trivial task. This expansion is not to be confused with our coarse-grained pruning efforts – our coarse-grained pruning adjustment for Distiller worked fine as explained in 4.2.3.2 and it is the model's `forward` functions that have caused problems.

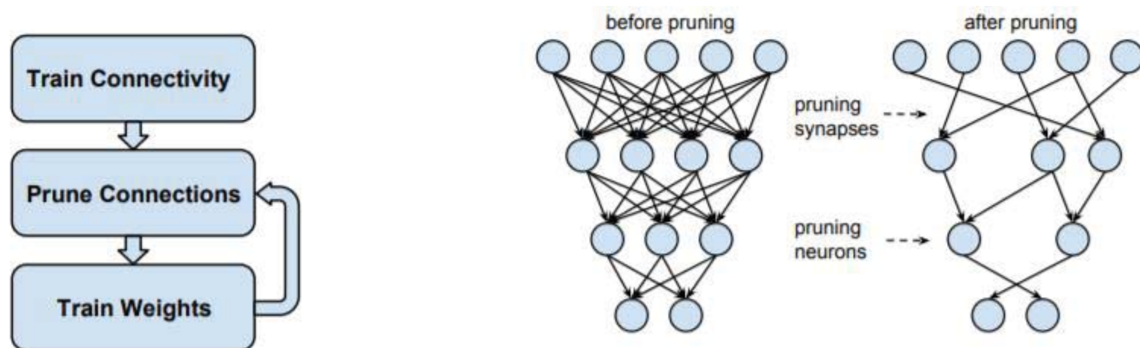


Figure 23: The pruning method outlined and illustrated by Han et al. consists of the three-step-pipeline in the picture [226]. Once connections are pruned, retraining is essential to avoid a dramatic loss of accuracy [226]. On the right-hand side, an example for a pruned network is shown. Once all connections to and from a neuron are pruned, we consider the neuron pruned as it is unreachable.

Initially intended for Conv2d layers only, we have expanded the tool to prune Conv1d and Conv3d layers as well. We have also changed the tool to support the pruning of biases, not just weights. Referring to the pruned weights and biases as “connections”, Han et al. point out in describing their pruning idea how “connections with weights below a threshold are removed from the network — converting a dense network into a sparse network” [226]. However, in this iterative pruning implementation, the affected connections are not physically removed from the network, but set to zero instead. Therefore, the model size will be calculated by determining the size of the connection affected and subtracting it from the total model size for every connection set to zero by the pruning algorithm. There is also a distinction between *Top-1* and *Top-5* accuracy the tool is able to make, i.e. whether the true prediction according to the ground truth falls within the first one or the first five guesses of the model, but we figured it was not necessary to use the Top-5 metric for anything in this thesis. The Top-1 accuracy measured by the tool has the same meaning as the OA measured by DeepHyperX in the reference runs.

4.1.3 Intel Distiller

Purpose: perform and analyze the impact of different post-training quantizations, varying the numbers of bits used for representing activations/weights/accumulators as well as channel-based pruning/thinning as a contrast to the fine-grained pruning provided by `iterative_pruning.py`. Expanded quantization to support Conv1d and Conv3d layers for this purpose. Connected to DeepHyperX to access hyperspectral datasets and models. Used WinMLRunner to obtain model sizes after quantization of all components with a fixed number of bits and TensorBoard to visualize the nodes of the quantized model

For the purposes of using quantization as an alternative to pruning for neural network compression, we use the Neural Network Distiller by Intel AI Lab called Intel Distiller, “an open-source Python package for neural network compression research” [292]. Being a complex research tool suite, Intel Distiller offers many different types of quantization, among other methods like regularization, knowledge distillation, conditional computation etc. that are not considered in this thesis. These quantization types include post-training quantization, which is what we are interested in because we would like to quantize the `pth` files of the trained models we received from DeepHyperX. There are a few post-training quantization solutions we have found for our situation in PyTorch, but no one tool exists that satisfies all of our wishes, i.e. fine-granularly customizable quantization settings, interoperability and physical reduction in model size. Intel Distiller provides the advantage that we can vary the number of bits used to quantize activations, weights and accumulators of a model independently from one another, so we can analyze the impact of quantizing each of these three components (accumulators are components that sum up the results of multiple views, or, in our case, calculations within the neural network). This is definitely its selling point as we have not seen this flexibility from other quantization tools we have been researching. We will be using Distiller’s quantization preset, which is linear and symmetric quantization (no need to adjust that because we primarily want to vary the bit numbers). However, after loading a `pth` model file quantized with Intel

Distiller, it becomes apparent that Distiller's custom naming of layers, in addition to wrapper layers inserted before and after a quantized layer, makes interoperability with other tools difficult. We would probably not need other tools if we could get an insight into the model size reduction and, optimally, inference time after quantization, but Intel Distiller has been designed "to study the effect of quantization on accuracy" [293]. To make matters worse, Distiller does not export quantized models to ONNX as of November 2019 because quantized operations are allegedly work in progress in PyTorch [294], so rather than having a model exported in ONNX, whose layers we could rename, we needed to find alternative quantization solutions that gave us an insight into model size reduction. Frameworks that promised to turn a PyTorch model into a Keras model like pytorch2keras did not work, but thankfully, the ONNX format, which in and of itself is undoubtedly a recent project with lots of work in progress, gives us a platform to export the `pth` files to (using the `torch.onnx.export` function from torch 1.2). ONNX is an open-source project introduced by Facebook and Microsoft [295], which only recently added quantization of operations in version 1.5 [296].

After exporting a `pth` file to `onnx`, we are left with the choice whether we want to quantize in ONNX directly or export the file to one of the supported formats. We have discovered the tool WinMLTools, which can be used for 8-bit integer and 16-bit IEEE 754 floating point format quantization in ONNX [297]. Using this tool, we have found out the corresponding model sizes for both quantization modes. Unfortunately, the models quantized by WinMLTools can only be read in by WinMLRunner, a "command-line based tool that can run `.onnx` or `.pb` models where the input and output variables are tensors or images" [298] and WinMLRunner is, by default, a Windows-based tool due to the `.exe` file needed to run it. Rather than using Wine to try and run it on the Linux-based test machine or modifying the source code in a Visual Studio project, trying to modify the output from `.exe` to `.out` so it can be run on Linux, we have found that WinMLRunner does not accept tensors with more than four dimensions, as we could see in the source code in Visual Studio and as we have been informed by running the program on a Windows computer for an ONNX model. **Our models for hyperspectral image classification require five-dimensional tensors**, so we cannot use WinMLRunner for inference. Nevertheless, we still obtained the quantized model sizes we wanted with WinMLTools and can refer to our Intel Distiller results as for the corresponding accuracies. We allow ourselves to do this because we have made sure that we do not deal with a lopsided comparison by comparing if the quantizations work the same way in Distiller as in WinMLTools when using the same number of bits for all components and they do for the 8- and 16-bit modes supported by both tools. We have not found other ONNX quantization tool suites like WinMLTools. We assume that is because both ONNX and quantization of models for hyperspectral datasets, at least in this combination, are novel topics that are being researched and tools built for as we speak.

ONNX models are supposed to be able to be exported to as many other deep learning frameworks as possible because that aligns well with interoperability as the purpose of ONNX being developed. Preferably, we would want to obtain a Keras model file (i.e. `hdf5` format) because we have successfully applied post-training quantization for neural networks for image classification in Keras in the past, know that it is a matter of very few lines of code and that it works. However, with tools like onnx2keras failing to convert our ONNX models to the high-level TensorFlow API called Keras, TensorFlow's `pb` file format is one of the next most promising export possibilities we assumed. Indeed, we at least succeeded in converting our ONNX models he, luo, santara and cao to `pb` with onnx_tf, whereas hu could not be converted to `pb` because of the warning given to us at the `pth` to `onnx` conversion, which told us that a feature we were using for the hu model was not supported in ONNX: "ONNX export squeeze with negative axis -1 might cause the `onnx` model to be incorrect. Negative axis is not supported in ONNX. Axis is converted to 1 based on input shape at export time. Passing a tensor of different rank in execution will be incorrect." As it is not at all trivial to convert a `pb` file to a Keras `h5` file, we wanted to explore the possibilities for working with the `pb` files.

A `pb` (or `ProtoBuf`) file "contains a complete TensorFlow program, including [the graph definition,] weights and computation", giving us all we need to run the model [299]. Consequently, we implemented

a minimal DeepHyperX framework for TensorFlow so that inference using the four `pb` files on IndianPines worked and we could measure the time needed. Our sole motivation for this was finding out the inference times for the physically shrunk models. We did manage to quantize the `pb` files with the TensorFlow API to obtain TensorFlow Lite models (ones that can be run on mobile devices) for precisely the same quantization modes described for WinMLTools earlier (16-bit floating point, 8-bit integer). For that, we chose to visualize our `pb` files in TensorBoard to try to identify the correct input and output nodes needed (e.g., Figure 64) and provide them to the quantization converter. We are happy to take the detour of `pth` to `onnx` to `pb` to `tflite` because just like we have said before, there is no single quantization tool in PyTorch that satisfies our need to measure all the metrics we are interested in. In particular, when it comes to the physical shrinking of the model, we would strongly argue that PyTorch is a bad choice because of its restrictions [300], but it was the language the hyperspectral framework was given in. Despite lots of work in progress for ONNX and model conversions from one framework to another, taking this successful detour in our case was still preferable to rewriting the entire framework for all models, e.g., in Keras. We know that because we have tried this as well and both the rewriting of three of the five models (with some framework-specific specialties, e.g., for padding) and the preprocessing of the datasets (with patch sizes, center pixels, augmentations for radiation / mixture / flip, supervision type) turned out not to be feasible. **Tensor splitting operations and parallel layers** are only really possible with a Lambda layer in Keras and even then, they **are often not possible to express** and Keras' high-level API poses significant challenges to making the models work because model-specific ideas lack the appropriate methods to express them. For example, PyTorch's layer definition in the `init` method and the specification how a tensor shall traverse the model in the `forward` function is a distinction Keras does not have. In Keras, only a functional style approach lets one access the tensor traversing the model at all and for any non-sequential model-specific features, Keras becomes a real challenge. There is a reason why, e.g., the models of Luo et al., He et al. and Hu et al. have been implemented in PyTorch and not in Keras or TensorFlow, and the DeepHyperX framework has been the platform to make that happen [301]. Should the Glow quantization framework for PyTorch [302, 303] evolve to support not just standard models on RGB pictures and datasets like MNIST, but hyperspectral models and the loading of hyperspectral data as well with an appropriate image classifier and be able to physically reduce the model size of `pth` files, we could consider recommending it as the alternative in the future once PyTorch supports more operators for quantization, but it seems unlikely for such a complex framework to quickly adjust for this new scenario, if this expansion is targeted at all.

With effort, we have also made Intel Distiller's coarse-grained pruning algorithm work for five-dimensional tensors. In contrast to the iterative weight pruning described in 4.1.2, this method "considers groups of elements which have some significance" [220]. To be more specific, we use the `L1RankedStructureParameterPruner`, which uses the "mean l_1-norm to rank and prune structures" (citation from Distiller source code comment for this class), which means that the cost function is amended by adding an l_1*-regularization term*, where "structures" means entire channels, i.e. all incoming connections to a neuron of a layer together are considered exactly one channel. Only convolutional layers (Conv1d, Conv2d, and after our adjustment, also Conv3d) are considered for this kind of model thinning (in particular, linear layers are not). There are several alternative coarse-grained pruning strategies available just for the method of pruning structures by ranking them such as `L2RankedStructureParameterPruner`, `ActivationRankedFilterPruner`, `RandomRankedFilterPruner`, `BernoulliFilterPruner` and `GradientRankedFilterPruner`, but we were happy with using this default implementation.

4.1.4 M2-DeepLearning

Purpose: this served as a basis for our minimal hyperspectral framework in Keras for exactly one easy-to-program model and one dataset (cao on IndianPines) to measure model size reduction and inference time speedups as an outlook (cf. chapter 5) what to expect once complicated models can be fully expressed in Keras and a framework similar to DeepHyperX is coded in Keras. Additionally, we used the visualization

methods of Keras-vis, but mainly our custom implementation of saliency maps, guided backpropagation and activation maps based on the deep-viz-keras framework (see 4.3).

By way of example, we took the cao model from the author's TensorFlow implementation [304] and converted it to Keras (the PyTorch visualization methods did not work for our case). We integrated the converted model in a Keras hyperspectral template we found, which is called "M2-DeepLearning" [305]. After that, we expanded the Keras implementation to support a similar channel-based pruning method to the one we used in Intel Distiller, i.e. APoZ-based pruning (because there was an implementation in Keras for that in keras-surgeon, which we happily used [306], and the differences to Distiller's l_1 pruning algorithm are insignificant for our purpose of visualization in our opinion), and added post-training quantization support (Keras' `h5` files are converted to TensorFlow Lite's `tflite` files by quantizing only the weights to 8-bit and leaving the rest in floating points [307]). Finally, we programmed several visualization methods, with the goal of finding out how these inherently unintuitive visualizations of 5x5-patches (in our opinion, because we find it hard to believe that one could see anything meaningful in a 5x5-grid) would change based on the coarse-grained pruning percentage. While Keras-vis does have a range of visualization methods available such as **`visualize_activation`** (generate the input that maximizes the filters' activations) and **`visualize_cam`** (generate a gradient-based class activation map), it is not as customizable as we wanted it to be, which we will come back to later. We have not found an easy way to generate the same visualizations for `tflite` models and will therefore not focus on visualization after post-training quantization, nor do we have time to mimic the same fine-grained pruning method in Keras, so these are possible expansions of the tool in the future. The visualizations we are talking about are three **different kinds of saliency maps** (**gradient-based, integrated gradients, guided backpropagation**) and **activation maps**. We did use parts of the same code Nagasubramanian et al. have adjusted to find out the most important bands for their hyperspectral 3D CNN with saliency maps [100, 141, 308]. As they put it, "this is the first work done on exploring the interpretation for the classification of hyperspectral data using saliency maps", and we agree that we could not find any other paper dealing with that topic, nor any other hyperspectral visualization paper applicable to our problem for that matter. Keras-vis provides exactly one `visualize_saliency` method, which allegedly "[g]enerates an attention heatmap" over the input [309], but we want to vary the method used from the "normal" gradient-based approach to integrated gradients and guided backpropagation, so we used the deep-viz-keras template [310] and adjusted it for our hyperspectral case. We will also present how a prior band selection with PCA and NMF respectively changes the outputs we get. In fact, we are going to talk about that at length because without band selection, one cannot see anything as all of our saliency map visualizations show entirely black patches for all bands and pruning percentages, which is not interesting. Out of approximately 4,500 patches of the size `patch_size * patch_size * number_of_bands`, we always choose the first such patch because it does not matter which patch we choose as long as we choose the same one for the comparison we are trying to make. Figure 66 captures our design-related considerations for the visualization pipeline.

4.2 Compressions

To analyze the impact of dimensionality reduction of images and the model respectively, numerous experiments have been conducted. We used six runs to minimize the 95% CIs (the default for this thesis, if no explanation is given) as far as we could because from experience, that is the number of runs that usually suffices for getting a meaningful statistical significance.

4.2.1 No Compression

To start with, the DeepHyperX framework is used to execute each model on each hyperspectral dataset to gather the reference data for the accuracy metrics, RAM, VRAM and time consumptions so that we can compare how these values change when we compress the model or the image bands. We also want to filter uninteresting datasets and models so that compression can proceed with only the ones that allow us to see the impact of compression. These are the criteria that make a dataset or model uninteresting:

- model is impractical to use due to **VRAM demands** that cannot be fulfilled on the test machine, e.g., the roy model on IndianPines. The dataset must not be large enough to demand such unrealistic resource requirements from many models (preferably all models work)
- model takes weeks to train for a single run, which is irreconcilable with the **time constraints** for writing this thesis, e.g., sharma on PaviaC
- multiple models show either a significant **difference between OA and AA** for a dataset or a **Kappa near zero**, both of which have been observed for the datasets Urban-162, Urban-210 and USA. The first phenomenon means that the precision (and therefore, also the F1 metric) is remarkably low for some classes, i.e. that some classes are not learned by the model at all, as opposed to knowing at least a bit about classification for each class (which the OA metric shows us and which we are interested in). The second one, in simplified terms, means that the model's categorization decisions are random instead of being conscious decisions based on the training, so the model cannot be trusted – in that case, perhaps a different approach to training would have been necessary
- a model whose **CI is too large** as this speaks against reliable reproducibility
- a dataset on which most models reach **very high or very low accuracies** – in that case, a possible improvement or loss of accuracy due to pruning would not be possible to observe
- a dataset **unknown** enough to the scientific community so that parallels could not be drawn
- a dataset for which the methods not related to neural networks (in particular, SVM, SVM_grid, SGD and nearest) achieve an accuracy so much higher than the neural networks that there is **no point in continuing to use neural networks** for the classification tasks even if pruned (since neural networks have shown to have a much higher RAM and VRAM demand compared to these methods, while not having a benefit over these machine learning methods in this scenario)
- the dataset does **not** have **enough classes** to the point that classification can be easily done correctly "on accident" (the Samson, JasperRidge-198 and JasperRidge-224 datasets seem susceptible to this)

We will proceed to present the results of executing each of the models with each dataset, both of which we outlined in 4.1.1. To anticipate the result, we will use the **IndianPines** dataset with the models **he, hu, luo, santara and cao** for the next experiments. To illustrate the dataset, a visualization is given in Figure 61, where the ground truth, a typical [43] train-test-split (80%-20%) and an RGB representation are shown. It contains 16 classes plus one undefined class, whose reflectances are shown in Figure 62. A typical confusion matrix, which DeepHyperX is able to create using the Visdom visualization server, is shown in Figure 63. The following will describe the reasoning behind that decision by analyzing the results of this reference experiment. These are partly very different from one another when varying the dataset, as we will point out. Thus, to make sure that the results of the thesis can be generalized for different hyperspectral datasets commonly mentioned in papers, all of the following experiments have been conducted with **PaviaU** and **SalinasA** as well, unless stated otherwise. Despite the aforementioned variations among different datasets, the result is that all of the conclusions drawn for the experiments with IndianPines can be generalized for PaviaU and SalinasA because the same trends have occurred.

For the IndianPines dataset, we see that the neural networks he and cao achieve the best results regarding the accuracy metrics, whereas the models hamida, li and mou suffer from an overly large CI. That is because for each of these three models, one of the six executions resulted in an outlier regarding accuracy. To provide a level playing field for all models, we decided not to set this number of executions as high as necessary until the CIs for all models would become low. It would be unfair to the other models that performed well; besides, to fulfill the scientific principle of reproducibility, we believe that it is the only rational decision not to proceed with these three models tainted with one outlier execution each. Apart from these outliers, it can be said that the models which do not perform well can be found at around 30% OA whereas suitable models lie at 80% OA. These are very good conditions for detecting a possible optimization or deterioration through the further experiments, e.g., model pruning. Generally speaking, OA, AA and Kappa are near each other for the IndianPines dataset and the *machine learning*

methods (by which we mean SVM, SVM_grid, SGD and nearest) perform mediocre with regard to accuracy, which is what we want since that gives us a reasonable argument as to why one should use neural networks at all. Last but not least, the roy model asked for 2.87 GiB additional VRAM while having 20.61 GiB already allocated, resulting in not being able to run.

As can be inferred from the graphs below, we decided not to include the VRAM and RAM consumptions for the machine learning methods, nor did we plot a model size for them. The reason is that these methods do not consume any VRAM and only need so little RAM that plotting it would reduce the visibility of the RAM usages of the neural networks, which we are primarily interested in, with the same reasoning applying to the model sizes. The VRAM usages and model sizes of the neural networks both did not vary at all, so there are no CIs attached to these two metrics in Figure 25 and Figure 27 respectively.

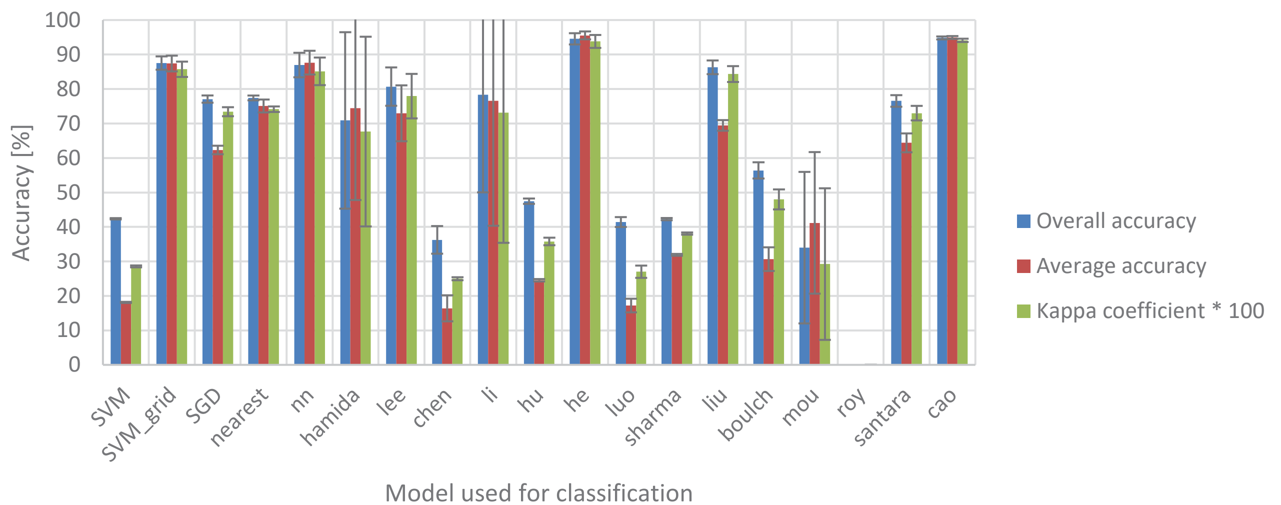


Figure 24: Model accuracies for the IndianPines dataset (n=6, CI=95%). We see the split of models performing reasonably well at around 80% OA vs. ones that do not (30% OA roughly). The size of the CI of hamida, li and mou disqualifies these models from the race, while he and cao perform the best with regard to accuracy.

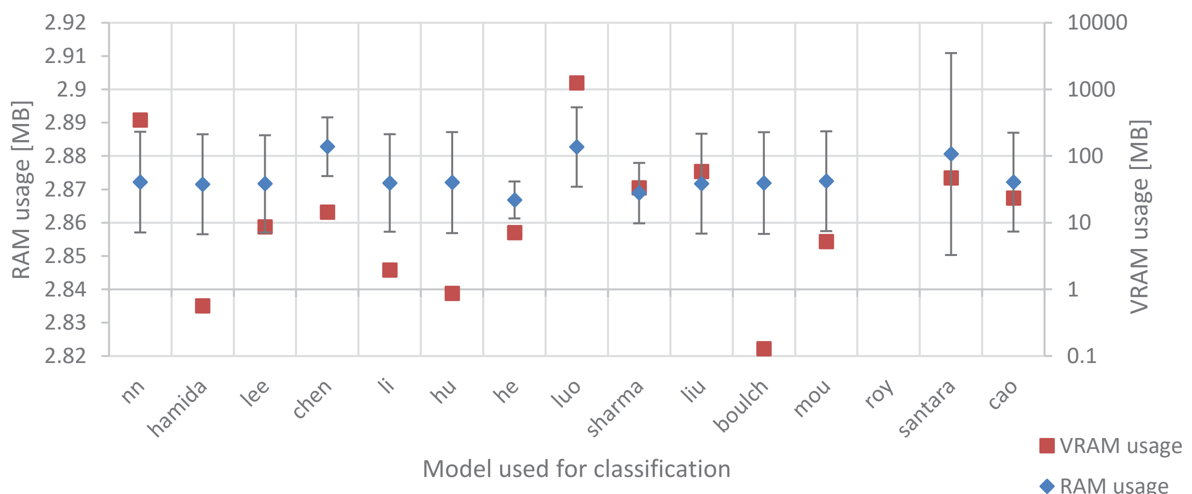


Figure 25: RAM and VRAM usages of the models on the IndianPines dataset (n=6; respective VRAM usages remain constant among the runs; for RAM usage, CI=95%). While all models consume roughly the same amount of RAM, we do not observe a correlation between VRAM usage and any of the accuracy metrics in Figure 24, with luo and nn consuming the most VRAM. The roy model asked for too much VRAM to be executed.

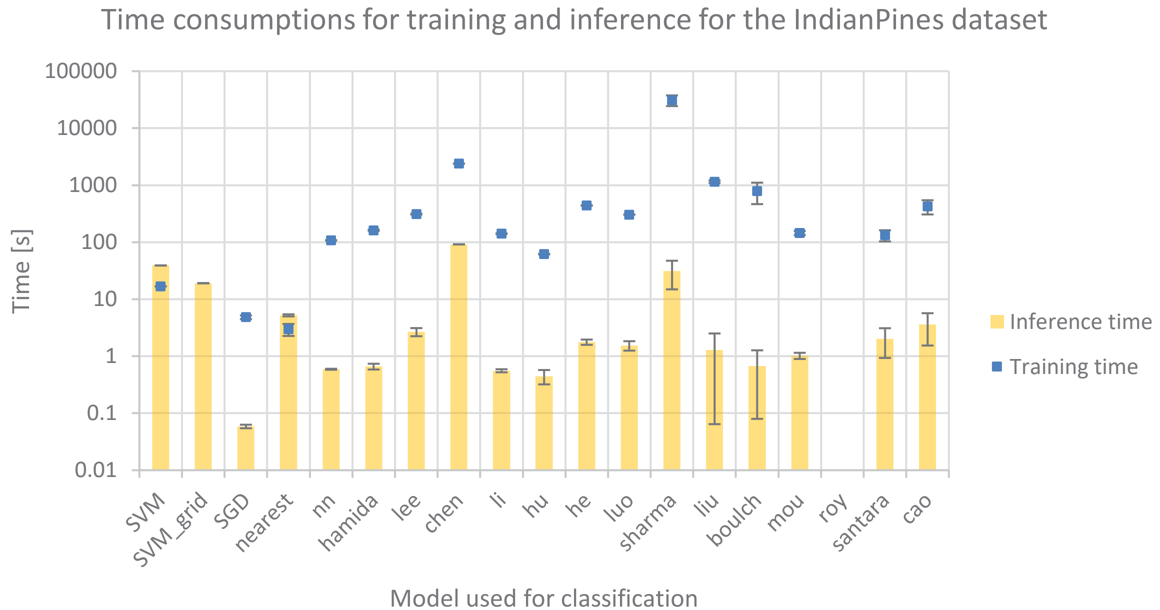


Figure 26: Time consumptions of the models on the IndianPines dataset (n=6, CI=95%). We generally see a strong trend for the neural networks that the higher the training time, the higher the inference time and the other way around. There is no correlation between either time and any accuracy metric from Figure 24, though.

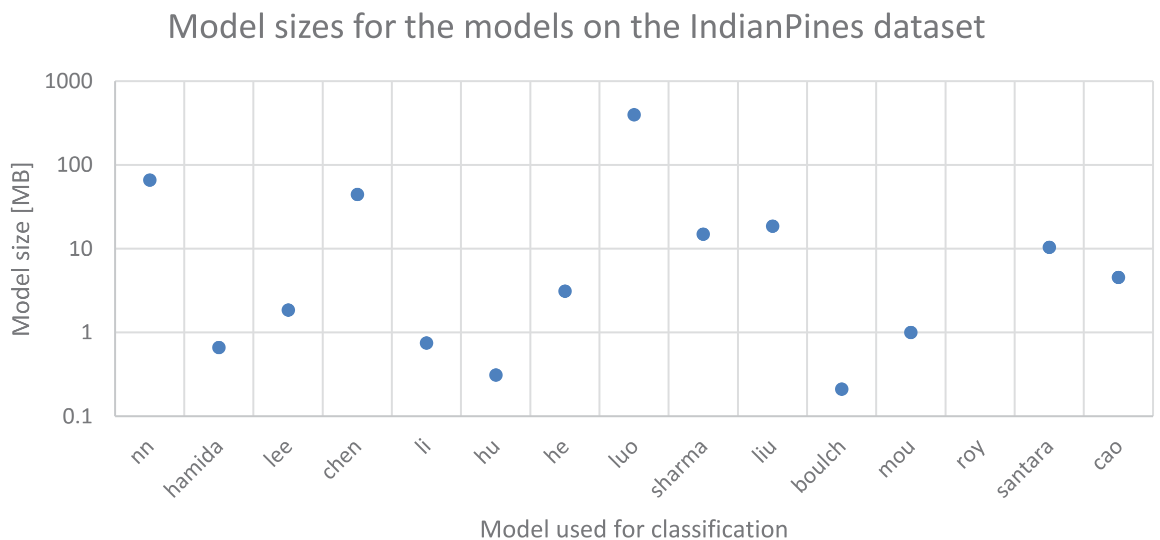


Figure 27: Model sizes on the IndianPines dataset (no variation, so no CIs). We see a strong correlation between model sizes and the VRAM usages from Figure 25. This makes sense because the tensors that are passed through the model are stored in VRAM so that the model can be executed on GPU instead of CPU for inference (because it is general knowledge that this is faster and the usual thing to do for neural networks, following appropriate experiments [66]). Moreover, the model sizes and time consumptions from Figure 26 also seem similar regarding the position of the data points, but that is just a general correlational trend that does come with exceptions, such as luo vs. sharma and santara vs. cao.

Throughout all datasets from Figure 22, we observe that the models' VRAM usages look extremely similar. While the absolute values of VRAM consumption vary according to the dataset used, the relative amount of VRAM one model consumes compared to another is roughly the same. Below is a suitable graph to back up this statement by showing the similarities in the relative distributions of VRAM consumptions compared to the VRAM usages for the IndianPines dataset in Figure 25.

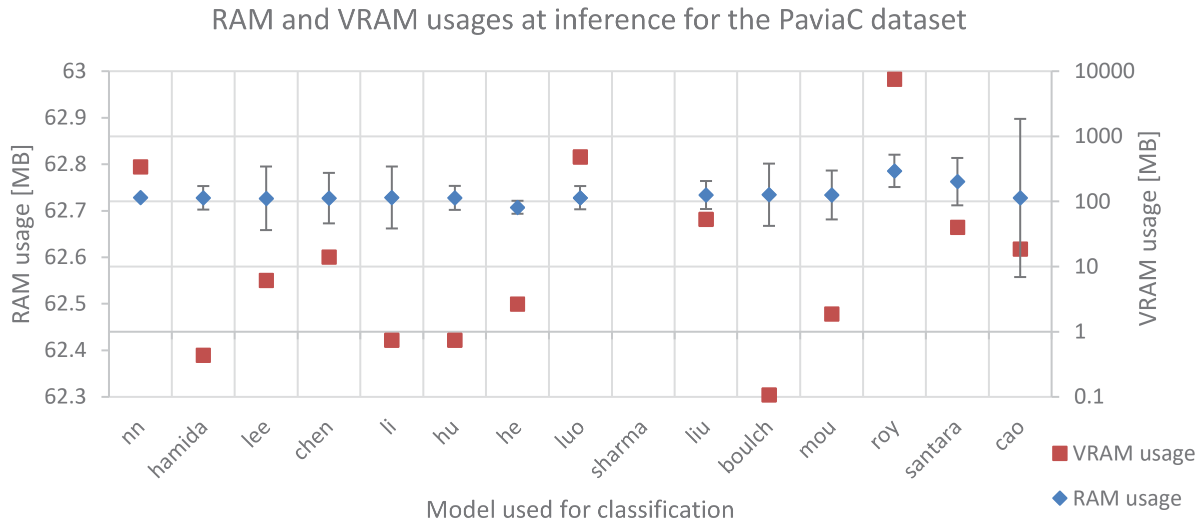


Figure 28: RAM and VRAM usages of the models for the PaviaC dataset (n=6; VRAM usages remain constant among the runs; for RAM usage, CI=95%). We witness the same pattern as in Figure 25 for VRAM consumption, which backs up our claim that the relative distributions of VRAM consumptions do not or only hardly vary across different datasets. The sharma model is not feasible to use for this dataset due to excessive runtime.

Regarding training and inference times, we have also observed that one and the same pattern prevailed throughout all datasets. Even the absolute numbers did not vary much. Our guess is that the two time metrics are highly model-specific, being much less dependent on the particular dataset used.

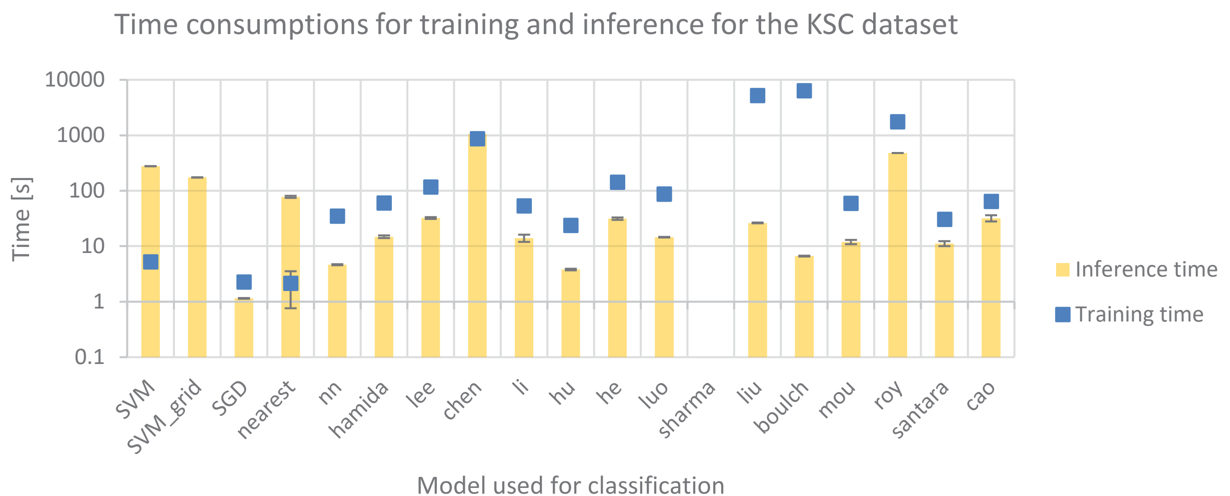


Figure 29: Time consumptions of the models for the KSC dataset (n=6, CI=95%). The same pattern as with the IndianPines dataset in Figure 26 can be observed, which shows us that it makes sense to think of training and inference times as parameters bound to the model rather than a dataset.

As for the presumably most important metric of accuracy, where no easy correlation to a different measured metric could be observed, most of our findings were very different depending on the dataset used, as we will point out in bullet points for the sake of clarity and brevity, with graphs below to back up the respective statement (">>" means "much greater than", analogous to "<<"):

- ***SalinasA***: machine learning methods prevail over neural networks, so there is little reason to optimize the neural networks for this dataset. The chen model performs awfully at around 18% OA while sharma and roy cannot be executed

- *Salinas*: OA of well performing models at around 90%, luo at 40% OA; neural networks have the edge over machine learning methods, roy asks for too much VRAM
- *PaviaC*: OAs are too good at around 98% for all models but luo, SVM and roy, so any optimization through pruning would not be visible had we chosen this dataset (see Figure 67)
- ***PaviaU***: better OAs than IndianPines, amounting to roughly 90%, but still optimization potential. Would have been the second choice after IndianPines
- *KSC*: extreme disparity between well-performing models (around 90% for SGD, nearest, lee, he, liu, roy, cao, santara) and poorly performing ones (18% OA for hamida, chen, li, hu, luo, mou); AA for first group is the lowest of the three accuracy metrics, while for the second group, OA > AA > Kappa holds
- *Botswana*: same trend as KSC with different models (chen, hu, luo perform badly, luo, li, roy, cao, santara, boulch particularly well), but OA, AA and Kappa are close to one another in the overwhelming majority of cases (see Figure 68)
- *Samson*: roughly 80% OA for all models but chen, mou and roy; OA > AA > Kappa
- *JapserRidge-198* and *JasperRidge-224*: same trend as Samson; roughly 70% OA for all models but chen, luo and sharma
- *Urban-162* and *Urban-210*: roughly 50% OA for well performing models, OA >> AA > Kappa (see Figure 69)
- *China*: varying OA between 90 and 97% for the majority of models, but the dataset does not seem to be well known or used often
- *USA*: same as China but with OA between 80% and 90%; OA >> AA > Kappa (see Figure 70)

To summarize the reasons for our choice of the models **he, hu, luo, santara and cao** in a simple way, these are the arguments that stand out (in addition to what has been already said):

- their accuracies serve as a **good starting point**: they are good enough to justify the choice of the model and bad enough to be able to be optimized by, e.g., pruning
- the differences in model size and therefore in the number of parameters, i.e. the number of layers or their dimensions regarding input and output size allow us to see **how pruning can help for both small and big models**
- **different architectures** are covered: 1D, 2D, 3D CNNs. We do not think it is a problem that we focus on CNNs because they simply perform well and allow us to focus. In particular, the muo RNN performs nowhere near well enough to justify further investigation.

The visualizations in the appendix allow us to make ourselves familiar with what makes the five models special in the architectural sense (cf. 9.2). To remember the design decisions for the upcoming experiments more easily, we will briefly summarize the respective ideas behind the models here:

- he (Figure 54): tensors are split into four parts and added twice
- hu (Figure 55): simple 1D model
- luo (Figure 56): extensive reshape operation involving concatenation
- santara (Figure 58): separate convolutions for the split tensors, followed by concatenation
- cao (Figure 57): simple 2D model

Since the correlation between model size and VRAM is very strong in the sense that the relative VRAM usages of one model compared to another can almost be inferred and RAM usage hardly varies, we will not proceed to measure VRAM and RAM in the following experiments and focus on the other metrics instead, in particular, accuracy and model size.

4.2.2 Image Channel Compression

Image channel compression means that we are only interested in keeping few of the many image bands (cf. 3.1) which the respective extraction or selection method deems most expressive according to its ideas. Using the feature extraction and selection techniques we had implemented in DeepHyperX on the training, validation and test datasets, we went on to analyze how this dimensionality reduction affected the accuracies of the models. We chose to vary the **number of components from 10 to 190 in steps of 10** components so we could hopefully observe a trend. The following two subsections will focus on OA considerations; to see the possible model size savings due to band selection, please refer to Figure 43.

4.2.2.1 Feature Extraction Techniques

Among the feature extraction techniques we have tried, **PCA** stands out as the most reliable one to use. It varies very little depending on the number of components remaining, constantly achieving accuracies that surpass the reference OAs, either by far in cases where the reference OAs were not the best (cf. santara 76% OA jump to 90% range, hu from 47% OA to 75-80%, luo from 41% to 80%), or by little when the reference OAs already were in the high 90% ranges (cf. cao 94.8% to 95-96% OA, he 94.5% to 96-97% OA). Judging from our results, PCA is undoubtedly the winner and it is understandable why it is so frequently used in dimensionality reduction research, especially for the hyperspectral case. PCA's variations IncrementalPCA, KernelPCA and SparsePCA and similar techniques like SVD perform just as well. Its top contender **NMF** does not deviate more than 3% from PCA for high reference OAs, improving them by 2% for he and cao. The average reference OAs have only been reduced by little by NMF, e.g., 5% for luo and 12% for hu, whereas santara got a 20% OA improvement. Next comes **LLE**, whose feasibility heavily depended on the model: while the improved OAs of santara and cao come close to PCA and NMF, the Kappa near zero reveals LLE is unusable for he, hu and luo. One might have assumed that the non-linear structure of hyperspectral data was conducive to make LLE the best technique in terms of OA or that PCA's non-linear variation KernelPCA outperformed PCA [52], but neither event has occurred.

Now that we have dealt with well-performing feature extraction techniques, it is time to mention which ones we advise against choosing. With Kappa near zero, the projection-based, non-linear techniques UMAP, t-SNE and ISOMAP perform awfully for all components for all models. We can confirm the issues Bachmann et al. have raised regarding ISOMAP's long runtime for datasets as large as the ones we have used (which has led to them developing a hybrid approach) [53]. Interestingly, the UMAP paper itself suggests using PCA and NMF if strong interpretability of the data is important because their axes have a meaning [179]. There are claims that UMAP, t-SNE and ISOMAP are suitable when wanting to reduce the dimensionality for visualization purposes [179, 311], but when setting the number of components to, e.g., three, the results we received were just as bad, from which we can conclude that these methods just are not suitable for our case of dimensionality reduction. They perform just as bad as using the boulch model as an autoencoder, with the output having three components (OA around 10%, AA in one-digit figures), where PCA would have achieved 94% OA / 90% AA. To be fair, the boulch model was always meant to visualize hyperspectral data with by obtaining the RGB bands [283]. If one wants to use autoencoders for dimensionality reduction, the literature advises to use stacked autoencoders for the hyperspectral case as they have proven to be successful [267, 312], which the boulch model is not. Other techniques unusable because of their accuracies and Kappa equal to zero include GRP (Gaussian random projection), SparseRandomProjection, Multidimensional Scaling (besides having terrible runtime properties and massive memory usage), MiniBatchDictionaryLearning, ICA, FactorAnalysis and FeatureAgglomeration. LDA only allows for n_classes – 1 columns (i.e. 16 components for IndianPines) and allows for very mediocre OAs (around 55%), which is too restrictive and not good enough. Cao et al. point out how "PCA and LDA fail to address the high-order dependencies" due to being "incapable of handling the nonlinear relationships" [262], but this really does not seem to matter for PCA, while the moderate LDA performance must be attributed to something else (otherwise, PCA would have suffered, too – but PCA's orthogonal transformation has worked out well, while LDA's attempt to "find a linear combination of features that characterizes or separates two or more classes of objects or events" [262] has not).

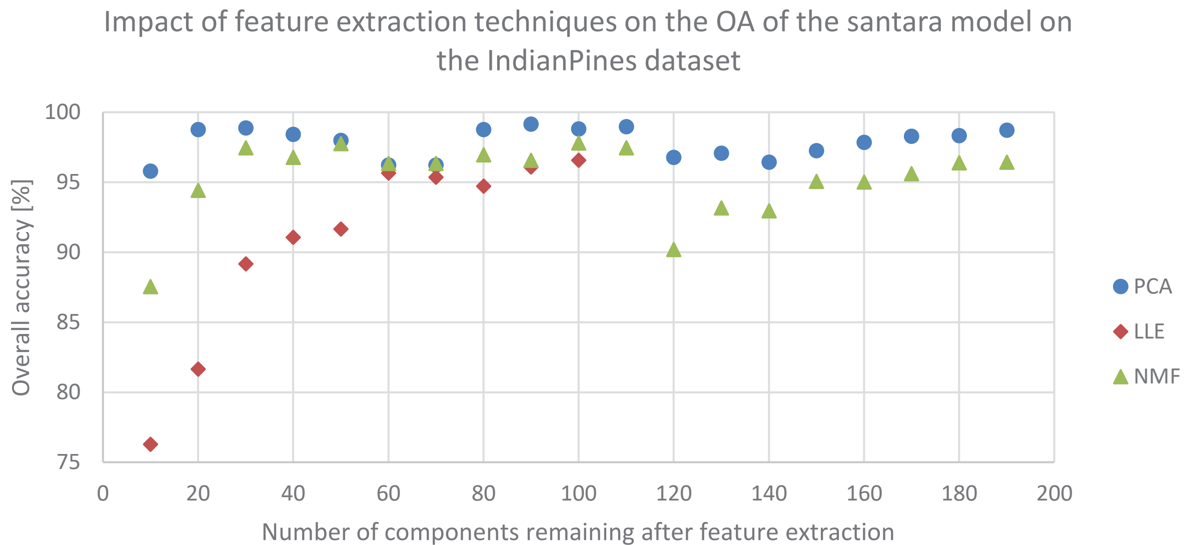


Figure 30: OAs of the santara model after applying the best-performing feature extraction techniques in terms of OA on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). PCA constantly scores OAs over 95% (works reliably), while with a growing number of components, the OAs for LLE and NMF grow until the dent at 110 remaining components. Even for just two components, PCA achieved 95.36% OA (not shown) and only at a single component did the OA disappoint (52.19%). LLE only works at all until 100 components. Almost all of the visible OAs lie above the reference value of 77%.

The fact that PCA performs very well starting at just two components is backed up by similar experience from Rodarmel and Shan [161], Estornell et al. [162] and Genç and Smith [163] (see 3.3.1). An observation from Figure 30 which catches the eye is the **dent** at 120 components for PCA and NMF. It happens on all five models, with the variation intensity order being "cao < santara = he = hu << luo". For luo, PCA varies from 90% OA between 50 and 90 components, falling to 75% OA at 100 components, to a mere 23% OA between 110 and 140 components, before recovering to 77% OA at 150 components and around 85% for more components. While we are not sure if it accounts for this massive amount of variation, 110 / 120 components might be the point where the factor of confusion by too many redundant bands described by Sarhrouni et al. comes into play [145]. From the poor performance of the non-linear methods UMAP, t-SNE and ISOMAP for CNNs of all dimensionalities (i.e. 1D, 2D and 3D CNNs), we can conclude that non-linear features apparently do not play a role for the IndianPines dataset (as well as PaviaU and SalinasA) when it comes to classification, regardless if it is spectral, spatial or spatiospectral features we are talking about. What is all the more surprising, though, is that the also non-linear LLE method fares so well. As it seems, the strategy of using "local symmetries, linear coefficients and reconstruction errors" (LLE) as opposed to the preservation of "geodesic distances between general pairs of data points" (ISOMAP) [174] or, in other words, the focus on using local dependencies as opposed to global ones for the calculations pays off in terms of accuracy.

4.2.2.2 Feature Selection Techniques

We have tried commonly used feature selection techniques by using scikit-learn's classes RandomForest, LogisticRegression and LinearRegression. None of the techniques are promising enough to recommend their usage for hyperspectral dimensionality reduction. In contrast to the feature extraction techniques, especially in Figure 30, the trend for feature selection techniques is that OA scales with a higher number of components in an approximately linear trend, whereas PCA and NMF in particular cope very well with few components (e.g., 10 out of 200), sharply increasing the OAs in the beginning until about 5 components out of 200, followed by stagnation in their 10% ranges. There are exceptions for that with the LogisticRegression method on the he and cao models, where the highest accuracies are reached with 40 / 30 components, dropping approximately linearly afterwards. For these exceptions, 90% / 94% OA are very decent OAs, but they do not surpass the respective reference OAs. The problem is that one would have to resort to **trial-and-error to find the number of components at which acceptable accuracies like**

these are reached because there is no one pattern for all models (as stated, for the other models, the pattern is the opposite). For all non-exceptions, considering the actual numbers of the OAs we reach with feature selection techniques, the three methods are only worth applying when wanting to prune very few channels as far as we are concerned. This is exacerbated by the fact that the reference OA of the santara model (76.53%), which we have not drawn in Figure 31 so our results remain visible, lies above all values plotted, which is a pattern we have encountered for the other four models as well.

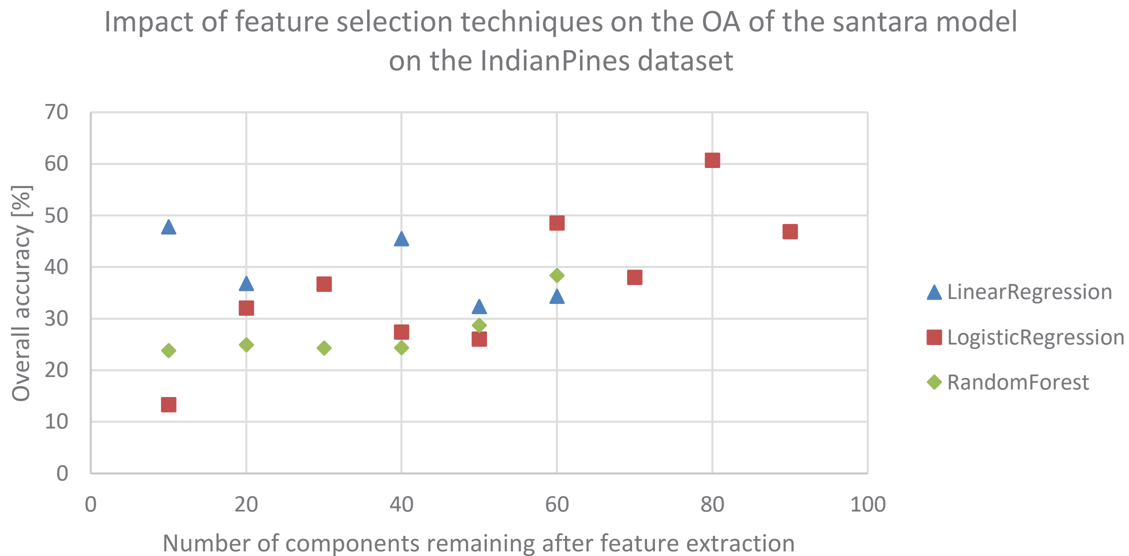


Figure 31: OAs of the santara model after applying feature selection techniques on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). All three techniques perform better when more components remain according to their roughly linear trends, but even then, none of them come close to the reference OA of 77%, with RandomForest and LogisticRegression crossing LinearRegression's downwards trend at 60 components. Beyond certain component numbers, the techniques performed too unreliably to justify including the values of, e.g., LinearRegression and RandomForest at 70 components and more (grave fluctuations between 5% and 20% OA).

Individually judging the feature selection techniques, the aforementioned LogisticRegression method can provide usable OAs, but the OAs are unpredictable with regard to both the number of components and the model and jumps around a lot. Still, it is better than LinearRegression, which jumps around less, but gives just bad accuracies, and RandomForest with a Kappa of zero and totally random OAs as seen in the confusion matrix (everything is classified as one class, which Kappa is very good at expressing in form of a metric). There are rare cases like the cao model, where LogisticRegression manages to come very close to the reference OA and even surpass it at 10 and 30 components respectively, while LinearRegression and RandomForest are stuck at 20% OA, but all OAs vary a lot. As a reminder, these are means from six executions we are talking about and even they vary a lot – if we had shown the 95%-CIs in Figure 32, they would have been large for linear and logistic regression. Increasing the number of runs would not have helped in this regard.

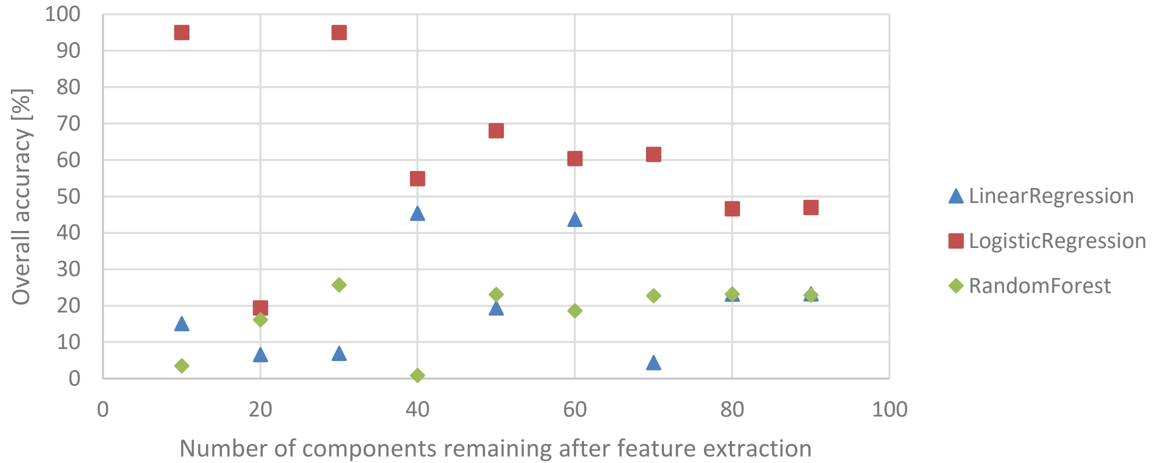


Figure 32: OAs of the cao model after applying feature selection techniques on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). The OAs vary a lot for linear regression, but this technique overall reaches the same OAs as random forest. Surprisingly, the more bands we select with logistic regression, the worse the OA gets, but the fact it surpasses the reference OA at 10 and 30 components we think is remarkable, although the high OAs come at the cost of an uncertainty we did not have for feature extraction techniques like PCA to this extent.

Just to take random forest as an example, one possible explanation for the linear OA development, whichever way it goes (in most cases, a mostly linear increase happens, but we have also just seen a decrease) could be the fact that the importances of wavelengths could be distributed linearly as well, as Abdel-Rahman et al. have found for their hyperspectral remote sensing dataset [272]. Consequently, selecting a few image bands more would presumably impact the OA linearly as well. This assumption is exactly what Li et al. can back up for the IndianPines dataset, i.e. that “when selecting the optimal combination of [8 to 12] bands”, their classification accuracy has risen approximately in a linear fashion [11]. We notice the difference to feature extraction techniques the moment we consider that the first principal component of PCA has been shown to account for very much variance, with the second one adding only little more importance [161, 163], where PCA shall function as a representative for the well-performing feature extraction methods PCA, NMF and LLE we have seen (although the other techniques obviously do not necessarily have the same focus of retaining variance, to say the least). Zhan et al. have also conducted experiments with different feature selection techniques (discriminative weighted band selection – DWBS, Ward’s linkage strategy using mutual information - WaLuMi, maximum-variance PCA – MVPCA, band selection CNN – BSCNN) and an SVM classifier, but in their illustration of the classification accuracy depending on the number of bands for the range between 15 and 50 bands selected, the trends for DWBS, WaLuMi and MVPCA are by and large linear [313].

To summarize, PCA and NMF are the most promising band selection techniques from our experiments. Lots of other feature extraction techniques fail to live up to the reference OAs. The OAs of three important feature selection representatives we have analyzed are too low and their results very uncertain, so we do not recommend these feature selection methods for hyperspectral dimensionality reduction, not least because we are fine with not keeping the original data but transforming it (a selling point of feature selection over extraction, which is useless in our case) and because transforming it using PCA and NMF can be done in a negligible amount of time. For a brief overview, we will also list these results from 4.2.2 in tabular form in our conclusion (chapter 6) in Figure 53. We will now move on to the topic of neural network compression.

4.2.3 Neural Network Compression

4.2.3.1 Fine-Grained Pruning

For the fine-grained parameter pruning, we have adjusted the alpha parameters as required to get different pruning percentages, which we aimed to distribute among the layers as equally as possible (notwithstanding different advice, since varying alpha values for different layers would have resulted in too many variables, thus hindering comparability), and tested how the model behaved to possibly witness a trend. That way we saw how much pruning was acceptable with regards to the loss of accuracy and where it began to suffer substantially. We needed to consider the model structure and distribution of parameters among the layers of he, hu, luo, santara and cao because the assumption that we would only need one alpha parameter for everything only holds true when all prunable layers of a model comprise a comparable number of parameters. This was the case for cao, hu, santara and roughly also for he, but not at all for our largest network luo, where the 100 million parameters of the layer `Linear-3` make the remaining 40,000 parameters negligible in terms of quantity (one might assume they would not in terms of quality, i.e. that the 40,000 parameters would carry a substantial semantic meaning for the network, but knowing our results, that is not the case). Therefore, we varied the alpha values differently depending on the model while using as few as can be sensibly justified based on the parameter distribution among the layers.

For **cao, hu and santara**, we iterated the single alpha value from 0.1 to 2.0 in steps of 0.1. We got the result that until a pruning percentage is reached, no accuracy losses occur. In fact, for cao and santara it is obvious that the **OA is even better until 40% pruning** are reached. After that, OA declines increasingly sharply. This is a trend that resembles the findings of Han et al., who also observed similarly shaped accuracy curves for their fine-grained model pruning and whose ideas the iterative pruning tool we used is based upon [226] (in particular, the santara model curve is strikingly similar). Zhu et al. attempted the same iterative weight pruning (using masks for weights and biases to set them to zero, just like us), coming to the conclusion that “a 50% sparse model performs just as well as the baseline” and that a “near-catastrophic degradation” of accuracy happens at around 87.5% in their evaluation of whether or not pruning is actually worth doing [229]. This applies very nicely to our experiment as well, so the fact that we are dealing with models for hyperspectral datasets as opposed to RGB images, which both of these papers use, does not seem to matter here, apart from these two percentages being slightly different in their absolute values respectively, but we witness the same trend. If we adjust the pruning percentage to not cross the OA reference line shown in the following line plots, we save 1.78KB, 0.07KB and 4.49KB in model size for cao, hu and santara respectively for total sizes of 4.56KB, 0.31KB and 10.44KB. These and all of the other model sizes are calculated by us based on the number of parameters decreasing after the pruning steps, as opposed to directly obtained through physically reduced model files in size, unless stated otherwise.

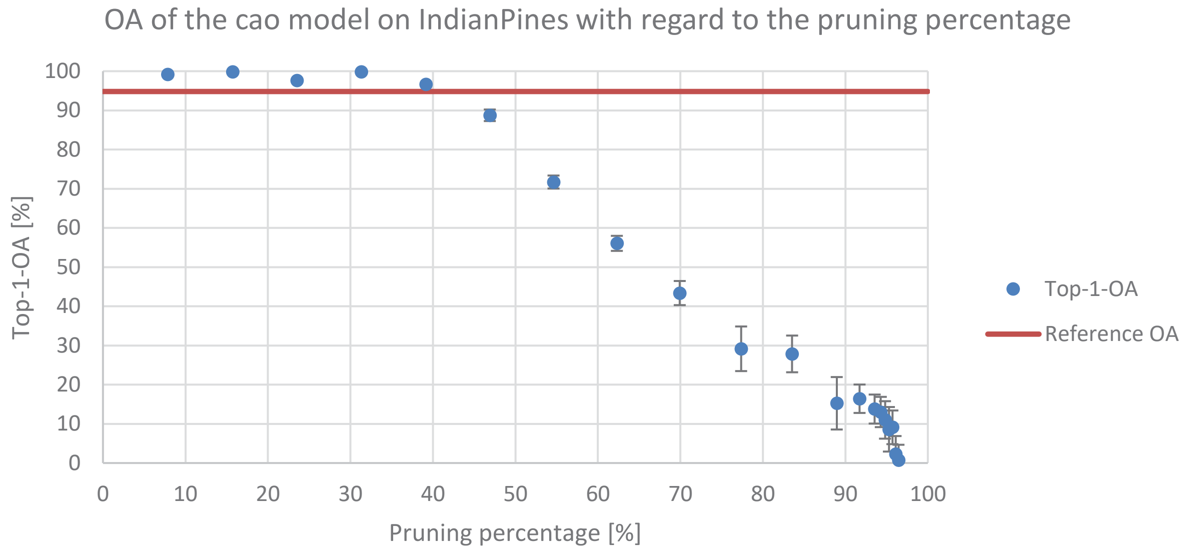


Figure 33: OA for the connection-wise threshold-based pruned cao model (n=6, CI=95%). Despite having a high base OA, the OA after pruning crosses the line at about 40% pruning percentage and the OAs we received at a lower pruning percentage are better than the OA of the original model. We see an approximately linear decrease until 77% and a sharp decline in OA from 93% pruning onwards.

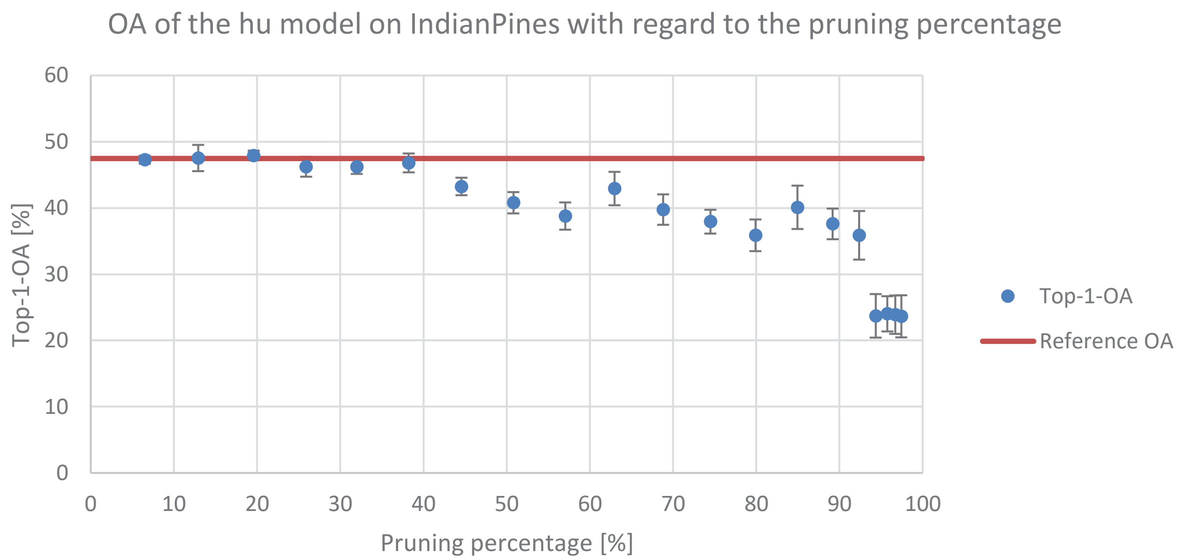


Figure 34: OA for the connection-wise threshold-based pruned hu model (n=6, CI=95%). Just like in Figure 33, we can identify three phases: an OA similar to the reference model between 0% and 38%, a slight decline afterwards until around 90% and an unpredictable decline of OA in the end, which makes the usage of a hu model pruned to a percentage greater than 90% unreliable.

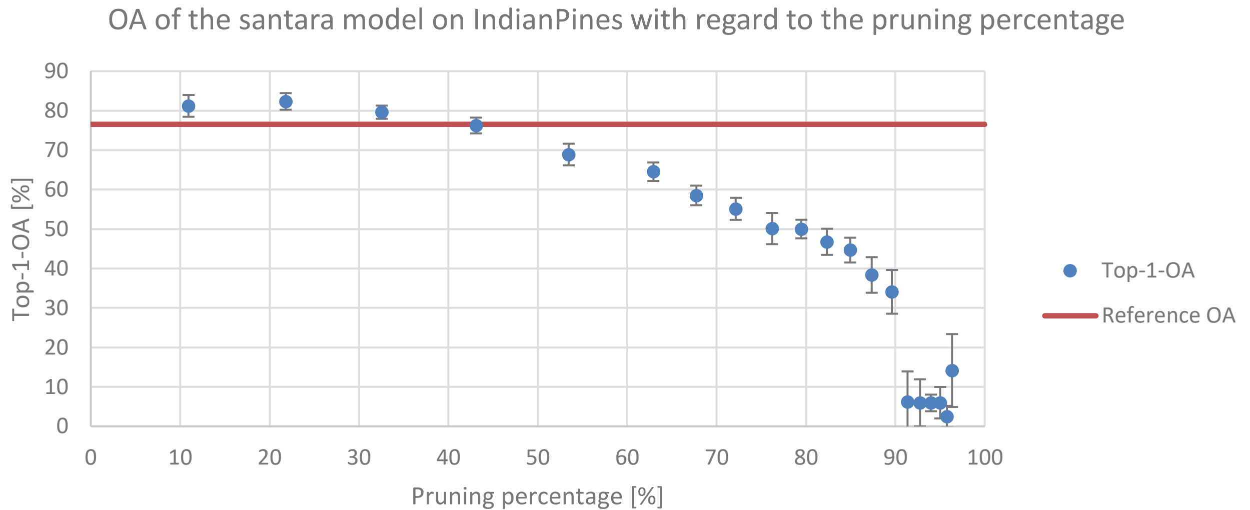


Figure 35: OA for the connection-wise threshold-based pruned santara model (n=6, CI=95%). The increasing steepness of the OA decline after 40% is most reminiscent of the trend outlined by Han et al. from the models we have pruned and analyzed.

The same pattern applied for the **he** model, although a single alpha value does not suffice to get below 18.16% pruning percentage. If we want the model to be pruned less than that, we can set an alpha value for convolutional layers at 0.05 (or even lower) and vary the alpha value for linear layers between 10^{-6} and 10^{-5}, which allows us to get to 15.89% and observe that the accuracy remains excellent, i.e. even better than in the original non-pruned model. This necessity to rely on multiple alpha values depending on layer type stems from the fact that the Conv3d layers incorporate roughly 11,000 parameters whereas the only linear layer as the last layer of the network contains about 270,000 parameters. We can safely prune 62.28%, land at 98.48% OA (higher than the 94.55% reference OA) and save 424.46KB of 1,117KB storage space.

Perhaps the most interesting case is the **luo** model as the largest model on IndianPines we have been able to run due to the presumably large compression potential. Indeed, it is only after 99.15% pruning that the OA gets worse than the reference model OA of 41.41%, which saves us a highly satisfying 396.95 MB - 3.37 MB = 393.58 MB storage space without any loss of accuracy. This high pruning percentage is automatically achieved with the alpha variation from 0.1 to 2.0 in 0.1 steps. Should we be interested in pruning only the giant linear layer, we would define a single alpha value for the second to last layer (the linear layer) and vary it from 10^{-29} to $2.9 * 10^{-29}$, finding out that most of the OAs lie above the reference OA and the ones that do not have a distance of at most one percent to the reference OA.

Though pruned connections are not physically removed from the network, we can calculate the reduced model size assuming a linear function respecting the base model size and taking only the non-pruned part of it. This assumption is justified because after reading in the `pth` files that resemble model checkpoints, we have made sure that apart from lightweight metadata, nothing but the weights and the biases (so-called `state_dict`) are stored in the checkpoints, not even the structure of the model itself because when loading the `state_dict`, the model must have been created in the first place before we can use the appropriate function to load the weights and biases, so the model object is aware of its structure and it need not be loaded from a `pth` file separately.

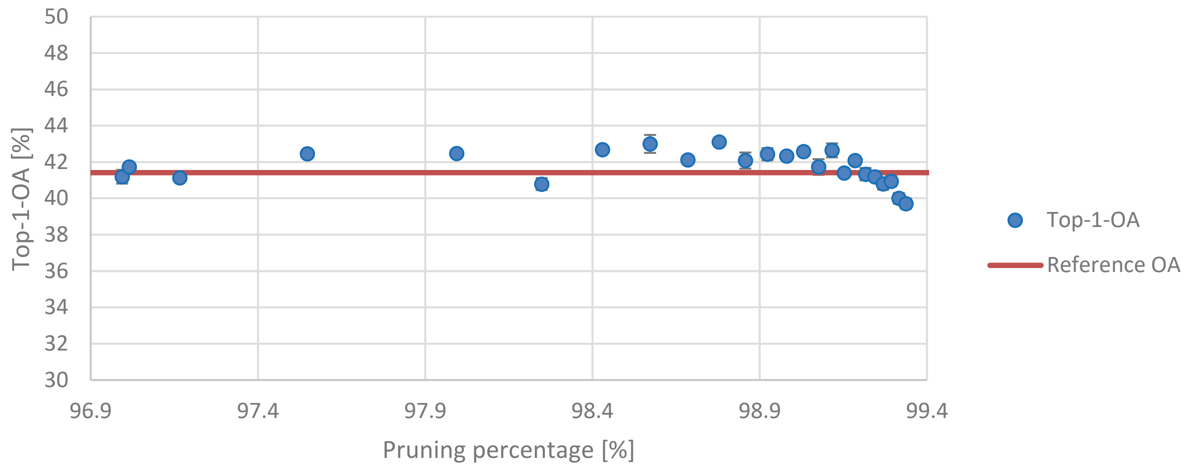


Figure 36: OA for the connection-wise threshold-based pruned luo model (n=6, CI=95%). To highlight how much we could prune before an OA loss worth mentioning occurs, we start plotting the data points at 97% in this figure. Since the number of parameters is primarily caused by the linear layer having more than 2,500 times more parameters than the other two convolutional layers and the other linear layer combined, it is pleasant to see that an overwhelming majority of these parameters can be safely pruned from the network without any negative consequence to OA. This confirms the finding of Han et al. that linear layers can be pruned more than convolutional layers [226]. Between 0% and 97% pruning, most OAs surpassed the reference OA with few exceptions, but deviations in both ways are around 1% from the reference OA.

We did not evaluate the pruning times because they were all insignificantly fast (half a minute at most for pruning one model) and just like the training times, they do not matter because they are onetime costs not relevant in productive use. Since the weights and biases are not physically removed from the network but zeroed instead, it does not make sense to measure inference time, RAM and VRAM usage hoping that these results would get better. Having measured these metrics by reading in the pruned `pth` model files in DeepHyperX for inference, we have the confidence that they indeed do not get better. Physically removing the weights and biases from the network is something we do not deem feasible in PyTorch "because the structure (layers, connectivity and layer sizes) are all described in code" [300], which means we would have to create numerous model classes for each model for each pruning percentage analyzed (as depending on this percentage, a different number of connections is zeroed) and even then, we could not be sure if these model classes could be used for replicability as at least this threshold-based weight pruning is non-deterministic by nature, i.e. we do not know in advance which connections would be pruned. PyTorch does support an export of the model as a trace-based graph in the ONNX format, an open format for interchangeable AI models, which could be imported into Caffe2 (where ONNX is Facebook's "primary means of transferring research models" [314]) or TensorFlow afterwards, but for the measurement of inference time, VRAM and RAM, this does not help us: we would have to copy the whole DeepHyperX infrastructure into Caffe2 or TensorFlow just for IndianPines inference if we decided to use Caffe2/TensorFlow for physical removal of the connections according to the same pruning algorithm we used and inference afterwards (so that the inferences work and are comparable with DeepHyperX's inferences – we think this effort would require significantly more time and resources), and if we used Caffe2/TensorFlow just for physical removal of the connections and wanted to transform the pruned model back into the ONNX graph that we read in in PyTorch, we would arrive at the exact problem of class explosion described above. The onnx-tf converter produces an error message as of now regarding allegedly negative dimensions and the onnx2keras converter did not work either, but we are happily witnessing that work is being done to make deep learning model files more interchangeable, i.e. independent from the framework used [314]. We leave this part as future work for anyone interested in evaluating the model files in the attachment from the viewpoint of inference time, VRAM and RAM, but it is for the reasons described above that we did not decide to proceed with these three metrics. They are irrelevant for the pruning method we chose.

4.2.3.2 Coarse-Grained Pruning

Now it is time to contrast our findings for fine-grained pruning with examples of coarse-grained pruning to see which granularity is preferable, keeping in mind that the implementation effort for connecting Intel Distiller's coarse-grained pruning is greater than using `iterative_pruning.py`'s fine-grained pruning as alluded to in 4.1.3. The fact that Intel Distiller's coarse-grained pruning only considers convolutional layers makes fully convolutional networks very good candidates for our search for model candidates. However, we have chosen the model luo because it belongs to our selection of the five models (so we can compare to reference values), it does have enough convolutional layers and the `forward` function is not too complicated to modify. The models hu and cao, which we have also tried to make sure our findings were not too model-specific, confirm our findings. To point out the challenges in finding the model and what we mean by complications for the `forward` function, i.e. the function in `torch.nn.Module` subclasses (e.g., CNNs, but also RNNs are possible) in PyTorch that describes how tensors are supposed to be passed around from input to output, we will introduce an example.

The model he is not a good candidate because it sums tensors in its `forward` function. Once a tensor gets one dimension trimmed by Intel Distiller's channel-based pruning (e.g., from 16 to 15 dimensions), the summing of in this case three 16-dimensional tensors and one 15-dimensional tensor cannot be done without adding a zero entry at the correct position for the 15-dimensional tensor. This goes beyond the scope of feasibility for this thesis, but fixing this would require Intel Distiller to give feedback to the model's `forward` function, e.g., in form of further parameters, so that the `forward` function knows both which of the four tensors is affected and what the affected position is, while continuing to work flawlessly for no pruning at all. Thankfully, there are working candidates for which we can contrast this coarse-grained pruning with the weights-based, fine-grained pruning provided by `iterative_pruning.py`. The better pruning method according to the criterion of having a high OA while keeping the pruning percentage high will be used in combination with post-training quantization in the end. Surely, one could refine this broadly stated criterion for concrete percentages and model sizes to have a mathematic definition, but we think the cases are clear-cut enough for intuition to suffice.

To make our considerations for choosing the model easy to follow, we will now describe what made us choose luo as the pruning candidate from the five models. he's `forward` function sums tensors, being a bad candidate. santara's idea of the neural network is based on parallel Conv1d layers. Once the four tensors have passed these Conv1d layers, they are merged. Obviously, this idea is bound to fail for channel-based pruning for dimensionality reasons, just like the summation of tensors for the he model was always going to fail once a relevant channel is pruned. The cao model is special in the sense that for the last max pooling before the linear layers, it uses the kernel size of 2, which works flawlessly for torch 1.1.0. For torch 1.2.0, this turns into a problem for whatever reason, which limits portability (we would have to downgrade PyTorch, ideally in a separate Conda environment). We would have to change the kernel size to 1 to make the max pooling layer work, which is a semantic change of the neural network we would rather not like to deal with, so we circumvented it with two Conda environments. The hu model did work, but we did not choose it as our primary candidate because having a Conv1d layer as the only convolutional layer makes it not that exciting for coarse-grained pruning. The luo seems appealing because two out of its four layers are convolutional layers (Conv3d, Conv2d) and it is a large model (letting us hope for a high model size saving potential, although admittedly, the 100 million parameters of the first linear layer are certainly the factor that dominates in the model size calculation, but the weight dimensions of the first linear layer will be pruned as well because they will need to adjust to the tensor's reduced size). If it were not for these arguments, we could have happily chosen a fully convolutional network (lee) that never relies on dimensionality issues, i.e. merging tensors, adding tensors, tensor reshaping never happen in the `forward` function, to explore the full potential of Distillers coarse-grained pruning. Trying it for the luo model, we ran into a problem worth elaborating on, which we can definitely generalize to all models consisting of a mix of convolutional layers first and linear layers after that in PyTorch (i.e. non-FCNs).

Since the size of all layers' inputs must be provided in the source code of the model in PyTorch by design, this also applies to the first linear layer after all convolutional layers, some of whose channels we might have successfully pruned with Distiller. This is something DeepHyperX and we solved in the models given / custom models we included by letting zeros of the correct model input shape pass all layers of the network until the linear layer was reached. Knowing the shape of the zeros, we could infer what the input shape of the linear layer should be. In fact, this is how DeepHyperX determines a first linear layer's input size. To avoid lots of redundant computation, this method we called `_get_final_flattened_size` is invoked only once in the model's `__init__` function. Now that a convolutional layer's number of channels has been reduced, that does impact the tensor shape when it reaches the first linear layer. For instance, when luo's first convolutional layer gets its number of outgoing channels reduced from 90 to 81 (and its successor its input channel number set accordingly), this means that a tensor of shape [1,1,200,3,3] that gets passed through the network will have the size [1,64,18,**79**] instead of [1,64,18,**88**]. It cannot be reshaped to 64*18*88 = 101376 anymore, which would have perfectly tied in with the first dimension of the weights of the first linear layer, which is shaped [101376, 1024], where 1024 is a fixed number we need not worry about, just like its biases, which are shaped [1024], need not be changed. It is necessary to change the input dimension of the layer (`in_features`) to 64*18*79 = 91008. **But the much greater challenge stems from taking only the** [91008, 1024] **parameters for the channels that would still exist**. We have not tried this due to the time frame for this thesis, so we rely on the iterative retraining after channel pruning to arrive at good weights (and biases as well). If one wanted to attempt this challenge, we suggest storing the indices created by Intel Distiller's thinning recipe algorithm and accessing it from the `forward` function of the model to find the correct positions to remove – but even then, one would have to analyze how the tensor gets passed around the model's convolutional layers and how exactly that translates into the positions of the linear layer's weights that have to be removed. What we have done is reinvoke the `_get_final_flattened_size` function when the reshaping of the tensor fails and reinitialize the linear layer with the updated flattened size as the number of input channels, which saves us the aforementioned trouble. We also let a variable that determined the number of outgoing channels for the first convolutional layer be dynamically adjusted according to the `out_channels` member of this layer whenever it changed due to Distiller having removed channels.

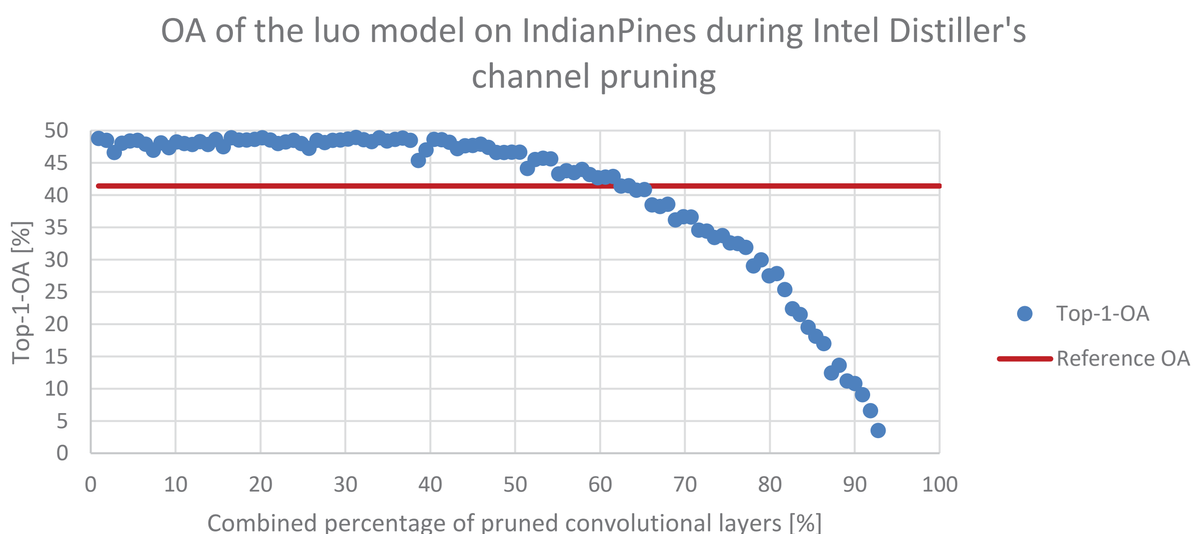


Figure 37: OA for the channel-wise pruned luo model using the l_1-norm (n=6, CIs not shown for the sake of visibility). With wide ranges of reaching 45-50% OA visible, channel pruning is beneficial for a good OA until it begins to drop at 65% combined pruning of the two convolutional layers increasingly rapidly. The structure of the curve is much clearer than in the fine-grained pruning shown in Figure 33, Figure 34, Figure 35 and Figure 36.

In the diagram showing the results, we see that just as in the fine-grained pruning, we could prune up until a certain percentage and the accuracy would not be affected, but rather than the threshold being 99% weights pruning, we need to stop at 65% channel pruning to maintain our reference OA, which means 32 remaining channels out of 90 for the first convolutional layer and 22 of 64 for the second one, resulting in model size savings of 258.02 MB to a new model size of 138.93 MB. Obviously, this is a bigger restriction and makes weight pruning preferable for luo (as well as hu and cao, where this conclusion was the same). Our guess is that the parameters are so dominated by especially the first linear layer that every convolutional channel we prune has all the more influence on the rest of the network that is not affected by channel pruning. But more importantly, it is inherent to coarse-grained pruning that weights are grouped together in channels, so if some of these weights are irrelevant and others important, this needed distinction would not be made by Distiller. In fact, if we look at academic findings, we think that Ma et al. hit the nail on the head regarding the comparison of pruning granularities in the quote from their September 2019 paper dealing with inference speedup on mobile devices through sparsity: "**fine-grained pruning can achieve high sparsity and accuracy**, but is not hardware friendly", while "**coarse-grained** pruning exploits hardware-efficient structures in pruning, but **suffers from accuracy drop when the pruning rate is high**" [315]. The higher hardware-efficiency of coarse-grained pruning is confirmed by Anwar et al., who state that "[i]rregular sparsity is difficult to exploit for efficient computation" [316], just like Li et al. criticize the "irregular sparsity" of networks whose weights have been pruned, proposing filter pruning as an alternative [228]. As an interim conclusion, we can say that we have evaluated two pruning methods whose main difference is the pruning granularity, arriving at the result that fine-grained pruning is preferable for our focus due to both better tool support and both OA and model-size-related results for hyperspectral image classification, so we choose to go with it for our compression pipelines in 4.2.4. Next, we are going to take a look at post-training quantization.

4.2.3.3 Post-Training Quantization

Having had the architectural discussion in 4.1.3, let us move on to presenting our results. We will proceed by highlighting the impacts of post-training quantization on OA with Distiller in three diagrams. In Figure 38, we see the model accuracies after quantizing the entire model with all of its components (i.e. activations, weights, accumulators) with a fixed number of bits, which we varied between 4, 8 and 16 (values that are not powers of two did not make sense as candidates for bit numbers as they would be just rounded up, wasting accuracy potential we could have had for the same amount of space used. 16 bits are our upper bound because our original models are 32-bit representations). All accuracies are abysmal as they do not even come close to the original models and the OA loss renders the models unusable in practice. It is hard to see any structural model pattern at all that explains the OA differences between the models, neither in DeepHyperX, e.g., through the number of dimensionality of the convolutional or linear layers used, nor in TensorBoard, where we do not find a correlation between the number of `add` or `mul` operations and the OAs in this picture. As we shall see later on, quantizing the accumulators definitely explains why the accuracies are bad, but not why they differ between the models. Anyway, now that we have tried it, we can safely say that quantization of all model parts with the same number of bits is not a sensible thing to do for a good accuracy-size-tradeoff. Besides, this eases our disappointment of not having been able to compare inference time, RAM and VRAM usage between quantized and non-quantized models due to the lack of tools available for our purposes: if the models perform that badly when all of their components have been quantized with one fixed number of bits (and other quantization tools, in general, do not support the fine-grained distinction of quantization bits for activations, weights and accumulators), it is not desirable to measure these metrics from a practical standpoint as they will not be productively deployed anywhere anyway, if we assume that people who quantize models check the accuracy first.

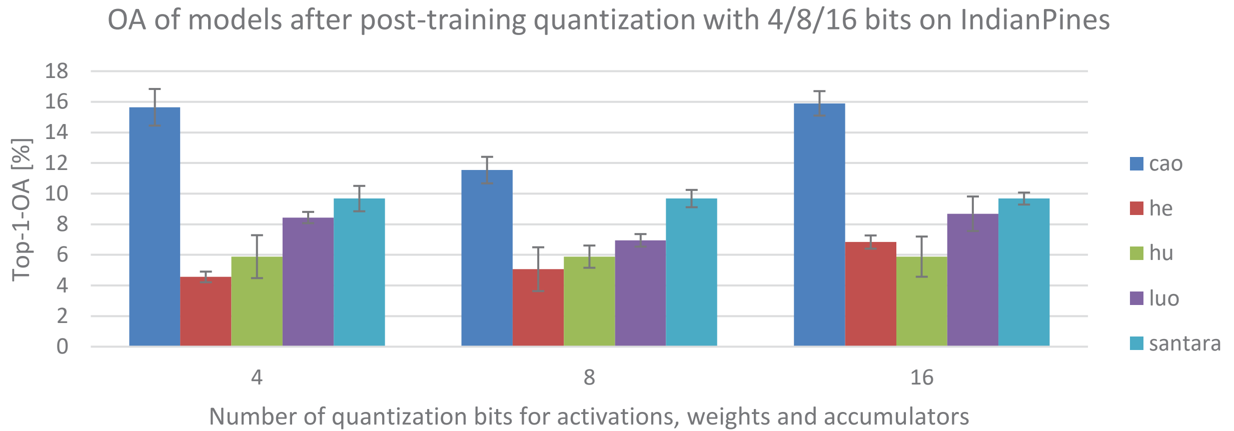


Figure 38: OAs for the cao, he, hu, luo and santara models having been quantized after training, in this case, with the same number of bits for all quantizable model components (n=6, CI=95%). This figure highlights that it is a bad idea to quantize activations, weights and accumulators with the same number of bits as the highest OA we reach in this diagram is 16% (compared to much higher base accuracies as shown in Figure 39 - not shown here to keep these low accuracies visible), rendering all models unusable in practice. Contrasting the models, we see that cao suffers the least in terms of OA, although the differences are marginal.

Now we would like to use the freedom of Intel Distiller to quantize activations, weights and accumulators individually. We received interesting results when leaving activations at 8 and weights at 16 bits while varying the accumulator bit count. As already alluded to, it turns out that any quantization of the accumulators below 16 bits will result in unusably awful OAs, which are around 10% for luo and santara, between 7% and 10% for he as well as hu and between 12% and 18% for cao. **With 32-bit accumulators, the greatest OAs can be reached using 8 bits for activations and 16 bits for weights**, albeit using 16 and 8 bits as well as 8 bits twice are also good combinations OA-wise. This means that the general advice in papers of using 8-bit integers for weights and activations stated by Nervana Systems indeed holds [238]. An advice that the more bits are used for activations and weights, the higher the OA, would be incorrect because things are not as easy as that. For now, the conclusion is that in general, it is not advisable to quantize accumulators and one should quantize activations or weights instead to save space.

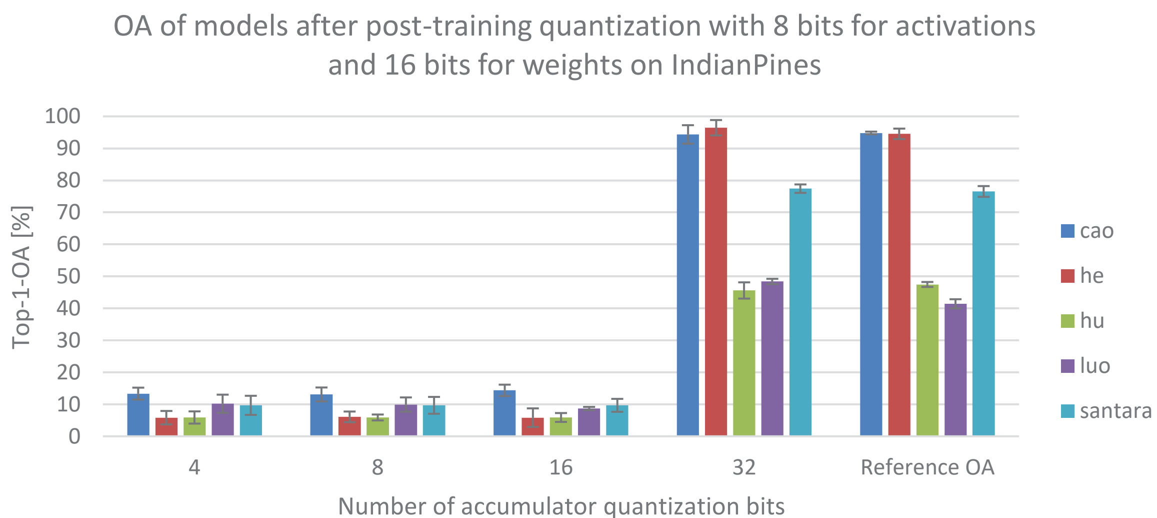


Figure 39: OAs for the quantized cao, he, hu, luo and santara models (n=6, CI=95%). All five models have been quantized with 8 bits for activations and 16 bits for weights. The significance of the variation of bits used to quantize the accumulators is displayed in this figure, which teaches us that it is a bad idea to quantize the accumulators at all (the reference OA models use 32 bits for representing activations, weights and accumulators respectively), considering the enormous OA differences across the board. The combination of 8/16/32 bits for activations/weights/accumulators gets us the best accuracies for the whole quantization measurement, in most cases even surpassing the reference OAs.

In the picture we can even see that some of the OAs of the column left to “Reference OA” surpass their respective reference OAs (while saving space through quantization). This is a rare occurrence because we have observed that nearly all accuracies are worse than their respective reference OA after quantization.

Our focus for the last diagram will be how to reach an acceptably high OA with 16 bits accumulators. Should we absolutely need to quantize the accumulators, **we should decide to use 16 bits and use one of the combinations (4,4), (4,8) or (8,4) for the number of quantization bits for (activations, weights)**, as we see in Figure 40. The accuracies we see for (8,8) are totally representative of the OAs for other combinations at 16-bit accumulator accuracy. Overall, if one did not have the time to experiment with these bit settings to find an acceptable combination for their product, it would definitely require considerable effort to empirically find it as the results vary a lot. We hope that our findings make that clear and show how post-training quantization can be used well for the models we used. For the cases where the OA we reached was not acceptable, it might be a good idea for further experiments to try quantization-aware training (as mentioned, we are focusing on post-training quantization in this thesis): Benoit et al. and Krishnamoorthi et al., who have found OA losses for small models like MobileNet after post-training quantization suggest that this can be a good idea for the RGB case, and given these experiences, this would be an interesting method to evaluate for the hyperspectral scenario [243, 247].

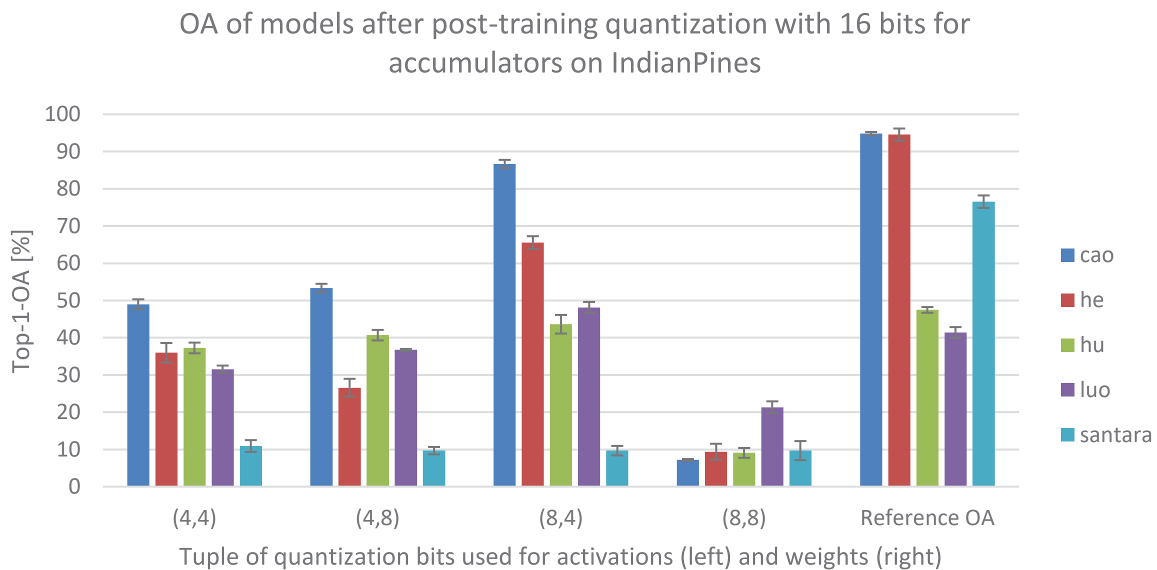
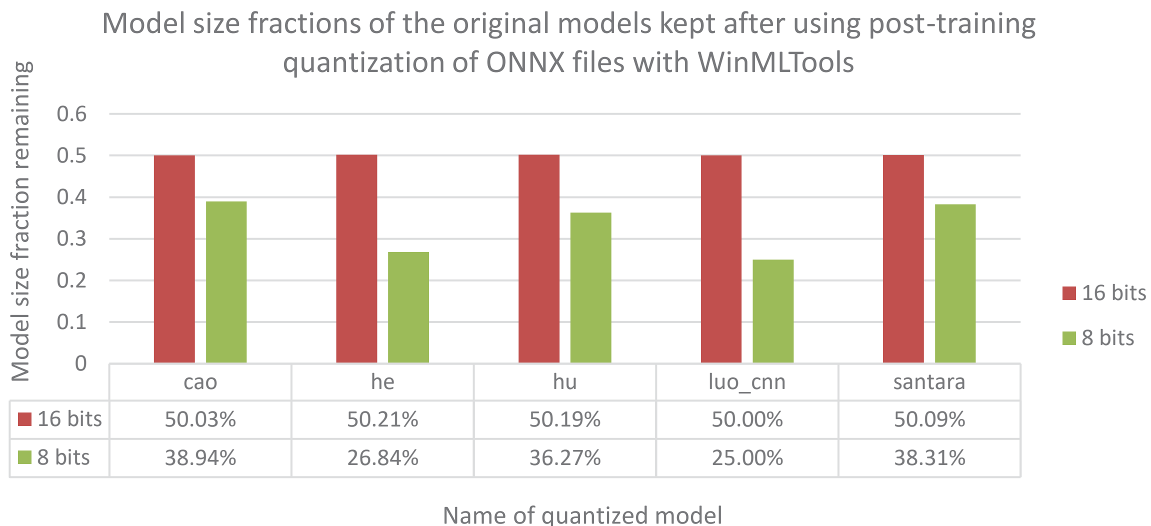


Figure 40: OAs for the quantized cao, he, hu, luo and santara models (n=6, CI=95%). All five models have been quantized with 16 bits for the accumulators. This figure shows that should we decide to quantize the accumulators to 16 bits (e.g., because model size was especially important to us), the only three promising results we will get unless we use (32,32) for (activations, weights) are (4,4), (4,8) and (8,4). All other combinations for all analyzed models always produced accuracies about as low as the ones we see for (8,8), i.e. around 10%. Even for these three combinations, we still lose a lot of OA, so we do not advise to quantize the accumulators.

Zhou S et al. have “observed that activations are more sensitive to quantizations than weights” [238, 255], so it is common practice to just quantize the weights and leave the other components as they are. In fact, the TensorFlow Lite converter does just that in its hybrid quantization by default, and for example, the research of Li et al. and Zhu et al. focuses on just that [317, 318], with their OA-related results confirming the aforementioned sensitivity-related finding. Given our results, we can overall confirm the trend that activations react in a more sensitive way than weights, but we would definitely like to add that accumulators apparently react more sensitively than activations. (But there are exceptions like the (8,8) combination in Figure 40, whose worse OA than (4,4)’s, (4,8)’s, and (8,4)’s cannot be explained by that reasoning at all, we just consider (8,8) an outlier.) On a similar note, our results confirm that the strategy of using more bits for accumulator quantization than for the other components (if we want to quantize the accumulator at all) makes sense: the Intel Distiller documentation reveals that the necessity for more

bits for the accumulator stems from the fact that it is the component that sums up intermediate results, so representing the values usually requires more bits – if we do not do that, we could run into an **overflow in the accumulator numbers**, severely damaging the validity of the calculations, which we suspect is exactly what must have happened in the experiment. To prevent this from happening, Nervana Systems summarizes the common practical advice: if we consider that multiplying two n-bit integers results in a $2n$-bit number at most and that convolutional layers accumulate such multiplications $c * k^2$ times with c input channels and the kernel width k, we need $2n + \log_2(c * k^2)$ bits for an accumulator [238]. Nervana Systems notes that for 4-bit integer quantization of weights and activations "and lower, it might be possible to use less than 32 bits" for an accumulator, which, looking at our observations, especially in Figure 40, we can confirm [238].

Compressing the physical model sizes through the post-training quantization offered by WinMLTools, we got the results we intuitively expected, i.e. that if we reduce the number of model representation bits by half, we get a model of half the model size. For large models as luo, this amounts to a model size of 207MB at 16 bits (415MB original at 32 bits, 103MB at 8 bits). Small models such as he and hu save 573KB / 145KB for 32 to 16 bits, medium-sized ones like cao and santara save 2.28MB / 4.59MB. Instead of including the plot of this simple trend in the absolute model sizes, we decided to show the relative model size reductions as they are more interesting to reason about.



	cao	he	hu	luo_cnn	santara
16 bits	50.03%	50.21%	50.19%	50.00%	50.09%
8 bits	38.94%	26.84%	36.27%	25.00%	38.31%

Figure 41: Relative model sizes remaining after post-training quantization of ONNX model files with WinMLTools. The greater the original model's size, the higher the number of parameters, the greater their impact on the calculated remaining model size fraction. It is a value pushed down by the quantized parameters for larger models, which becomes more apparent for 8 bits than it is visible for 16 bits.

The relative model size reductions depicted in Figure 41 show us that for 16-bit quantization, the relative savings lie around 50%, with the smaller models achieving lower reductions compared to their original models. That is because the small overhead due to metadata and layer information is simply not reflected in the fractions we see in the table anymore when, e.g., luo has a massive fully connected layer with more than 100 million quantized parameters. In that case, the overwhelming number of parameters pushes the remaining model size fraction down so much that it seems as if we did not have any metadata saved in the ONNX files at all. Consequently, larger models (which means more parameters) have a lower model size fraction remaining after 16-bit quantization. Whereas the percentages do not vary much with just 0.21% maximum difference between the smallest and the largest model at 16-bit quantization, this difference becomes more apparent with the 13.94% difference at 8 bits, but the same metadata vs. number of parameters argument holds. **The metadata (including codebook, graph structure etc.) just carries more weight**, as can be seen when loading the quantized `onnx` files with the `onnx.load` function.

As for the inference times of the quantized `pb` graphs, i.e. the `tflite` models, we did not experience any speedup on our desktop computer. That was to be expected because the latency speedup is to be expected on appropriate hardware, whereas in our case, an 16-bit floating point quantization lead to an increase of the inference time by 4-7% for the four models compared to the reference values and the inference times after 8-bit integer quantization lied between 5% and 10% above the reference. Consequently, we implemented a rudimentary Android app with Android Studio that uses TensorFlow Lite for inference, but the inference ran into problems others have also experienced and which there is no fix for at the moment, mainly that reshaping the tensor for our needs does not work for the quantized model [319] and that memory allocation fails [320]. While we were at it, we experienced first-hand how massively we would need to restrict the test set in size for processing purposes because everything else would require so many resources that even in our virtual Android device, more than 10GB of RAM were needed just to process the dataset we saved as a JSON file and the runtime was absolutely awful, i.e. multiple minutes just for processing 1% of the IndianPines dataset – for comparison, the usual train-test-split is 80%-20% [43]. This highlights how important band selection methods are to compress a hyperspectral dataset, because IndianPines' 200 bands just contribute too much to storage and memory requirements for an app centered around hyperspectral datasets to be usable in practice on a mobile phone in any way. For RGB datasets, Krishnamoorthi et al. experienced a 2-3x inference time speedup for the quantized neural network on appropriate hardware [247] and we hope that similar research for hyperspectral datasets will follow.

To be able to explain the accuracies for running the `tflite` models on our desktop computer, which we are going to mention in a moment, we will quickly analyze how TensorFlow's quantizations relate to Intel Distiller's quantization options. We used TensorFlow's post-training float16 quantization and post-training integer quantization as our two quantizations because these are exactly the modes WinMLTools also supported. They convert the weights to float16 / int8 respectively. As TensorFlow points out in describing the integer quantization, "all weights and activations are quantized statically during model conversion" to 8-bit ints [321]. For weight quantization, "activations are always stored in floating point" and are quantized to 8 bits of precision when the operation is supported [307]. In both cases, the 32-bit floating point accumulators are approximated by the formula "`real_value = (int8_value - zero_point) * scale`" [307]. Essentially, these two modes can be compared to 8/16/32 (post-training quantization) and 8/8/32 for the number of bits for quantizing activations/weights/accumulators. Both of these modes performed among the best in Intel Distiller – 8/16/32 was the best combination accuracy-wise and 8/8/32 performs just as well but for about 1% OA for all four models. This explains why apart from slight deviations of around 1% for each model, the OAs we obtained for the `tflite` models are the same we got for the Distiller modes 8/16/32 and 8/8/32 respectively, which is a very satisfying finding (as shown in Figure 39, the OAs lie around 93/95/48/78 percent for cao/he/luo/santara). If we contrast this with research, FP16 for weights and activations "works with little effort" and 8 bits for convolutional layers (and 4 bits for fully connected layers, if we wanted to vary the bits by layer, which we did not) can be used with care according to Dally [239]. For anyone curious to replicate the experiment to find out the inference speeds, we recommend waiting for more quantized operators to be supported, especially Conv3d (very important layer for many of our models – might be not nearly as important for non-hyperspectral image classification tasks), for the TOCO quantization tool (that converts `pb` to `tflite`) to allow the reshaping nodes we need in the graph and for the interoperability of PyTorch and TensorFlow through ONNX to be developed to a degree that the choice of the framework does not matter anymore for hyperspectral image classification because conversion is fluent and integrated enough for the user to not notice it. This is not the case right now, so this is all we could do. Let us now present whether or not it makes sense to use a compression pipeline consisting of several compressions (alluding to Han et al.'s compression pipeline [12]).

4.2.4 Combinations of Compressions

Now that we have analyzed the impact of the band selection techniques presented, the weight-based pruning provided by `iterative_pruning.py` and the post-training quantization with varying bit numbers made possible by Intel Distiller, we have combined these three procedures in a compression pipeline. This means that we first compressed the hyperspectral dataset with dimensionality reduction methods (in particular, PCA, NMF, UMAP, LogisticRegression and LinearRegression), then we varied the alpha values just like we did in 4.2.3.1 (including the same variation of alpha values by layer type) to achieve different model pruning percentages and analyze how well the pruned model would work, and in the end, we applied the bit variations for activations/weights/accumulators from 4.2.3.3 for post-training quantization. This hyperspectral image-model-compression pipeline of ours lead to very promising results OA-wise overall, as we are about to explain in more detail. Due to the lack of time and resources for this extremely time-consuming task (due to the countless combinations of model, band selection technique, number of components, alpha parameters for pruning as well as quantization bits), we could only afford to conduct all of the following experiments once, but all results can be reproduced with the model files attached if desired.

Since varying the number of components for band selection would lead to too many combinations, we decided to restrict the number to five sampling points, for each of which we have observed meaningful value differences in 4.2.2. These are [40,70,100,140,170] for all feature extraction techniques, [10,30,50,70,90] for LogisticRegression and [20,30,40,50,60] for LinearRegression. Now we are going to evaluate two compression pipelines, i.e. “band selection” → “fine-grained pruning” (4.2.4.1) and “band selection” → “fine-grained pruning” → “post-training quantization” (4.2.4.2).

4.2.4.1 Band Selection and Fine-Grained Pruning

In 4.2.3.1, we have already stated that moderate pruning can be recommended as no OA loss occurs while the number of zeroed weights increases, which translates to a reduced physical model size using an appropriate deep learning framework with hyperspectral models and datasets included (e.g., Keras, TensorFlow, but in contrast to PyTorch, a framework similar to DeepHyperX has yet to be developed in these languages). The question is if this behavior is retained when we apply band selection techniques prior to this fine-grained pruning. From our experiments, we can tell that applying the two techniques we considered the best (good OA, predictable behavior, short runtime, low memory consumption) in 4.2.3.1, i.e. PCA and NMF, **helps to prolong the pruning percentage point** at which the good OA starts to drop more and more (at least for the he and hu models). When this point remains the same, chances are that the OA has been drastically improved compared to the reference OA (see luo, santara models; the cao model is an exception; more on that later), **much more so than PCA/NMF or the pruning can do on their own**. If we look at Figure 42, we see that the reference OA of 94.55% is crossed at 64% pruning percentage for no prior band selection, while this is the case for 80/83/85/85/90% for PCA with 40/70/100/140/170 components respectively prior to pruning, which we will abbreviate with PCA-40 and equivalents. It is indeed the case that in all examples, we see a point at which a sharp OA decline starts (just like for pruning without band selection), but knowing this, one could choose the pruning percentage accordingly to the tradeoff between pruned weights and OA he desires. For cases like PCA-170 with 86% OA at 93% pruning, this decline is so late in the graph that we think the neural network designer really does not need to spend much time thinking about a good pruning percentage as long as he does not carry the pruning percentage to an unintuitive excess because judging from the practical perspective, we think it is unlikely that a person with said responsibility and appropriate experience would expect a good OA at, e.g., 99.92% pruning from gut feeling. Figure 42 presents the basis for this intuition. Of course, this compression pipeline also has an impact on the potential model size reduction we could perform: under the condition that we are satisfied with keeping the respective reference OA, i.e. the OA for no pruning and no band selection, and consider the average of the five OAs for PCA-X with $X \in \{40,70,100,140,170\}$ as the new OA we compare against, we can reduce the model sizes of he/hu/luo/santara by at least 82/95/90/85% (in absolute values, by 937.26KB/294.45KB/357.25MB/8.87MB) respectively.

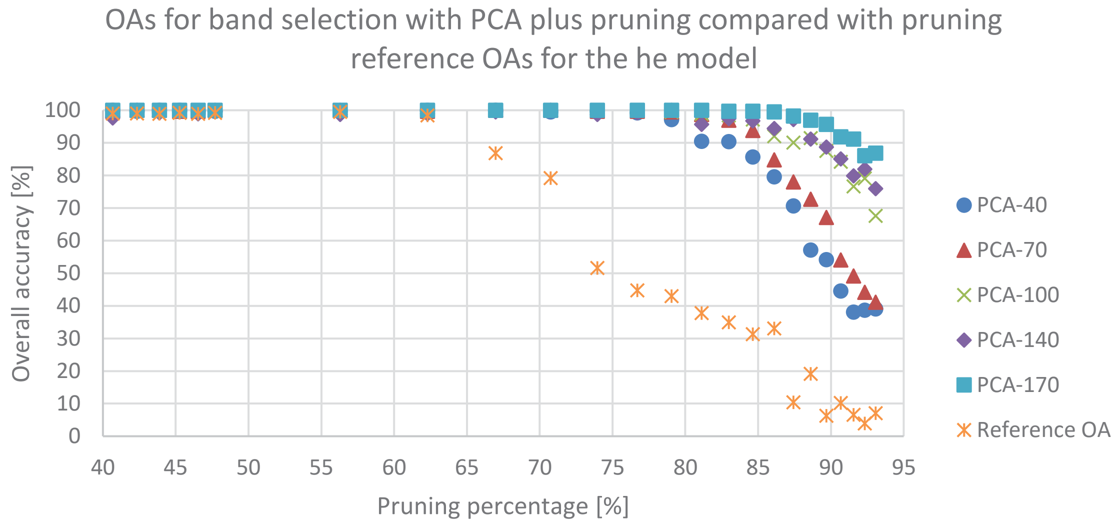


Figure 42: OAs of the models by their fine-grained pruning percentages for five component numbers. PCA was used for feature extraction as a result of the conclusions of 4.2.2. Compared to the reference OA (no band selection technique, number of bands stays at the 200 for the IndianPines dataset), high accuracies are retained longer and the trend is visible much more clearly. Until around 60%, all accuracies of the PCA-X curves for $X \in \{40,70,100,140,170\}$ are very close to or precisely 100%. After that, using more PCA components leads to a shallower accuracy descent.

What we also see in Figure 42 is that the precise number of components for PCA does not matter that much (OA is roughly the same until 70% pruning; up to 5% differences for 30 component difference until 85% pruning; only after that will the OA differences account for 20% ranges), what is important is to choose a band selection technique like PCA at all because the OA differences between the component numbers for PCA are not nearly as large as the difference between 200 components without using PCA at all and using PCA for almost whatever number of components: judging from our experience in the experiment, greater than 5 would be a reasonable number – this is in line with [161] – but if one wants to take risks, even 2 PCA components have sufficed in our experiments (cf. caption of Figure 30). On the other hand, if we want to point out the differences at high pruning percentages, **more PCA components lead not only to a higher pruning percentage at which the descent starts, but the descent is also less steep**. What is more, Figure 42 is well-suited to point out that in our PCA plus pruning pipeline, it is frequently the case that all PCA-X curves with $X \in \{40,70,100,140,170\}$ reach **accuracies of 100% for long stretches of pruning percentages**. We have not shown the accuracies before 40% pruning so we could see the OA development in the end, but the overwhelming majority of accuracies for all PCA-X curves for the he model with less than 40% pruning are exactly 100%, with few 99.74% “outliers” (for the reference OA, it is more like 99% instead of 100%). In our opinion, these are amazing results, and if we look at the confusion matrix, we see that the accuracy improvements between not having and having applied fine-grained pruning on top of PCA are due to the fact that samples classified as “unclassified” after PCA are classified as the correct class after pruning. This leads to the accuracy improvement from the high 90s to 100%, as apart from this “unclassified” phenomenon, there is nearly no false classification made with the 16 classes other than “unclassified”, but even these misclassifications also disappear with added pruning in moderate ranges (i.e. until approximately 60%).

We know that both PCA/NMF and pruning can cause the accuracy to improve. But the combination of image compression and model pruning can improve them more than the techniques individually. For example, he’s reference OA of 94% went to 99.8% OA for 70% pruning, hu from 47% to 77% for 63% pruning and santara from 76% OA to 98% for 72% pruning. These OA values are averages for PCA-X with $X \in \{40,70,100,140,170\}$ components. We justify this averaging because the OAs do not differ that much depending on the number of components used for these models. Only for the luo model (our biggest model with the huge linear layer) is this not the case: while the pruning percentage does not impact the OA unless it becomes very large (cf. Figure 36), the number of components does all the more, with an

average OA of 90% for 70 components, 93% for 140 components and just 24% for 170 components. **This is reminiscent of the dent** we experienced at 110 components for the luo model when applying PCA for band selection purposes – judging from the data, where only 4-5% OA variations occurred for the same number of components among all pruning percentages tried, the number of components, i.e. the dent, clearly outweighs the pruning percentage in terms of importance. For the four other models, more components mean a higher OA (improvements are 1-9% OA for adding 30 components before the point is reached when the OA starts to drop). One might assume it is the opposite for luo (OA for PCA-170 is closer to the 41% reference OA than the 90% OA for PCA-70), but this is not the case as we have seen from our further experiments for 150 (94% OA), 160 (89% OA), 180 (23% OA) and 190 (87% OA) components. The way the OA jumps around is unpredictable in this case. Regardless, especially regarding model size considerations, it might be worthwhile to use fewer bands: the OA drop is typically not large for fewer components for moderate pruning percentages (see above), but the model generated for this reduced number of bands is smaller. To help strike this tricky balance, Figure 43 shows the relative fractions of the model sizes kept for the five models, where the respective reference model size is for 200 components (i.e. no band selection). The model sizes generated do not depend on the band selection method, but solely on the number of components used for the band selection method, regardless if it is PCA, NMF, LinearRegression etc.

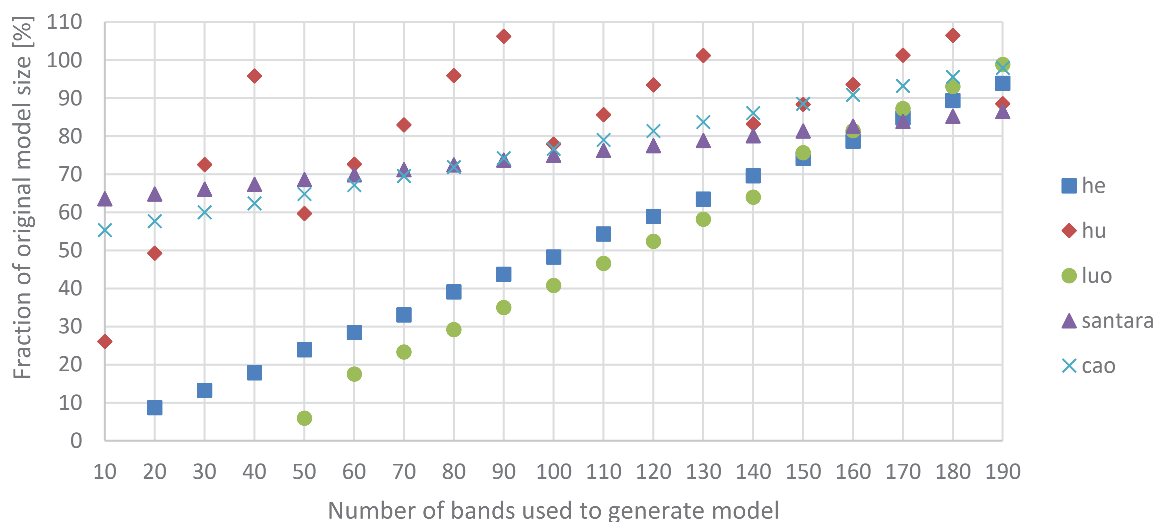


Figure 43: Relative model size savings with regard to the number of components used for the band selection technique, with which the model was generated. hu's model size is expressed in four rays getting increasingly shallower (due to ceil functions determining the two output dimensions of Conv1d and MaxPool1d), sometimes surpassing its reference size, while the other models' sizes scale linearly with the number of components. The absolute reference model sizes are 1.14MB (he), 309.94KB (hu), 396.95MB (luo), 10.44MB (santara) and 4.56MB (cao). The luo model cannot be generated for 40 (and fewer) components because the kernel size of its second convolutional layer would have been greater than the actual input size, which is not allowed. The same reasoning holds true for he and 10 components.

As for the impact of the band selection techniques, we also should not generalize about PCA improving all OAs because the highest OA of the cao model (the CNN relying on Markov random fields) is 81% for 23% pruning, while the reference OA lies at 95%. This is surprising in the sense that PCA alone did improve the OA, but PCA plus pruning did not (even if we tune the pruning to be as slight as possible, there are no signs for the 79% OA at 8% pruning to come closer to the 95% reference OA at all). The same observation can be stated for NMF on the hu and luo models prior to pruning. In most cases though, i.e. all other tested combinations of PCA / NMF and models, we can recommend the usage of a well-performing band selection method on a hyperspectral dataset before compressing the model with fine-grained pruning, but ideally, a neural network designer needs to find out first if the OA for his model is improved by this

procedure. The results have shown that the coherences described are often true as rules of thumb to determine if the circumstances (specifically, the model, the band selection method, the number of components used and the amount of pruning) allow the OA to improve or if it is worsened by band selection plus pruning, but there is no guarantee that these observations always hold.

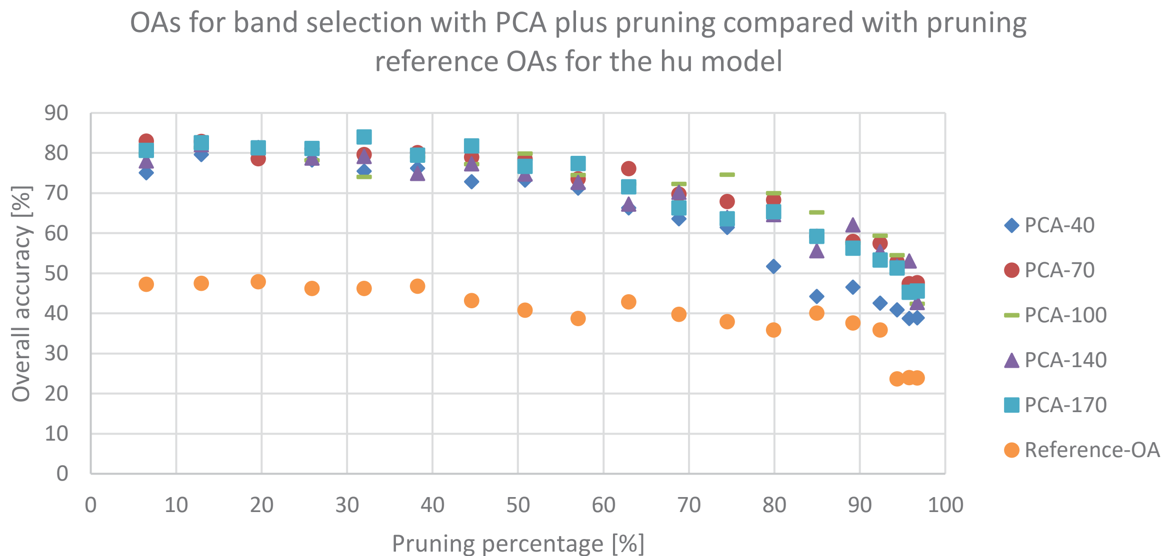


Figure 44. OAs of the models depending on the fine-grained pruning percentages for the five component numbers with PCA as the band selection technique. ***hu's reference OA is twice as worse as he's, which shows the potential of raising the OA by this pipeline more clearly****: until around 60% pruning, the mean OA of all PCA-X curves for $X \in \{40,70,100,140,170\}$ is almost double the reference OA. Only then do the OAs begin to decline and the difference between the Reference-OA and the PCA-X curves begins to shrink.*

In addition, just like we generally observed that if a band selection technique like PCA or NMF performs well on its own, it is going to perform even better in the combination with pruning, if a band selection technique performs badly like UMAP, LinearRegression, LogisticRegression, RandomForest and Autoencoder, it is going to continue to perform badly in the combination with pruning overall. The only exception, again, is the luo model with LogisticRegression, where the OAs for 70 components lie around and at times even surpass the reference OA of 41% by 1-2%, whereas a lower accuracy of around 37% can be observed for luo for 50 components, but this proximity to a linear function depending on the number of bands selected for feature selection techniques has been mentioned in 4.2.2.2 already.

If we direct our attention at Figure 45 and observe the curves per model as opposed to the big picture, we witness different curve patterns. luo's OA remains rather constant – comparable with Figure 36, but notice how Figure 36 only concerns high pruning percentages because the pattern for all previous pruning percentages looked precisely as constant as in Figure 45 – it is a large model after all, so our interpretation is that moderate pruning does not affect it nearly as much as the other models. hu's curve is different in the sense that we see the OA decrease at around 80%, reminding us of the trends in Figure 42 and Figure 44. The shape of cao's curve we think can be best described by remaining constant, but with large accuracy variations, which contribute to making the diagram look messy (but excluding the model from the diagram we think would be an even worse idea). For he and santara, one could identify the three pruning phases of rather constant OAs (although extremely varying around their respective constant OA value) until 70% and a rather linear decrease until 80%, followed by very low OA afterwards.

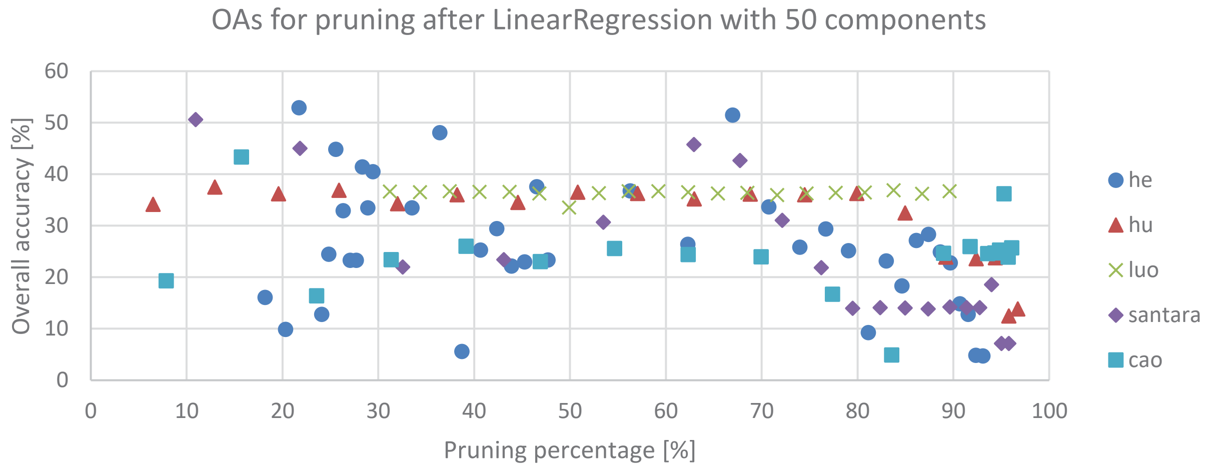


Figure 45: OAs of the models for LinearRegression with 50 components followed by fine-grained pruning (best viewed in color). The accuracies of he in particular are all over the place, they do not surpass the reference OA at any time. While the luo OAs remain constant and we observe the same curve type for hu as in Figure 44 with PCA, the other accuracy curves only vaguely follow a recognizable pattern. This unpredictability / chaos combined with a low OA is characteristic for the feature selection algorithms analyzed, of which LinearRegression represents the trend we encountered well.

If we had decided to plot Figure 44 for LinearRegression instead of PCA, we could also see staggered lines, so there would not be added value by including such a visualization. Overall, we were not able to find other work that explores the combination of hyperspectral CNN classification with both band selection and model compression. To us it seems like the combination of these three aspects is a very novel topic indeed: obviously, compression pipelines like Han et al.'s exist, but they are focused on targeting model compression [12]. Clearly, hyperspectral band selection is not an unexplored topic (cf. 3.3.1), nor is model compression on RGB datasets a new endeavor (cf. 3.3.2, 3.3.3 and 9.1), but we are exploring the combination of both. The closest topics we could find are Cai et al.'s end-to-end trainable hyperspectral band selection framework, where two variants of band selection networks are constructed as alternatives to other band selection methods (one is a fully connected network, the other a CNN) [322], as well as Tian et al.'s neuron-level pruning framework for both convolutional and fully connected layers using LDA [323]. What we can say is that we are impressed by the 100% OA on the he model for up to 60% pruning percentage for PCA followed by pruning and similarly extreme OA improvements for santara and, with limitations, for luo and hu, but the flipside is that the OAs can disappoint as well (cf. cao). While we do not know aspects like the computational capabilities of the remote sensing satellite which might be used for preprocessing purposes according to this compression pipeline to judge if the tradeoff we elaborated on in this thesis is in general worth it, we can only say that massive OA improvements are possible for certain models and suggest that the researcher conducts an evaluation according to his model, objectives and other factors like computational/bandwidth/storage size limitations beforehand to see if he is pleased with the tradeoff he would achieve using this pipeline, and act accordingly. Rather than just assuming a cause out of mid-air for the accumulation of improvements brought by band selection and fine-grained pruning together, we are looking forward to further research that analyzes the reasons on a more low-level basis, as opposed to our high-level approach to evaluate the suitability of many different band selection techniques, model compressions, models, and combinations of these.

4.2.4.2 Band Selection, Fine-Grained Pruning and Post-Training Quantization

In 4.2.3.3 we praised the bit triples (8,16,32) as the best OA option, (8,8,32) for a similar OA with more model size savings and the budget alternative (8,4,16) for post-training quantization of (activations, weights, accumulators) with the said number of bits. **This remains true if we include band selection and pruning** (and as special cases, just band selection if we prune 0% of the model, or just pruning if we leave the 200 components as they are, not performing any transformation of image bands) prior to the post-training quantization. The impacts of the number of components and the pruning percentage described in the previous section are the same with or without quantization.

To illustrate whether or not adding post-training quantization to our compression pipeline consisting of band selection and pruning is worth it, we will show side-by-side OA comparisons with and without quantization. One needs to decide for himself if the model size savings, which are heavily dependent on the number of activations, weights and accumulators included in the model, make up for the accuracy loss observed and are thus worth the effort of using the compression pipeline for the model. The concrete fractions of model sizes kept for the post-training quantization in the pipeline do not differ from what we have calculated in Figure 41 already for post-training quantization outside any compression pipeline.

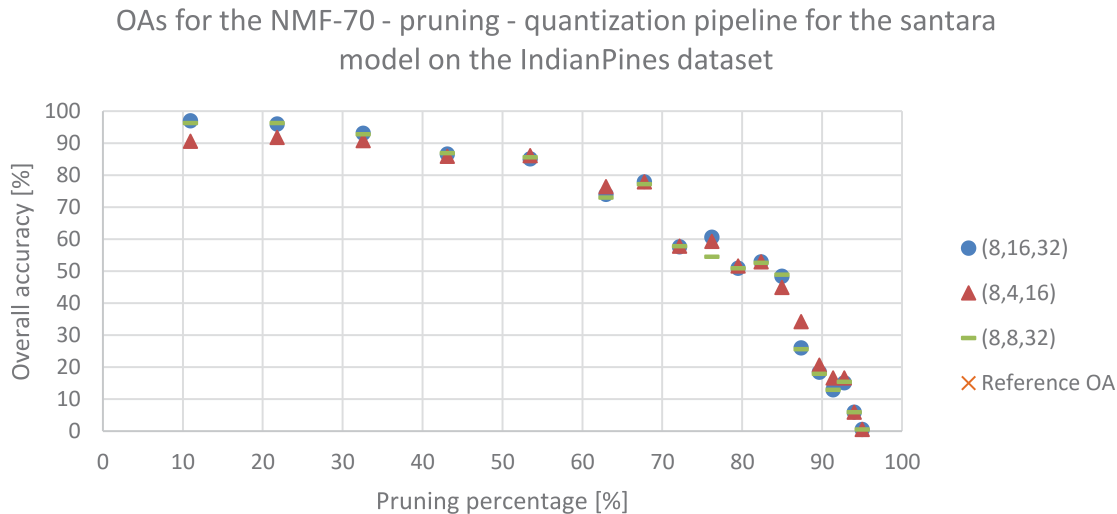


Figure 46: OAs for the NMF-70 - (fine-grained l_1-based) pruning - (post-training) quantization pipeline for three (activations, weights, accumulators) bit combinations compared to the NMF-70-pruning reference OA curve. The values are often so close to each other that no matter which line style we chose, we would not be able to see all four values. That means that not much accuracy is lost here due to the added quantization, and the amounts of OA lost for the three curves mostly correspond to our findings from 4.2.3.3. The results for other component numbers in $X \in \{40,70,100,140,170\}$ for NMF-X reflect the same findings, as does PCA-X for that matter.

In Figure 46 we see how the reference OA (NMF-70 plus pruning) compares with the OA curves for quantization on top with the named number of bits for the three components. Just like in 4.2.3.3 we only rarely saw OA improvements for (8,16,32) quantization compared to the reference OA and slight (8,4,16) to negligible (8,8,32) OA losses, the same generally applies in this pipeline for all band selection methods (good and bad) for all models, all numbers of components and all alpha values. There really is no deviation from the findings of 4.2.3.3 we could present here for these three cases, and the same reasoning applies (e.g., the recommended number of bits for the accumulator, that weights are less sensitive to quantization than accumulators and activations and so on). However, in addition to these three bit combinations we know to perform well, we also tried quantizing all three component types with 16, 8, 4, 2 and 1 bit(s). In general, just like we have seen before, all of these combinations perform awfully. For instance, for the NMF technique on the santara model and components in {40,70,100,140,170}, the accuracies for 16-/8-/4-/2-/1-bit quantization all lie at 0.496% and do not change at all depending on the alpha value (which determines the pruning percentage). Similarly awful OAs can be encountered for numerous other combinations of band selection technique, model and 16-/8-/4-/2-/1-bit quantization, so this is not an option. We can only reinforce the finding we have presented already that if we want to get good OAs with few bits, these can be reached with (8,16,32), (8,8,32) and (8,4,16) combinations for quantization. Future work could find out the reason why that is by performing low-level analysis of the calculations performed on the nodes of the model, which we can see in TensorBoard for our `pb` files (an example is Figure 64), but this is beyond the scope of this thesis. Alternatively, we can refer to [238] for existing recommendations, as we did in 4.2.3.3.

4.3 Visualizations

If we group the neural networks he and luo together as 3D CNNs, santara and cao as 2D CNNs, consider hu as a 1D CNN and average the accuracies, we absolutely do not see a trend that any dimensionality of CNNs is more appropriate to use than another. Architecture-based evaluations are too model-specific for us to be able to group CNNs that have been designed with different ideas in mind together just like that. However, having evaluated neural network and image compression methods, we can visualize a model to prove that the model compressions work as intended, i.e. that they have pruned irrelevant parts of the model according to their criteria, which means that important neurons continue to work as they have before. As said in 2.2.3, Hohman et al. list and categorize works on visualization from the theoretical perspective in their survey [108]. As for the implementation, numerous visualization techniques exist, e.g., for PyTorch [324] and TensorFlow [325], but as expected, the fact we are dealing with hyperspectral datasets severely limits our visualization options. Given that our five models take number-of-bands-dimensional patches of the image as their input tensor for pixel-wise classification (because we need to classify every pixel based on itself and the patch surrounding it) as opposed to an entire image, it should be clear that we should not expect to see a picture the human eye could easily recognize. Rather than that, it suffices to show that the visualizations do not change much before pruning (reference OA) and until the amount of pruning that crosses the reference OA. To briefly recapitulate section 2.2.3 on what the terminology means to better understand what is depicted in the forthcoming visualizations,

- **gradient-based saliency maps** show the gradients of the size of the input patch for the trainable weights and image bands considered, which are model parameters for a chosen layer (cf. Figure 47, Figure 48, Figure 59, Figure 60);
- **activation maps** show the input that would maximize the output of (in our case, all) filter indices at the chosen layer index for the chosen image bands (cf. Figure 49);
- **guided-backpropagation-based saliency maps** show the gradients calculated with guided backpropagation, which is backpropagation where negative gradients do not flow backwards (cf. Figure 50).

4.3.1 Gradient-Based Saliency Maps

As it turns out, the gradient-based and integrated gradient-based saliency maps are always identical, so we are going to group these saliency maps together. We have chosen the pruning percentages 40%, 55%, 70% and 85%. Looking at Figure 33, these are the sensible percentages to look at because OA starts to almost linearly decline from 40% all the way up to 85%. As already stated, the saliency maps for no band selection are all black squares, but the visualizations for NMF and PCA are exciting because one sees the development of the gradient for increasing pruning percentages. Below are examples for a randomly chosen trainable weight, i.e. the biases of the `dense_1` layer (first dense layer of the cao model). Brighter pixels denote a higher gradient.

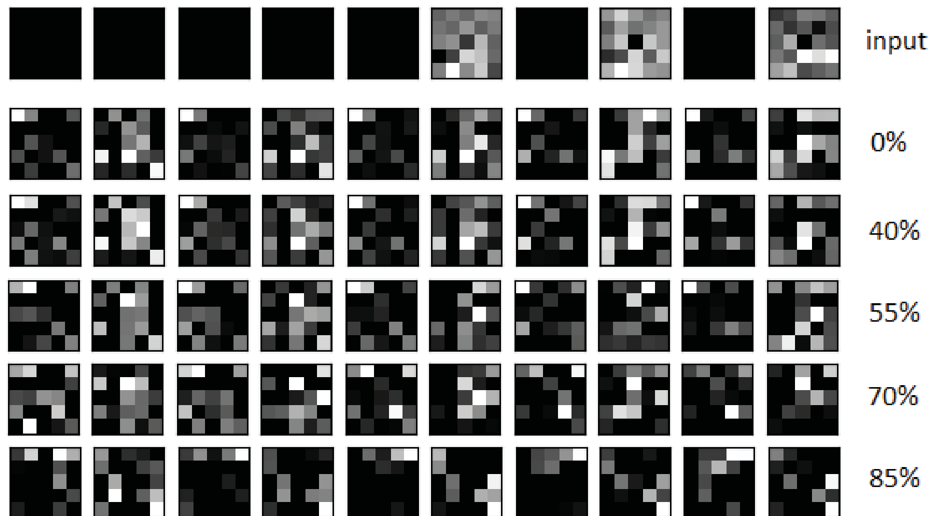


Figure 47: Gradient-based saliency maps for 0 to 85% APoZ-based channel pruning for bands 0-90 in steps of 10 for PCA on the biases of the `dense_1` layer. The features at 0% are still very visible at 40%, which makes sense because until 40%, the OA can be maintained. The more we prune afterwards, the more distortion we start to witness: the position and value of the gradients progressively shifts. At 85%, the gradients have become particularly low.

The application of NMF on the dataset is not as instantaneous as PCA, but compared to PCA, the NMF results vary less with increasing pruning percentage. The same trainable weights as above are chosen:

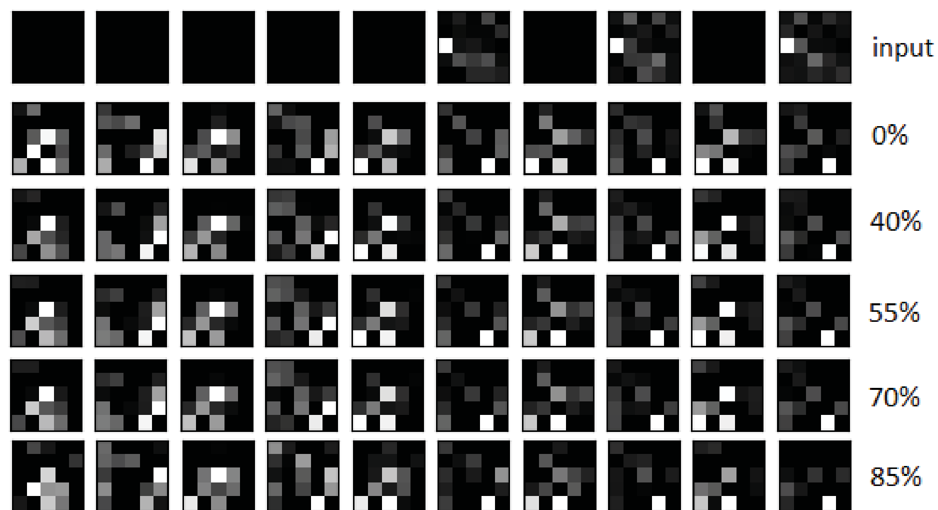


Figure 48: Gradient-based saliency maps for 0 to 85% APoZ-based channel pruning for bands 0-90 in steps of 10 for NMF on the biases of the `dense_1` layer. In contrast to Figure 47, features remain even more visible at higher pruning percentages, i.e. the pixel maps do not change as much as in Figure 47 for more pruning – likely attributed to NMF's non-negativity constraint, as we think. Gradients at individual pixels get darker for a higher pruning percentage, often brightening up the surrounding area (e.g., bands 0 and 20 for 70 vs. 85% pruning). The input appears less bright than PCA's.

In summary, we are relieved to see that the visualizations overall vary just like the accuracy curves we saw earlier, so **important neurons are unimpaired by our pruning**, while the variations at higher pruning percentages might be attributed to the lack of neurons of average importance (because only the most important channels remain), but this latter suggestion is just an educated guess. What matters is that the decisive neurons are not pruned away, i.e. that the way the channel pruning we analyzed works actually does work. A related paper whose, as of November 2019, anonymous authors decided to visualize masked weights of LeNet's second convolutional layer with regard to the pruning iteration and the precise pruning method used is called "On Iterative Neural Network Pruning, Reinitialization and the Similarity of Masks". We can think of the pruning iteration as a metric equivalent enough to the pruning percentage because a higher iteration means additional pruning, assuming the additional pruning is distributed

roughly in a linear way (even if it is not, this is not crucial). Interestingly, the authors' results for both l_1-norm-based structured and unstructured pruning with rewinding resemble our visualizations in the sense that some components in the gradient-based saliency maps change, but main shapes remain the same (this is exacerbated by the fact that the iterations the authors compared were the 2nd/5th and 5th/10th for structured / unstructured l_1-based pruning with rewinding respectively) [326]. An explanation attempt of ours is that whether we decide to use l_1 or APoZ as the pruning criterion, it is just a criterion after all that defines the relevance of weights/channels and chances are that at least in earlier pruning iterations, unimportant weights/channels will not differ as much, but this theory definitely yet needs to be proven or refuted. As the authors point out, rewinding network weights is an alternative to finetuning the pruned model, and unstructured pruning "removes individual connections, while structured pruning removes entire units or channels" [326].

The visualization reassures us that the conclusions drawn when we analyzed coarse-grained pruning are founded. What is more, Figure 59 and Figure 60 show us that **it does not matter which layer we choose, the observations hold**. Besides, we could not see that a layer in the beginning of the CNN would behave differently than one closer to its output layer (e.g., it is not the case that more pruning is more distorting for the gradient-based saliency maps of latter layers as opposed for former ones). Let us now demonstrate whether the alternative visualizations we tried give us any meaningful insight.

4.3.2 Activation Maps and Guided Backpropagation

The activation map remains completely unchanged throughout all layers and all pruning percentages (0%, 40%, 55%, 70%, 85%), but varies among the band compression techniques (none, PCA, NMF) as shown in Figure 49. Because of the absolute lack of change for the layer variation, we did not feel the need to vary filter indices for the activation map, i.e. filters within the selected layer.

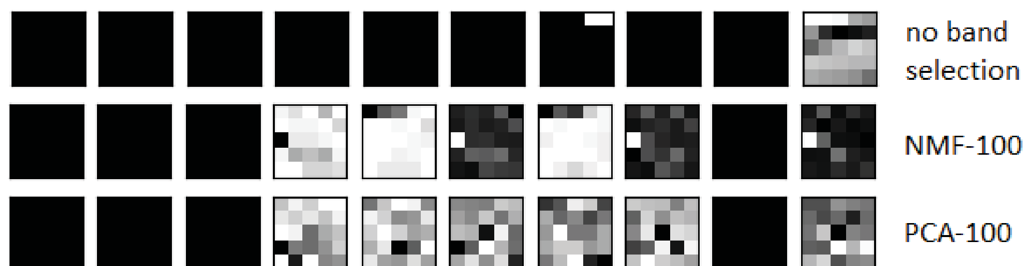


Figure 49: Activation maps for bands 0 to 90 in increments of 10 from left to right for any pruning percentage and layer generated with the `visualize_activation` method. We see that entirely black patches as the best model input for no band selection may or may not remain black when we use band selection. For example, bands 30, 40 and 60 seemed particularly relevant for NMF with 100 bands, whereas PCA shows additional interest in the bands 50, 70 and 90 (at least more obviously than NMF).

While the `visualize_activation` method generates the best model input for maximizing the outputs of all filters in the layer, the `visualize_cam` method generates a gradient-based class activation map for that purpose. What we obtained for the latter method is not worth presenting, though, as the patches for the dense layers remain entirely black and the flatten layer shows a diagonally rising gradient (the method did not work for the convolutional layers), which confirms our decision to implement our own methods. As for `visualize_activation` though, Marcos et al. employed semantically interpretable activation maps (SIAM) for an RGB dataset in September 2019, which they claim helps to determine "what attributes in the image are contributing to the final score and where they are located" [327]. They use linear combinations of "predefined attributes at different locations of the image" for realizing their SIAM idea [327], which might be worth using for building upon our work in a visualization-centric manner.

For the guided backpropagation map we obtained at zero pruning, we wanted to find out if there are any cues as to how the different gradients between the layers relate. For a pruned model, guided backpropagation did not work because certain tensors could not be found in the TensorFlow graph to which the Keras model was copied so that the ReLU gradient got overwritten with guided ReLU, so we left out that part. Aside from occasional structural similarities between weights and biases for guided backpropagation for the non-pruned model, we could not gain any helpful insight. This is part of the reason why we do not think it would be useful to compare this finding of no finding with current papers like Springenberg et al.'s, who also do not reason about why the visualization looks the way it does, apart from proposing it as a new approach and claiming it gave them cleaner visualizations than the deconvolutional network approach of Zeiler and Fergus [46, 103], with the other part being that the scenarios are too different (especially hyperspectral vs. RGB) to compare without solid scientific background of why pixel X looks the way it does and how the gradient changes according to influence Y.

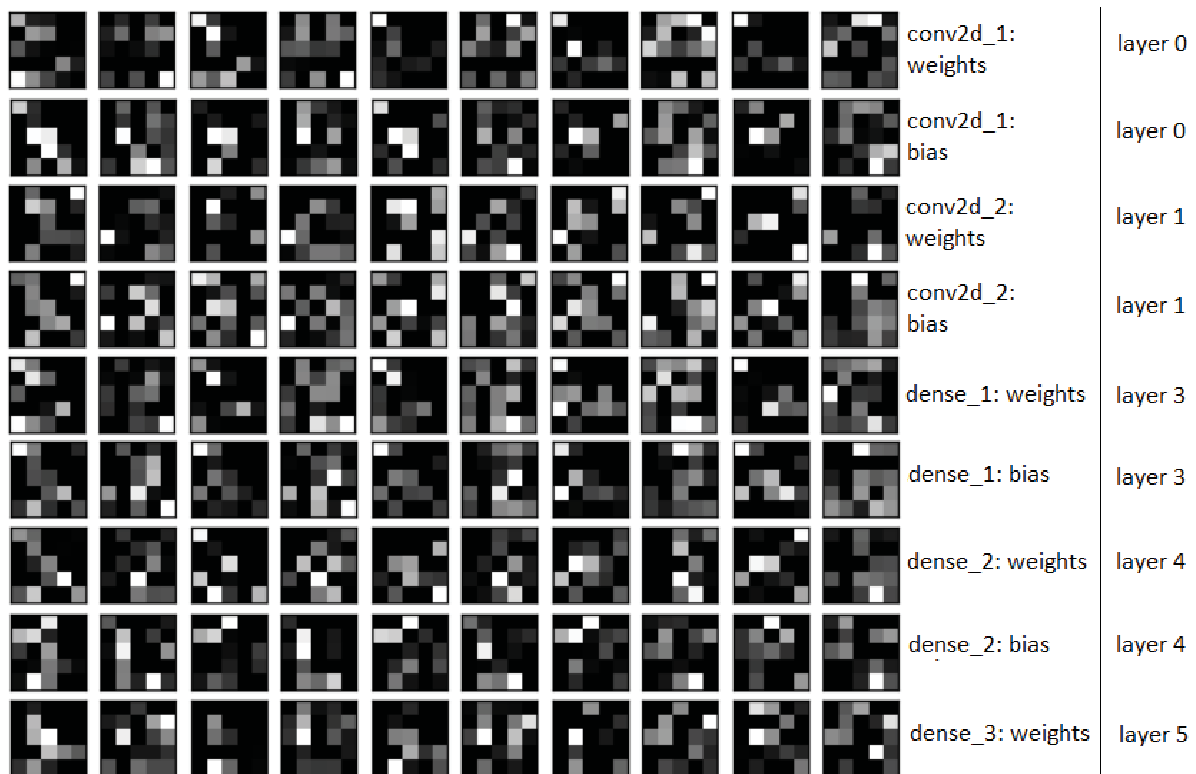


Figure 50: Saliency maps obtained through guided backpropagation for bands 0 to 90 in increments of 10 from left to right for no pruning. On occasion, the weights and biases for a chosen layer might look similar if we look at dots for very high gradients or line structures (e.g., the `dense_1` layer), but this very well might be coincidence.

5. Outlook

To circumvent PyTorch's limitations, we have coded a minimal Keras implementation based on 4.1.4 which allows us to measure model size and inference time for one pruning variation and band selection. Analogous to our inference time measurements for the cao model in this Keras implementation, one could expand it to include RAM and VRAM measurements as we did in the DeepHyperX reference measurement (cf. 4.2.1). Not taking into account the reference measurement and the combinations of compressions we have done, the table below captures the measured metrics for the compressions and outlines the challenges to make measuring the other metrics work or meaningful.

	Pruning		**Quantization**		**Band selection**
	fine-grained (weight-specific)	*coarse-grained* (channel-specific)	*fine-grained* (component-specific)	*coarse-grained* (all components or hybrid quantization)	
Overall accuracy	done	done	done	done	done
Model size	PyTorch limitation, but could calculate	Keras minimal	ONNX export is work in progress	ONNX	done
Inference time	PyTorch limitation, so no improvement	Keras minimal	ONNX export is work in progress	got worse - need special hardware	Keras minimal

Figure 51: Table summarizing the metrics measured for the compressions shown. Inference time measurements do not make sense in PyTorch, but considering Intel Distiller's lack of ONNX export for quantized models and Distiller being the only suite suitable for us to support fine-grained quantization, we could not export and convert the quantized model to any other language. However, we did implement our minimal Keras implementation to measure these metrics, but that is only possible for models whose structures are possible to express in Keras, which is very difficult for, e.g., tensor splitting. PyTorch's lack of physical model size reduction, on the other hand, causes inference times not to shrink after (fine-grained) pruning, so there is no point in conducting in-depth inference time measurements in this case. The color scheme green/yellow/orange denotes whether we could achieve results / experienced minor or circumventable difficulties / were stuck due to force majeure.

The entries denoted with "Keras minimal" are our Keras measurements, for which we can see appropriate diagrams in the appendix. Figure 71 shows us the inference times with regard to the band selection technique used (none, PCA, NMF) depending on the pruning percentage. As described in the caption, we see that **prior band selection does help in cutting the inference time by roughly 20% and that more pruning does overall mean a decreasing inference time**. Although the absolute numbers do not matter anyway as they are incomparable to our PyTorch results (due to the different framework and both our and the framework's implementations used, but also because we measured the inference time for all patches as opposed to all bands), the reason we did this visualization and included it in the outlook was to have this (promising) trend visible for the inference times which a researcher could build upon if he wished to, e.g., by including other datasets and models (or easier models to include). PyTorch's inference time measurements we have conducted had the same result but for measuring inaccuracies with and without pruning as the model could not physically shrink in size, which would have been just irritating to have as an allegedly representative result for the pruning's influence on the inference time. The inference times of Figure 72 reveal that the difference between having had a prior band selection like PCA or NMF and not having applied one becomes way more relevant after quantization if we measure the absolute values (whereas the non-quantized variant of Figure 71 shows very scattered dots). As for the relative inference time savings, Figure 73 shows that the fraction of the initial inference time decreases **by little for pruning** (with the dots scattered a lot), **but by a lot for the quantization on top**, where we witness a linear decrease. However, to not be misled by the graph, we should keep in mind that the absolute reference values are about 20 times as high for pruning plus quantization as they are for pruning only because we did not run the `tflite` model on a platform which supported the quantized operations as opposed to unboxing the INT8 quantized weights to FP32 and boxing them to INT8 again (that is how the hybrid quantization works on an unsupported platform), so this is only logical. Finally, Figure 74 shows us how much pruning and quantization are able to reduce the model size. Pruning linearly decreases the model size from its original 4.5MB to almost zero bytes, but quantization matters more because it reduces

the model size from 4.5MB to 1.5MB right off the bat, i.e. at zero percent pruning, followed by a linear decrease to almost zero bytes. The band selection techniques used help lower the model size by 1MB (pruning) / 0.25MB (pruning plus quantization). This direct measurement would not have been possible in PyTorch, where the model size (of 4,465KB without band selection of cao on IndianPines, so not that different from our Keras base size, which definitely makes sense as framework-specific overheads / metadata should not account for much in our estimation) would have remained constant and we would have to calculate the model size savings manually by searching for zeroed weights/channels and determining their contribution to the total model size.

Throughout our hyperspectral compression journey, the one thing we noticed is that lots of tools are either not implemented or being implemented / expanded, which posed a formidable challenge to us. Starting from Intel Distiller's lack of ONNX export for quantized models (due to PyTorch allegedly still developing support for quantized operations, and ONNX being a relatively new project motivated by interoperability) to the difficulties of ONNX quantization and export from a TensorFlow or ONNX model to a Keras model (circumventing the error messages of TensorFlow's TOCO quantizer was hard enough), to the lack of support for five-dimensional tensors (WinMLTools and Intel Distiller, though we fixed this for Distiller ourselves by modifying the source code), all the way to hyperspectral visualization for TensorFlow Lite models and PyTorch Lite as a means of mobile deployment / TensorFlow Lite not working in our sample app, which is hindered by memory and file size constraints of the hyperspectral use case anyway, we have encountered all sorts of difficulties we had to maneuver around. We did the best we could with the current possibilities, which included DeepHyperX for PyTorch as the only hyperspectral toolkit we could reliably build upon, despite PyTorch's model size constraints and challenging low-level specification (which, on the other hand, allows for complex model structures not possible in, e.g., Keras at the moment), but to avoid frustration amid time and resource constraints, we can only recommend practically motivated further research in this area that combines two novel topics, i.e. "hyperspectral" and "neural network compression", once these tool cornerstones are fully constructed. Still, if we take a look at current research trends and the most popular deep learning frameworks in 2019 for both research and production, "every major conference in 2019 has had a majority of papers implemented in PyTorch", presumably due to PyTorch's simplicity, not rapidly changing API design and performance, with its top contender TensorFlow being popular for industry applications, holding advantages in terms of no native need for Python and addressing mobile and serving concerns [328]. An optimistic impression can also be held regarding research on deep learning applied on HSI: looking at Figure 52, this combination of trending topics has been a popular field of research in recent years and is becoming ever more so.

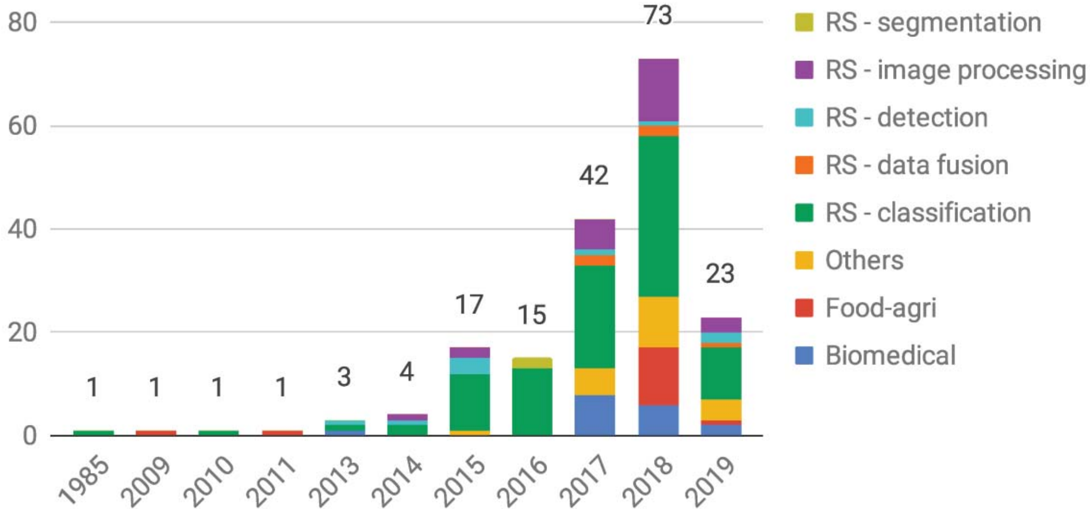


Figure 52: Number of articles about deep learning for HSI per year with regard to the application domains (RS = remote sensing), where "[t]he last column comprises published and in-press papers found up to 31 January 2019" [13]. We see an almost exponential increase of articles, which is particularly visible between 2016 and 2018, being mostly attributed to the popularity increase of remote sensing classification, but also due to new HSI applications like food and agriculture [13].

6. Conclusion

This thesis analyzed the implications of compression on multiple levels for hyperspectral models and datasets given the state-of-the-art frameworks suitable for this purpose. Our reference execution revealed that the suitability of both datasets and models greatly varies for evaluating model compression techniques. Our conclusion here is that the well-known datasets IndianPines, SalinasA and PaviaU offer a good benchmark, with the models lee, li, he, liu and cao outperforming the others. CNNs are indeed a good method for HSI classification tasks – the mou RNN did not perform particularly well, just like the machine learning methods overall. We found the VRAM and model size metrics to be correlated very strongly. To proceed with different models in terms of architecture and ideas for the upcoming compressions, we chose the models he, hu, luo, santara and cao.

Our feature extraction experiments have shown that PCA was the best-performing method in terms of overall accuracy. Just like NMF, both techniques produced the best model-independent results and have a low runtime. LLE can also yield good results due to the locality of features considered (unlike the global approach of ISOMAP). There was no benefit in using the PCA variations IncrementalPCA, KernelPCA and SparsePCA instead of the pure PCA. Similar techniques like SVD also produced only mediocre results, as did LDA. LLE and LDA have the disadvantage of limited flexibility in usage due to constraints for the number of components that can be used. The non-linear projection-based techniques UMAP, t-SNE and ISOMAP, besides having a long runtime, produced awful results, leading us to believe that considering non-linearity apparently is not as important for the IndianPines, SalinasA and PaviaU datasets. ICA and the boulch autoencoder also cannot be recommended. The rather exotic, because rarely used methods of GaussianRandomProjection, SparseRandomProjection, MultidimensionalScaling, MiniBatchDictionaryLearning, FactorAnalysis and FeatureAgglomeration all failed to cause a good OA. As for the influence of the number of components, for PCA, NMF and LLE we first witnessed a fast rise for the first few components, followed by the rise getting increasingly smaller. However, having too many components might lead to an accuracy dent. Many components can be saved without sacrificing accuracy, e.g., 100 out of IndianPines' 200 total bands. In fact, feature extraction techniques can significantly improve the baseline accuracy. Altogether, the significant and consistent improvements beyond the OA baselines for most, if not all models, make us recommend the feature extraction methods PCA, NMF and LLE as we think their good reputation is justified.

The feature selection experiments have shown a different picture. Mainly, the analyzed LinearRegression, LogisticRegression and RandomForest techniques are more unpredictable because there is lots of fluctuation around their overall linear trend respectively. It is rare for one of the techniques to surpass a reference OA. In fact, only logistic regression succeeds in doing this at times for component numbers that are undoubtedly more specific than for the well-performing feature extraction methods, so they need to be chosen with more care then employing a feature selection method. The linear curve for more components can also be a linear decrease (cf. LogisticRegression on the cao model), though most often, it is an increase. In general, we do not think feature selection is advisable to use due to too much uncertainty on how to choose the component number, the OA unlikely to be good apart from very specific component numbers and better alternatives existing (PCA, NMF, LLE), which do not depend on the number of components as strongly. The following table serves as a compact comparison of the image compression experiments at one sight for $A = \{10,20, \dots ,190\}$ describing the number of components remaining after applying the band selection method.

	he	hu	luo	santara	cao
Reference OA	94.55%	47.46%	41.41%	76.54%	94.81%
PCA	A	A	$A \setminus \{110,120,130,140\}$	A	$A \setminus \{50,70\}$
NMF	$A \setminus \{120,130,140\}$	$\emptyset$	$\emptyset$	A	A
LLE	$\emptyset$	$\emptyset$	$\emptyset$	$\{20,\dots,100\}$	$\{10\} \cup \{30,\dots,100\}$
Logistic Regression	$\emptyset$	$\{20,30,40,60\}$	$\emptyset$	$\emptyset$	$\{10,30\}$

Figure 53: Table showing the numbers of components as a subset of $A = \{10,20,\dots,190\}$ for which the achieved OA surpasses the respective reference OA; techniques that are not shown, but used in the experiment, never managed to surpass a reference OA, especially including LinearRegression and RandomForest, but also other exotic techniques. Note that LLE did not work for more than 100 components due to its component requirements. The color scheme green/yellow/orange denotes how satisfying the results are in terms of a high/mediocre/bad reference OA and variability of the band selection technique in terms of the choice of the number of components.

For model compression, we have considered fine-grained pruning, coarse-grained pruning and post-training quantization. The layer-based variation for the threshold-based pruning we used turned out to be useful to account for giant, mostly not that relevant layers regarding importance of each parameter, as opposed to smaller layers. Nonetheless, the numbers for the alpha parameters to somehow get a working fine-grained configuration in place need to be found out first, e.g., the experimental way, but this is an effort. It is compounded by practical challenges that custom implementations are most of the times required for pruning in popular frameworks (cf. PyTorch's `iterative_pruning.py` or Keras' not natively integrated APoZ pruning; same with TensorFlow, which has been undergoing changes to simplify the pruning API), making pruning complicated for non-experts. Fortunately, the APIs are catching up, less custom code is required, and/or the code for pruning is becoming more intuitive among different popular deep learning frameworks, but even then, the choice of the pruning criterion can be the next problem because it is such a specific question with so many variants that in practice, probably any default criterion will suffice. Being able to use pruning out-of-the-box might take some time, but we feel it is the correct direction to be heading towards. Once one gets pruning to work, our results show that using it until 40-50% in what we identified as the first pruning phase, where accuracy remains constant overall, is absolutely recommendable, as model size is saved according to the pruning percentage with no accuracy loss whatsoever. In fact, the OA might very well be improved. At the same time, for the model size saving to have the effect of physically reducing the file size roughly according to the pruning percentage used (as opposed to still having to cope with space-consuming masked zero parameters), it is advisable to use a framework that supports physical model size reduction (e.g., TensorFlow or Keras). We chose PyTorch due to its superiority for research purposes (and research frameworks like Intel Distiller being available in PyTorch) because, e.g., it allows for complicated models splitting tensors in ways we could not express in Keras just like that. However, once a decision for a deep learning framework is made, we advise to stick with that framework for now, as our journey regarding interoperability with ONNX and appropriate converters has shown that they are currently very experimental for productive / reliable use, or specialized use (five-dimensional tensors because of the hyperspectral aspect of our thesis) at this stage and might end up not working at all. Coming back to the pruning phases, the first phase up until 40% pruning or, for models with a reasonable (not too high) number of parameters, 50% pruning with good, overall constant OA that might be better than the reference OA is followed by a predictable accuracy decrease (up to 80%/90%), ending with a steep, unpredictable decrease afterwards.

For coarse-grained pruning, PyTorch is architecturally not well-suited for this task since a change of the number of output channels severely affects the `forward` function due to dimensionality mismatch, resulting in errors. On the bright side, Intel Distiller can explore lots of possibilities regarding criterion variation if desired, once one adjusts the 4D tensor code fragments to work with 5D. The curve we obtained looked much clearer than the ones we saw for fine-grained pruning, perhaps because channels

of different layers were being pruned as equally as possible (but this is speculation that needs to be evaluated), whereas for fine-grained pruning, this was a matter of adequately adjusting the alpha values for the different layers, which we (as the results show, rightfully) set differently for, e.g., the luo model with its giant linear layer. In detail, the constant phase can be identified until 40% and the slight fall until 75%. Pruning entire output channels means that connections are bundled together, but this coarse granularity might not account for the fact that connections from different output channels are the least relevant, and by removing an entire channel, more important connections might get removed as well (although arguably, they, too, are factored into the channel removal process according to the pruning criteria, making such a removal less likely). Because of its better accuracies and easier usability for PyTorch, we continued with and recommend fine-grained pruning over coarse-grained pruning. This answer might vary for different frameworks, though. For example, there is an easy-to-find and effective regarding accuracy APoZ-based channel-pruning implementation for Keras, which does not lead to any errors, so it might be preferable for Keras. What matters is to prune at all if model size savings are important. The related work we cited shows that what matters is the architecture defined by the fine-grained pruning, but there are concerns regarding efficient computation amid the very fine-grained sparsity (connection holes) created, but these are being tackled, so for a faster inference time, coarse-grained pruning is preferable, but for model size reductions, either method works well [329, 330].

Our experience with post-training quantization shows that it is for a reason that TensorFlow's / Keras' default quantizer only quantizes weights [307]. The experiments have shown that it is not wise to quantize accumulators, preferably at all, or if absolutely necessary, one should not use fewer than $2n + \log_2(c * k^2)$ bits for n-bit integers, c input channels and the kernel width k [238], which we could confirm. We have seen that activation quantization is riskier than weight quantization, but not nearly as risky as accumulator quantization due to their central role of accumulating values. However, activation quantization is trickier than weight quantization because to quantize activations, one needs to get to know their values first, and these have to be calculated, unlike weights, which are known right off-the-bat. Speaking of which, separate component quantization is risky in general because if one messes up by using too few bits, one might experience devastating OAs like 0.496% like we did. Presumably, this is part of the reason why the default behavior of TensorFlow's integer quantizer is to just consider the weights and activations, not the accumulators [321]. The best bit triples for activations/weights/accumulators we have found are (8,16,32) for the best OA, (8,8,32) for a middle ground and (8,4,16) for a budget alternative, regardless of the model used. The model size savings are very real, but the chances of improving the OA are particularly slim, unlike for pruning. One can only hope for an OA reduction which is as low as possible. The mobile inference we have tried to evaluate the quantized models, e.g., a `tflite` model on an Android phone, are a huge issue because currently, the dataset alone consumes too much storage, even when just using a tiny fraction of it (like 1%), and the inference also crashes because of operations like reshaping a tensor by the TOCO quantizer, which we did not want to happen, but had no control over. To experience the speedup, specialized hardware is therefore probably needed, otherwise, the lower-precision calculations are not taken advantage of if the platform running the neural network does not support the quantized operations. In this case, the wrapping overhead from INT8 to FP32 when finding that the platform is unsupported causes the inference time to increase.

We have tried two compression pipelines, finding that the effects of individual compressions accumulate, which we can use to our advantage. Using band selection followed by fine-grained pruning resulted in marvelous OAs for long stretches of pruning percentages, which are way longer than for pruning alone: for instance, for PCA and fine-grained pruning applied on the he model, all values until 70% pruning percentage vary between 99.74% and 100% OA. After that, a gap between different PCA component numbers comes into existence, which widens with increasing pruning percentage: a higher component number leads to a slightly longer retention of the splendid OA and to a less steep descent after that point is reached. We can replace PCA with any other well-performing band selection technique like NMF in the compression pipeline with the same result of accumulating OA improvements. Conversely, what performed poorly and unpredictably on its own will have the same effect in the compression pipeline,

which we have shown for LinearRegression. Since we are aiming for good OAs, we can recommend moderate fine-grained pruning (40%-50%) together with PCA for the best OAs for combined image and model compression (unless a better band selection or pruning technique exists which we are simply not aware of). Regarding model size considerations, using band selection at all, regardless of the precise technique used, results in a reduced model size, which depends on the number of components used (e.g., 70% savings for PCA-170 on he vs. additional 20% savings for using PCA-40 instead - both savings compared to the original model, i.e. with no band selection). There are exceptions like the hu model, where model size is shaped in four linear rays, and other models, whose sizes are also reduced with fewer components, but not as much due to the lack of steepness of their linear model size graphs (santara, cao). Large models like luo are where the compression pipeline truly shines regarding model size reductions, as massive savings are possible through the combination of fine-grained pruning and fewer components, in case one wants to surpass the methods' individual contributions.

If we expand the pipeline with post-training quantization, its effects, which stay the same as in the single application, are also accumulated. Consequently, no OA improvements are visible due to post-training quantization (to be more precise: there were extremely rare occurrences for post-training quantization alone where this happened – OA improvements in the pipeline due to post-training quantization are just as rare). Depending on how much one values model size reductions, our recommendations of (8,16,32), (8,8,32) and (8,4,16) as bit triples for the component-wise quantization of activations/weights/accumulators remain the same.

Our visualizations with gradient-based saliency maps do reflect the development for increasing pruning percentages well in our opinion. They give us the confidence that important neurons remain unimpaired until high pruning percentages are reached. This means that gradient patterns remain unchanged or that they slightly change, which is visually represented by a different shading. For the third pruning phase, one does see big location and magnitude-related changes of the gradients, which is exactly how it should be for a fitting visualization. The NMF visualizations make for a more stable pattern and fewer changes, likely due to the non-negativity constraint, which does not exist for PCA, causing more distinct variations for PCA's gradient-based saliency maps by comparison. Activation maps can be generated, but they absolutely do not change for any pruning percentage. They just vary among different band selection techniques. Guided backpropagation did not give us an insight either and highlighted the implementation-related challenges because the visualization generation did not work for pruned models due to tensors that could not be found.

All of our experiences of conducting the above work show that from a practical standpoint, there is still some way to go for deep learning frameworks regarding their interoperability, the availability, user-friendly usage and robust applicability of techniques like pruning, quantization and visualization for neural networks used for hyperspectral image classification purposes, but we are confident that these challenges will be mastered in the near future. This would open up the theoretical potential of dimensionality reduction on different levels, as we have researched, analyzed and evaluated in this thesis, to ordinary end-users and the industry alike.

7. Bibliography

1. Krizhevsky, A., I. Sutskever, and G.E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks.* Commun. ACM, 2017. **60**(6): p. 84-90 [cited 17.08.2019]; Available from: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.
2. Qin, J., et al., *Hyperspectral and Multispectral Imaging for Evaluating Food Safety and Quality.* Journal of Food Engineering, 2013. **118**: p. 157–171 [cited 17.08.2019]; Available from: https://pubag.nal.usda.gov/download/56643/PDF.
3. Thenkabail, P., et al., *Hyperspectral Remote Sensing of Vegetation and Agricultural Crops.* Photogrammetric Engineering and Remote Sensing, 2014. **80** [cited 17.08.2019]; Available from: https://www.researchgate.net/publication/264422171_Hyperspectral_Remote_Sensing_of_Vegetation_and_Agricultural_Crops.
4. Park, B. and R. Lu, *Hyperspectral Imaging Technology in Food and Agriculture*. Food Engineering Series, ed. G.V. Barbosa-Cánovas. 2015. 1-390 [cited 27.09.2019]; Available from: https://www.springer.com/gp/book/9781493928354.
5. Goetz, A., *Three Decades of Hyperspectral Remote Sensing of the Earth: A Personal View.* Remote Sensing of Environment - REMOTE SENS ENVIRON, 2009. **113** [cited 17.08.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S003442570900073X.
6. Adão, T., et al., *Hyperspectral Imaging: A Review on UAV-Based Sensors, Data Processing and Applications for Agriculture and Forestry.* Remote Sensing, 2017. **2017**: p. 1110 [cited 17.08.2019]; Available from: https://www.mdpi.com/2072-4292/9/11/1110.
7. Burger, J. and A. Gowen, *Data Handling in Hyperspectral Image Analysis.* Chemometrics and Intelligent Laboratory Systems - CHEMOMETR INTELL LAB SYST, 2011. **108**: p. 13-22 [cited 17.08.2019]; Available from: https://www.sciencedirect.com/science/article/abs/pii/S0169743911000761.
8. Ma, W., et al. *The Hughes Phenomenon in Hyperspectral Classification Based on the Ground Spectrum of Grasslands in the Region Around Qinghai Lake*. 2013. [cited 14.08.2019]; Available from: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/8910/89101G/The-Hughes-phenomenon-in-hyperspectral-classification-based-on-the-ground/10.1117/12.2034457.full.
9. Wang, J. and C.-I. Chang, *Independent Component Analysis-Based Dimensionality Reduction with Applications in Hyperspectral Image Analysis.* IEEE Transactions on Geoscience and Remote Sensing, 2006. **44**(6): p. 1586-1600 [cited 08.10.2019]; Available from: https://ieeexplore.ieee.org/document/1634722.
10. Chi, M., R. Feng, and L. Bruzzone, *Classification of Hyperspectral Remote-Sensing Data with Primal SVM for Small-Sized Training Dataset Problem.* Advances in Space Research, 2008. **41**: p. 1793-1799 [cited 17.08.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S0273117708000677.
11. Li, S., et al., *A Novel Approach to Hyperspectral Band Selection Based on Spectral Shape Similarity Analysis and Fast Branch and Bound Search.* Eng. Appl. Artif. Intell., 2014. **27**(C): p. 241-250 [cited 17.08.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S0952197613001383.
12. Han, S., H. Mao, and W. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. [cited 04.08.2019]; Available from: https://arxiv.org/abs/1510.00149.
13. Signoroni, A., et al., *Deep Learning Meets Hyperspectral Image Analysis: A Multidisciplinary Review.* Journal of Imaging, 2019. **5**: p. 52 [cited 28.09.2019]; Available from: https://www.mdpi.com/2313-433X/5/5/52/htm.
14. Roy, S.K., et al., *HybridSN: Exploring 3D-2D CNN Feature Hierarchy for Hyperspectral Image Classification.* CoRR, 2019. **abs/1902.06701** [cited 02.08.2019]; Available from: http://arxiv.org/abs/1902.06701.
15. Landgrebe, D., *Hyperspectral Image Data Analysis.* IEEE Signal Processing Magazine, 2002. **19**(1): p. 17-28 [cited 26.09.2019]; Available from: https://ieeexplore.ieee.org/document/974718.
16. Wong, G., *Snapshot Hyperspectral Imaging and Practical Applications.* Journal of Physics: Conference Series, 2009. **178**: p. 012048 [cited 08.11.2019]; Available from: http://dx.doi.org/10.1088/1742-6596/178/1/012048.
17. Kohler, D.D.R., et al., *New Approach for the Radiometric Calibration of Spectral Imaging Systems.* Optics Express, 2004. **12**(11): p. 2463-2477 [cited 08.11.2019]; Available from: http://www.opticsexpress.org/abstract.cfm?URI=oe-12-11-2463.
18. Bioucas-Dias, J.M., et al., *Hyperspectral Remote Sensing Data Analysis and Future Challenges.* IEEE Geoscience and Remote Sensing Magazine, 2013. **1**(2): p. 6-36 [cited 26.10.2019]; Available from: https://ieeexplore.ieee.org/document/6555921.
19. Zhang, B., et al., *Hyperspectral Image Processing and Analysis System (HIPAS) and Its Applications.* Photogrammetric Engineering and Remote Sensing, 2000. **66**: p. 5 [cited 26.09.2019]; Available from: https://www.researchgate.net/publication/272093100_Hyperspectral_Image_Processing_and_Analysis_System_HIPAS_and_Its_Applications.
20. Wikipedia. *Hyperspectral Imaging*. 2018 [cited 26.09.2019]; Available from: https://en.wikipedia.org/wiki/Hyperspectral_imaging.
21. Hagen, N. and M. Kudenov, *Review of Snapshot Spectral Imaging Technologies.* Optical Engineering, 2013. **52**: p. 090901 [cited 07.11.2019]; Available from: https://www.researchgate.net/publication/275085644_Review_of_snapshot_spectral_imaging_technologies.
22. Wikipedia. *Electromagnetic Spectrum*. 2019 [cited 26.09.2019]; Available from: https://en.wikipedia.org/wiki/Electromagnetic_spectrum.

23. Uzair, M., A. Mahmood, and A. Mian, *Hyperspectral Face Recognition With Spatiospectral Information Fusion and PLS Regression.* IEEE Transactions on Image Processing, 2015. **24**(3): p. 1127-1137 [cited 26.09.2019]; Available from: http://ieeexplore.ieee.org/document/7010906.
24. Meier, R.R., *Ultraviolet Spectroscopy and Remote Sensing of the Upper Atmosphere.* Space Science Reviews, 1991. **58**(1): p. 1-185 [cited 26.09.2019]; Available from: https://doi.org/10.1007/BF01206000.
25. Coulter, D.W., P.L. Hauff, and W.L. Kerby, *Airborne Hyperspectral Remote Sensing.* Advances in Airborne Geophysics, 2007 [cited 27.09.2019]; Available from: https://www.semanticscholar.org/paper/Airborne-Hyperspectral-Remote-Sensing-Coulter/3fbb92ddefc39deb240bb006f85687c573c40029.
26. Backman, V., et al., *Detection of Preinvasive Cancer Cells.* Nature, 2000. **406**(6791): p. 35 [cited 08.11.2019]; Available from: https://www.nature.com/articles/35017638.
27. Krutz, D., et al., *The Instrument Design of the DLR Earth Sensing Imaging Spectrometer (DESIS).* Sensors (Basel, Switzerland), 2019. **19**(7): p. 1622 [cited 08.11.2019]; Available from: https://www.ncbi.nlm.nih.gov/pubmed/30987374.
28. Eckardt, A., et al. *DESIS (DLR Earth Sensing Imaging Spectrometer for the ISS-MUSES platform)*. in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. 2015. [cited 08.11.2019]; Available from: https://ieeexplore.ieee.org/document/7326053.
29. Transon, J., et al., *Survey of Hyperspectral Earth Observation Applications from Space in the Sentinel-2 Context.* Remote Sensing, 2018. **10**(2): p. 157 [cited 08.11.2019]; Available from: https://www.mdpi.com/2072-4292/10/2/157.
30. Lu, G. and B. Fei, *Medical Hyperspectral Imaging: A Review.* Journal of biomedical optics, 2014. **19**(1): p. 10901-10901 [cited 07.11.2019]; Available from: https://www.ncbi.nlm.nih.gov/pubmed/24441941.
31. Grusche, S., *Basic Slit Spectroscope Reveals Three-Dimensional Scenes Through Diagonal Slices of Hyperspectral Cubes.* Applied Optics, 2014. **53**(20): p. 4594-4603 [cited 07.11.2019]; Available from: http://ao.osa.org/abstract.cfm?URI=ao-53-20-4594.
32. Cao, X., et al., *Computational Snapshot Multispectral Cameras: Toward Dynamic Capture of the Spectral World.* IEEE Signal Processing Magazine, 2016. **33**(5): p. 95-108 [cited 08.11.2019]; Available from: https://ieeexplore.ieee.org/document/7559979.
33. Oppenheim, A.V., *Discrete-Time Signal Processing*. 1999: Pearson Education India [cited 08.11.2019]; Available from: http://entsphere.com/pub/pdf/Discrete-Time%20Signal%20Processing%20-%202ed%20-%20Oppenheim.pdf.
34. Sun, J., Q. Lv, and J. Yin, *Applications of CS Based Spectrum Recovery in Hyperspectral Images.* International Conference on Signal Processing Proceedings, ICSP, 2015. **2015**: p. 928-933 [cited 08.11.2019]; Available from: https://www.researchgate.net/publication/282233818_Applications_of_CS_based_spectrum_recovery_in_hyperspectral_images.
35. Candès, E., *Compressive Sampling.* Proceedings oh the International Congress of Mathematicians, Vol. 3, 2006-01-01, ISBN 978-3-03719-022-7, pags. 1433-1452, 2006. **3** [cited 08.11.2019]; Available from: https://www.researchgate.net/publication/41537648_Compressive_sampling.
36. Chang, C.-I., *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. 2003, New York: Kluwer Academic/Plenum Publishers [cited 27.09.2019]; Available from: https://books.google.de/books?hl=de&lr=&id=JhBbXwFaA6sC&oi=fnd&pg=PA1&dq=Hyperspectral+imaging&ots=r2iNz_D1yQ&sig=1JCrwTkMa9Kg2Uh18QFlPVg1yo0#v=onepage&q=Hyperspectral%20imaging&f=false.
37. Hughes, G., *On the Mean Accuracy of Statistical Pattern Recognizers.* IEEE Transactions on Information Theory, 1968. **14**(1): p. 55-63 [cited 28.09.2019]; Available from: https://ieeexplore.ieee.org/document/1054102.
38. Cavallaro, G. *Introduction to Deep Learning for Remote Sensing & 1D/2D CNNs for Hyperspectral Images*. 2019 [cited 27.09.2019]; Available from: http://www.morrisriedel.de/wp-content/uploads/2019/05/Lecture_5_Introduction_to_Deep_Learning_for_Remote_Sensing_and_1D-2D_CNNs_for_Hyperspectral_Images_Classification.pdf.
39. Kondermann, D., *Ground Truth Design Principles: An Overview*, in *Proceedings of the International Workshop on Video and Image Ground Truth in Computer Vision Applications*. 2013, ACM: St. Petersburg, Russia. p. 1-4. Available from: http://dl.acm.org/ft_gateway.cfm?id=2501114.
40. Khan, A., et al. *A Survey of the Recent Architectures of Deep Convolutional Neural Networks*. arXiv e-prints, 2019. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190106032K.
41. Pan, B., Z. Shi, and X. Xu, *R-VCANet: A New Deep-Learning-Based Hyperspectral Image Classification Method.* IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2017. **10**(5): p. 1975-1986 [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/7855674.
42. Zahangir Alom, M., et al. *The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches*. arXiv e-prints, 2018. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180301164Z.
43. Livingstone, D.J., *Artificial Neural Networks: Methods and Applications (Methods in Molecular Biology)*. 2008: Humana Press. 254 [cited 09.11.2019]; Available from: https://link.springer.com/book/10.1007/978-1-60327-101-1.
44. Zell, A., *Simulation neuronaler Netze*. 2003, München: Oldenbourg [cited 30.09.2019]; Available from: https://books.google.com/books/about/Simulation_neuronaler_Netze.html?hl=de&id=bACTSgAACAAJ.

45. Dawson, C.W. and R. Wilby, *An Artificial Neural Network Approach to Rainfall-Runoff Modelling.* Hydrological Sciences Journal, 1998. **43**(1): p. 47-66 [cited 30.09.2019]; Available from: https://doi.org/10.1080/02626669809492102.
46. Springenberg, J.T., et al. *Striving for Simplicity: The All Convolutional Net*. arXiv e-prints, 2014. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1412.6806S.
47. Scherer, D., A. Müller, and S. Behnke, *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*, in *Proceedings of the 20th international conference on Artificial neural networks: Part III*. 2010, Springer-Verlag: Thessaloniki, Greece. p. 92-101. Available from: https://www.researchgate.net/publication/221080312_Evaluation_of_pooling_operations_in_convolutional_architectures_for_object_recognition.
48. Wang, S., et al., *Multiple Sclerosis Identification by 14-Layer Convolutional Neural Network With Batch Normalization, Dropout, and Stochastic Pooling.* Frontiers in Neuroscience, 2018. **12**: p. 818 [cited 01.10.2019]; Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6236001/.
49. Yu, D., et al., *Mixed Pooling for Convolutional Neural Networks*. 2014. 364-375 [cited 01.10.2019]; Available from: https://www.researchgate.net/publication/300020038_Mixed_Pooling_for_Convolutional_Neural_Networks.
50. Gu, J., et al. *Recent Advances in Convolutional Neural Networks*. arXiv e-prints, 2015. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151207108G.
51. Li, Y., H. Zhang, and Q. Shen, *Spectral–Spatial Classification of Hyperspectral Imagery with 3D Convolutional Neural Network.* Remote Sensing, 2017. **9**: p. 67 [cited 02.08.2019]; Available from: https://www.mdpi.com/2072-4292/9/1/67.
52. Iliopoulos, A.-S., T. Liu, and X. Sun *Hyperspectral Image Classification and Clutter Detection via Multiple Structural Embeddings and Dimension Reductions*. arXiv e-prints, 2015. [cited 26.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150601115I.
53. Bachmann, C.M., T.L. Ainsworth, and R.A. Fusina, *Exploiting Manifold Geometry in Hyperspectral Imagery.* IEEE Transactions on Geoscience and Remote Sensing, 2005. **43**(3): p. 441-454 [cited 26.10.2019]; Available from: https://ieeexplore.ieee.org/document/1396318.
54. Chen, Y., et al., *Deep Feature Extraction and Classification of Hyperspectral Images Based on Convolutional Neural Networks.* IEEE Transactions on Geoscience and Remote Sensing, 2016. **54**(10): p. 6232-6251 [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/document/7514991.
55. Li, C., et al., *Hyperspectral Remote Sensing Image Classification Based on Maximum Overlap Pooling Convolutional Neural Network.* Sensors (Basel, Switzerland), 2018. **18**(10): p. 3587 [cited 30.09.2019]; Available from: https://www.ncbi.nlm.nih.gov/pubmed/30360445.
56. Hu, W., et al., *Deep Convolutional Neural Networks for Hyperspectral Image Classification.* Journal of Sensors, 2015. **2015**: p. 12 [cited 02.08.2019]; Available from: http://dx.doi.org/10.1155/2015/258619.
57. Fukushima, K., *Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition.* Neural Networks, 1988. **1**: p. 119-130 [cited 30.09.2019]; Available from: https://www.sciencedirect.com/science/article/pii/0893608088900147.
58. LeCun, Y., et al., *Backpropagation Applied to Handwritten Zip Code Recognition.* Neural Comput., 1989. **1**(4): p. 541-551 [cited 30.09.2019]; Available from: http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf.
59. Liu, C.-L., et al., *Handwritten Digit Recognition: Benchmarking of State-Of-The-Art Techniques.* Pattern Recognition, 2003. **36**: p. 2271-2285 [cited 01.10.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S0031320303000852.
60. Lecun, Y., K. Kavukcuoglu, and C. Farabet, *Convolutional Networks and Applications in Vision*. 2010. 253-256 [cited 01.10.2019]; Available from: https://ieeexplore.ieee.org/document/5537907.
61. Matsugu, M., et al. *Convolutional Spiking Neural Network Model for Robust Face Detection*. in *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP '02.* 2002. [cited 01.10.2019]; Available from: http://ieeexplore.ieee.org/document/1198140.
62. Ying-Nong, C., et al. *The Application of a Convolution Neural Network on Face and License Plate Detection*. in *18th International Conference on Pattern Recognition (ICPR'06)*. 2006. [cited 01.10.2019]; Available from: https://ieeexplore.ieee.org/document/1699586.
63. Fasel, B. *Facial Expression Analysis Using Shape and Motion Information Extracted by Convolutional Neural Networks*. in *Proceedings of the 12th IEEE Workshop on Neural Networks for Signal Processing*. 2002. [cited 01.10.2019]; Available from: https://ieeexplore.ieee.org/document/1030072.
64. Nickolls, J., et al., *Scalable Parallel Programming with CUDA.* Queue, 2008. **6**(2): p. 40-53 [cited 01.10.2019]; Available from: https://www.researchgate.net/publication/213877997_Scalable_Parallel_Programming_with_CUDA.
65. Lindholm, E., et al., *NVIDIA Tesla: A Unified Graphics and Computing Architecture.* IEEE Micro, 2008. **28**(2): p. 39-55 [cited 01.10.2019]; Available from: https://ieeexplore.ieee.org/document/4523358.
66. Oh, K.-S. and K. Jung, *GPU Implementation of Neural Networks.* Pattern Recognition, 2004. **37**: p. 1311-1314 [cited 01.10.2019]; Available from: https://www.researchgate.net/publication/222114533_GPU_implementation_of_neural_networks.
67. Schmidhuber, J., *Multi-Column Deep Neural Networks for Image Classification*, in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012, IEEE Computer Society. p. 3642-3649. Available from: https://ieeexplore.ieee.org/document/6248110.

68. Ben Hamida, A., et al., *3-D Deep Learning Approach for Remote Sensing Image Classification.* IEEE Transactions on Geoscience and Remote Sensing, 2018. **56**(8): p. 4420-4434 [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/8344565.
69. Lee, H. and H. Kwon. *Contextual Deep CNN Based Hyperspectral Classification*. in *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. 2016. [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/7729859/.
70. He, M., B. Li, and H. Chen. *Multi-Scale 3D Deep Convolutional Neural Network for Hyperspectral Image Classification*. in *2017 IEEE International Conference on Image Processing (ICIP)*. 2017. [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/document/8297014.
71. Luo, Y., et al., *HSI-CNN: A Novel Convolution Neural Network for Hyperspectral Image.* CoRR, 2018. **abs/1802.10478** [cited 04.08.2019]; Available from: http://arxiv.org/abs/1802.10478.
72. Yang, X., et al., *Hyperspectral Image Classification With Deep Learning Models.* IEEE Transactions on Geoscience and Remote Sensing, 2018. **PP**: p. 1-16 [cited 05.10.2019]; Available from: https://ieeexplore.ieee.org/document/8340197.
73. Smithson, S.C., et al. *Neural Networks Designing Neural Networks: Multi-Objective Hyper-Parameter Optimization*. arXiv e-prints, 2016. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161102120S.
74. Dong, J.-D., et al. *DPP-Net: Device-Aware Progressive Search for Pareto-Optimal Neural Architectures*. arXiv e-prints, 2018. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180608198D.
75. Hsu, C.-H., et al. *MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning*. arXiv e-prints, 2018. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180610332H.
76. Wistuba, M., A. Rawat, and T. Pedapati *A Survey on Neural Architecture Search*. arXiv e-prints, 2019. [cited 30.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190501392W.
77. Li, X., et al. *Partial Order Pruning: for Best Speed/Accuracy Trade-off in Neural Architecture Search*. arXiv e-prints, 2019. [cited 17.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190303777L.
78. Mattioli, F., et al., *An Experiment on the Use of Genetic Algorithms for Topology Selection in Deep Learning.* J. Electrical and Computer Engineering, 2019. **2019**: p. 3217542:1-3217542:12 [cited 30.09.2019]; Available from: https://www.hindawi.com/journals/jece/2019/3217542/.
79. Isikdogan, L.F. *How to Design a Convolutional Neural Network*. 2018 20.02.2018 [cited 30.09.2019]; Available from: http://www.isikdogan.com/blog/how-to-design-a-convolutional-neural-network.html.
80. Bottou, L., *Stochastic Gradient Descent Tricks*, in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G.B. Orr, and K.-R. Müller, Editors. 2012, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 421-436 [cited 03.10.2019]; Available from: https://doi.org/10.1007/978-3-642-35289-8_25.
81. Wijnhoven, R.G.J. and P.H.N.d. With. *Fast Training of Object Detection Using Stochastic Gradient Descent*. in *2010 20th International Conference on Pattern Recognition*. 2010. [cited 03.10.2019]; Available from: https://ieeexplore.ieee.org/document/5597822.
82. Zinkevich, M., et al., *Parallelized Stochastic Gradient Descent*, in *Advances in Neural Information Processing Systems 23*, J.D. Lafferty, et al., Editors. 2010, Curran Associates, Inc. p. 2595-2603 [cited 03.10.2019]; Available from: http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.pdf.
83. Recht, B., et al., *Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent*, in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, et al., Editors. 2011, Curran Associates, Inc. p. 693-701 [cited 03.10.2019]; Available from: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.
84. Johnson, R. and T. Zhang, *Accelerating Stochastic Gradient Descent Using Predictive Variance Reduction*, in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. 2013, Curran Associates Inc.: Lake Tahoe, Nevada. p. 315-323. Available from: https://papers.nips.cc/paper/4937-accelerating-stochastic-gradient-descent-using-predictive-variance-reduction.pdf.
85. Buntine, W.L. and A.S. Weigend, *Bayesian Back-Propagation.* Complex Systems, 1991. **5** [cited 05.10.2019]; Available from: https://pdfs.semanticscholar.org/c836/84f6207697c12850db423fd9747572cf1784.pdf.
86. Jimenez Rezende, D., S. Mohamed, and D. Wierstra *Stochastic Backpropagation and Approximate Inference in Deep Generative Models*. arXiv e-prints, 2014. [cited 03.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1401.4082J.
87. Riedmiller, M. and H. Braun. *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*. in *IEEE International Conference on Neural Networks*. 1993. [cited 03.10.2019]; Available from: https://ieeexplore.ieee.org/document/298623.
88. Vora, K. and S.B. Yagnik. *A Survey on Backpropagation Algorithms for Feedforward Neural Networks*. 2013. [cited 03.10.2019]; Available from: https://www.semanticscholar.org/paper/A-Survey-on-Backpropagation-Algorithms-for-Neural-Vora-Yagnik/fcd17e08bf906bea2f8cd306a65784901d47b91f.
89. Yeremia, H., et al., *Genetic Algorithm and Neural Network for Optical Character Recognition.* Journal of Computer Science, 2013. **9**: p. 1435-1442 [cited 03.10.2019]; Available from: https://www.researchgate.net/publication/286169640_Genetic_algorithm_and_neural_network_for_optical_character_recognition.
90. Rajasekaran, S. and G.A.V. Pai. *Neural Networks, Fuzzy Logic, and Genetic Algorithms : Synthesis and Applications*. 2013. [cited 03.10.2019]; Available from:

https://books.google.com/books/about/NEURAL_NETWORKS_FUZZY_LOGIC_AND_GENETIC.html?id=bVbj9nhvHd4C.
91. Maas, A.L. *Rectifier Nonlinearities Improve Neural Network Acoustic Models*. 2013. [cited 03.10.2019]; Available from: https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.
92. Ioffe, S. and C. Szegedy *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv e-prints, 2015. [cited 03.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150203167I.
93. James, G., et al., *An Introduction to Statistical Learning: with Applications in R*. 2014: Springer Publishing Company, Incorporated. 430 [cited 06.10.2019]; Available from: https://www.springer.com/de/book/9781461471370.
94. Prechelt, L., *Early Stopping - But When?*, in *Neural Networks: Tricks of the Trade*, G.B. Orr and K.-R. Müller, Editors. 1998, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 55-69 [cited 03.10.2019]; Available from: https://doi.org/10.1007/3-540-49430-8_3.
95. Srivastava, N., et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of Machine Learning Research, 2014. **15**: p. 1929-1958 [cited 03.10.2019]; Available from: http://jmlr.org/papers/v15/srivastava14a.html.
96. Kang, X., S. Li, and J.A. Benediktsson, *Spectral–Spatial Hyperspectral Image Classification With Edge-Preserving Filtering.* IEEE Transactions on Geoscience and Remote Sensing, 2014. **52**(5): p. 2666-2677 [cited 03.10.2019]; Available from: https://ieeexplore.ieee.org/document/6553593.
97. Vieira, S.M., U. Kaymak, and J.M.C. Sousa. *Cohen's Kappa Coefficient as a Performance Measure for Feature Selection*. in *International Conference on Fuzzy Systems*. 2010. [cited 09.11.2019]; Available from: https://ieeexplore.ieee.org/document/5584447.
98. Su, J., et al., *Dimension Reduction Aided Hyperspectral Image Classification with a Small-Sized Training Dataset: Experimental Comparisons.* Sensors, 2017. **17**: p. 2726 [cited 08.10.2019]; Available from: https://www.researchgate.net/publication/321323013_Dimension_Reduction_Aided_Hyperspectral_Image_Classification_with_a_Small-sized_Training_Dataset_Experimental_Comparisons.
99. Li, F.-F., J. Johnson, and S. Yeung. *Lecture 12 - Visualizing and Understanding*. CS231n: Convolutional Neural Networks for Visual Recognition 2017 [cited 05.10.2019]; Available from: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture12.pdf.
100. Simonyan, K., A. Vedaldi, and A. Zisserman *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. arXiv e-prints, 2013. [cited 12.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2013arXiv1312.6034S.
101. Shrikumar, A., P. Greenside, and A. Kundaje *Learning Important Features Through Propagating Activation Differences*. arXiv e-prints, 2017. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170402685S.
102. Binder, A., et al. *Layer-Wise Relevance Propagation for Neural Networks with Local Renormalization Layers*. arXiv e-prints, 2016. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160400825B.
103. Zeiler, M.D. and R. Fergus, *Visualizing and Understanding Convolutional Networks.* CoRR, 2013. **abs/1311.2901** [cited 04.08.2019]; Available from: http://arxiv.org/abs/1311.2901.
104. Sundararajan, M., A. Taly, and Q. Yan *Axiomatic Attribution for Deep Networks*. arXiv e-prints, 2017. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170301365S.
105. Erhan, D., et al. *Visualizing Higher-Layer Features of a Deep Network*. 2009. [cited 05.10.2019]; Available from: https://www.researchgate.net/publication/265022827_Visualizing_Higher-Layer_Features_of_a_Deep_Network.
106. Nguyen, A., J. Yosinski, and J. Clune *Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks*. arXiv e-prints, 2016. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160203616N.
107. Bojarski, M., et al. *VisualBackProp: Efficient Visualization of CNNs*. arXiv e-prints, 2016. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161105418B.
108. Hohman, F., et al., *Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers.* CoRR, 2018. **abs/1801.06889** [cited 04.08.2019]; Available from: http://arxiv.org/abs/1801.06889.
109. Alsallakh, B., et al., *Do Convolutional Neural Networks Learn Class Hierarchy?* CoRR, 2017. **abs/1710.06501** [cited 04.08.2019]; Available from: http://arxiv.org/abs/1710.06501.
110. Elman, J.L., *Finding Structure in Time.* Cognitive Science, 1990. **14**: p. 179-211 [cited 05.10.2019]; Available from: https://www.sciencedirect.com/science/article/pii/036402139090002E.
111. Jordan, M.I. *Serial Order: A Parallel Distributed Processing Approach*. 1997. [cited 05.10.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S0166411597801112.
112. Luo, H. *Shorten Spatial-Spectral RNN with Parallel-GRU for Hyperspectral Image Classification*. arXiv e-prints, 2018. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181012563L.
113. Mou, L., P. Ghamisi, and X.X. Zhu, *Deep Recurrent Neural Networks for Hyperspectral Image Classification.* IEEE Transactions on Geoscience and Remote Sensing, 2017. **55**(7): p. 3639-3655 [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/document/7914752.
114. Socher, R., et al., *Parsing Natural Scenes and Natural Language with Recursive Neural Networks*, in *Proceedings of the 28th International Conference on International Conference on Machine Learning*. 2011, Omnipress: Bellevue, Washington, USA. p. 129-136. Available from: https://ai.stanford.edu/~ang/papers/icml11-ParsingWithRecursiveNeuralNetworks.pdf.

115. Shi, X., et al. *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. arXiv e-prints, 2015. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150604214S.
116. Gers, F.A. and J. Schmidhuber. *Recurrent Nets That Time and Count*. in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. 2000. [cited 05.10.2019]; Available from: https://ieeexplore.ieee.org/document/861302.
117. Gers, F.A., et al., *Learning Precise Timing with LSTM Recurrent Networks.* J. Mach. Learn. Res., 2003. **3**: p. 115-143 [cited 05.10.2019]; Available from: http://www.jmlr.org/papers/volume3/gers02a/gers02a.pdf.
118. Chung, J., et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv e-prints, 2014. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1412.3555C.
119. Hang, R., et al., *Cascaded Recurrent Neural Networks for Hyperspectral Image Classification.* IEEE Transactions on Geoscience and Remote Sensing, 2019. **57**: p. 5384-5394 [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019ITGRS..57.5384H.
120. Wu, H. and S. Prasad, *Convolutional Recurrent Neural Networks for Hyperspectral Data Classification.* Remote Sensing, 2017. **9**: p. 298 [cited 05.10.2019]; Available from: https://www.mdpi.com/2072-4292/9/3/298.
121. Zhang, X., et al., *Spatial Sequential Recurrent Neural Network for Hyperspectral Image Classification.* IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2018. **11**(11): p. 4141-4155 [cited 05.10.2019]; Available from: https://ieeexplore.ieee.org/document/8399509/.
122. Liu, Q., et al. *Bidirectional-Convolutional LSTM Based Spectral-Spatial Feature Learning for Hyperspectral Image Classification*. arXiv e-prints, 2017. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170307910L.
123. Hinton, G.E., S. Osindero, and Y.W. Teh, *A Fast Learning Algorithm for Deep Belief Nets.* Neural Computation, 2006. **18**: p. 1527-1554 [cited 05.10.2019]; Available from: http://www.cs.toronto.edu/~fritz/absps/ncfast.pdf.
124. Bengio, Y., A. Courville, and P. Vincent *Representation Learning: A Review and New Perspectives*. arXiv e-prints, 2012. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2012arXiv1206.5538B.
125. Bengio, Y., et al., *Greedy Layer-Wise Training of Deep Networks*, in *Proceedings of the 19th International Conference on Neural Information Processing Systems*. 2006, MIT Press: Canada. p. 153-160. Available from: https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf.
126. Sze, V., et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. arXiv e-prints, 2017. [cited 05.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170309039S.
127. Merolla, P.A., et al., *A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface.* Science, 2014. **345**: p. 668-673 [cited 05.10.2019]; Available from: https://www.semanticscholar.org/paper/680a38e8f025685b192e9e0cf755c6b664963551?p2df.
128. Bansal, K., et al. *HOList: An Environment for Machine Learning of Higher-Order Theorem Proving*. arXiv e-prints, 2019. [cited 28.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190403241B.
129. Kaliszyk, C., F. Chollet, and C. Szegedy *HolStep: A Machine Learning Dataset for Higher-Order Logic Theorem Proving*. arXiv e-prints, 2017. [cited 28.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170300426K.
130. Rocca, G.L., *Knowledge Based Engineering: Between AI and CAD. Review of a Language Based Technology to Support Engineering Design.* Advanced Engineering Informatics, 2012. **26**(2): p. 159-179 [cited 28.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/S1474034612000092.
131. Vapnik, V.N. and A.Y. Chervonenkis, *Theory of Pattern Recognition in Russian*. 1974, USSR: Nauka [cited 06.10.2019]; Available from: https://www.semanticscholar.org/paper/Theory-of-pattern-recognition-(in-russian)-Vapnik-Chervonenkis/9c9ca61b3abd84710fa32691057cffd79204a71b.
132. Rüping, S., *SVM Kernels for Time Series Analysis*. 2001: Dortmund. Available from: http://hdl.handle.net/10419/77140.
133. Pan, B., Z. Shi, and X. Xu, *MugNet: Deep Learning for Hyperspectral Image Classification Using Limited Samples.* ISPRS Journal of Photogrammetry and Remote Sensing, 2018. **145**: p. 108-119 [cited 27.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/S0924271617303416.
134. Hao, S., et al., *A Deep Network Architecture for Super-Resolution-Aided Hyperspectral Image Classification With Classwise Loss.* IEEE Transactions on Geoscience and Remote Sensing, 2018. **56**(8): p. 4650-4663 [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/8390939.
135. Wang, W., et al. *Hyperspectral Image Classification Based on Capsule Network*. in *IGARSS 2018 - 2018 IEEE International Geoscience and Remote Sensing Symposium*. 2018. [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/8518951.
136. Haut, J.M., et al., *Low–High-Power Consumption Architectures for Deep-Learning Models Applied to Hyperspectral Image Classification.* IEEE Geoscience and Remote Sensing Letters, 2019. **16**(5): p. 776-780 [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/abstract/document/8554064.
137. He, N., et al., *Feature Extraction With Multiscale Covariance Maps for Hyperspectral Image Classification.* IEEE Transactions on Geoscience and Remote Sensing, 2019. **57**(2): p. 755-769 [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/document/8439081.
138. Olah, C., A. Mordvintsev, and L. Schubert, *Feature Visualization.* Distill, 2017 [cited 26.10.2019]; Available from: https://distill.pub/2017/feature-visualization.
139. Carter, S., et al., *Activation Atlas.* Distill, 2019 [cited 26.10.2019]; Available from: https://distill.pub/2019/activation-atlas.

140. Masood, F., I.-u.-H. Qazi, and K. Khurshid, *Saliency Based Visualization of Hyperspectral Satellite Images using Hierarchical Fusion.* Journal of Applied Remote Sensing, 2018. **12** [cited 27.10.2019]; Available from: https://www.researchgate.net/publication/328164366_Saliency_Based_Visualization_of_Hyperspectral_Satellite_Images_using_Hierarchical_Fusion.
141. Nagasubramanian, K., et al. *Explaining Hyperspectral Imaging Based Plant Disease Identification: 3D CNN and Saliency Maps*. arXiv e-prints, 2018. [cited 12.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180408831N.
142. Liang, J., et al. *Salient Object Detection in Hyperspectral Imagery*. in *2013 IEEE International Conference on Image Processing*. 2013. [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/document/6738493.
143. Zhang, X., et al., *Target Detection of Hyperspectral Image Based on Spectral Saliency.* IET Image Processing, 2019. **13**(2): p. 316-322 [cited 27.10.2019]; Available from: https://ieeexplore.ieee.org/document/8649993.
144. Jimenez, L.O. and D.A. Landgrebe, *Supervised Classification in High-Dimensional Space: Geometrical, Statistical, and Asymptotical Properties of Multivariate Data.* IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 1998. **28**(1): p. 39-54 [cited 15.10.2019]; Available from: https://ieeexplore.ieee.org/document/661089.
145. Sarhrouni, E., A. Hammouch, and D. Aboutajdine. *Band Selection and Classification of Hyperspectral Images Using Mutual Information: An Algorithm Based on Minimizing the Error Probability Using the Inequality of Fano*. in *2012 International Conference on Multimedia Computing and Systems*. 2012. [cited 04.08.2019]; Available from: http://ieeexplore.ieee.org/document/6320192/.
146. Gui, Y., et al., *Compressed Pseudo-Analog Transmission System for Remote Sensing Images over Bandwidth-Constrained Wireless Channels.* IEEE Transactions on Circuits and Systems for Video Technology, 2019. **PP**: p. 1-1 [cited 15.10.2019]; Available from: https://ieeexplore.ieee.org/document/8795548/.
147. International Organization for Standardization, *ISO 18381:2013: Space Data and Information Transfer Systems — Lossless Multispectral and Hyperspectral Image Compression*. 2013: Switzerland. Available from: https://www.iso.org/standard/62319.html.
148. The Consultative Committee for Space Data Systems, *Lossless Multispectral and Hyperspectral Image Compression*, in *Informational Report - CCSDS 120.2-G-1*. 2015. p. 99. Available from: https://public.ccsds.org/Pubs/120x2g1.pdf.
149. Li, C. and B. Wang. *Principal Components Analysis*. 2014 [cited 08.10.2019]; Available from: http://www.ccs.neu.edu/home/vip/teach/MLcourse/5_features_dimensions/lecture_notes/PCA/PCA.pdf.
150. Hlaváč, V. *Principal Component Analysis (PCA) - Application to images*. Image Processing 2018 [cited 21.10.2019]; Available from: http://people.ciirc.cvut.cz/~hlavac/TeachPresEn/11ImageProc/15PCA.pdf.
151. Jensen, A.R., *Chapter 8 - The Factor Structure of Reaction Time in Elementary Cognitive Tasks*, in *Clocking the Mind*, A.R. Jensen, Editor. 2006, Elsevier Science Ltd: Oxford. p. 137-153 [cited 08.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/B9780080449395500094.
152. Pearson, K., *LIII. On Lines and Planes of Closest Fit to Systems of Points in Space.* The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 1901. **2**(11): p. 559-572 [cited 07.10.2019]; Available from: https://doi.org/10.1080/14786440109462720.
153. Wold, S., K.H. Esbensen, and P. Geladi. *Principal Component Analysis*. 1987. [cited 07.10.2019]; Available from: https://www.sciencedirect.com/science/article/pii/0169743987800849.
154. Dunteman, G.H., *Principal Components Analysis*. 1989, Sage Publications: Newbury Park, California. Available from: https://methods.sagepub.com/book/principal-components-analysis.
155. Jolliffe, I., *Principal Component Analysis*. 2002, New York: Springer Verlag [cited 07.10.2019]; Available from: https://www.springer.com/de/book/9780387954424.
156. Ng, A. *Part XI - Principal Components Analysis*. CS229 Lecture notes 2019 [cited 08.10.2019]; Available from: http://cs229.stanford.edu/notes/cs229-notes10.pdf.
157. Sorzano, C.O.S., J. Vargas, and A.P. Montano *A Survey of Dimensionality Reduction Techniques*. arXiv e-prints, 2014. [cited 08.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1403.2877S.
158. Scholkopf, B., A. Smola, and K.-R. Müller. *Kernel Principal Component Analysis*. in *ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING*. 1999. MIT Press [cited 08.10.2019]; Available from: http://pca.narod.ru/scholkopf_kernel.pdf.
159. Pedregosa, F., et al. *Scikit-Learn: Machine Learning in Python*. arXiv e-prints, 2012. [cited 08.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2012arXiv1201.0490P.
160. Mather, P.M. and M. Koch, *Computer Processing of Remotely-Sensed Images: An Introduction.* 2011 [cited 26.10.2019]; Available from: http://www.knovel.com/knovel2/Toc.jsp?BookID=5033.
161. Rodarmel, C. and J. Shan, *Principal Component Analysis for Hyperspectral Image Classification.* Surv Land inf Syst, 2002. **62** [cited 26.10.2019]; Available from: https://www.researchgate.net/publication/265198128_Principal_Component_Analysis_for_Hyperspectral_Image_Classification.
162. Estornell, J., et al., *Principal Component Analysis Applied to Remote Sensing.* Modelling in Science Education and Learning, 2013. **6 (2)**: p. 83-89 [cited 26.10.2019]; Available from: https://www.researchgate.net/publication/259638444_Principal_component_analysis_applied_to_remote_sensing.
163. Genc, L. and S. Smith *Assessment of Principal Component Analysis (PCA) for Moderate and High Resolution Satellite Data*. 2016. 39 - 58; Available from: https://dergipark.org.tr/download/article-file/213715.

164. Zhu, C. and J. Yu, *Nonmetric Multidimensional Scaling Corrects for Population Structure in Association Mapping with Different Sample Types.* Genetics, 2009. **182**(3): p. 875-888 [cited 08.10.2019]; Available from: https://www.ncbi.nlm.nih.gov/pubmed/19414565.
165. Cox, T. and M. Cox, *Multidimensional Scaling, Second Edition*. 2000 [cited 26.10.2019]; Available from: https://www.researchgate.net/publication/256168803_Multidimensional_Scaling_Second_Edition.
166. Kumar Pal, A. *Multi-Dimension Scaling (MDS)*. Dimension Reduction 2018 [cited 08.10.2019]; Available from: https://blog.paperspace.com/dimension-reduction-with-multi-dimension-scaling/.
167. Abdi, H., *Singular Value Decomposition (SVD) and Generalized Singular Value Decomposition (GSVD).* Encyclopedia of Measurement and Statistics., 2007 [cited 26.10.2019]; Available from: https://personal.utdallas.edu/~herve/Abdi-SVD2007-pretty.pdf.
168. Sharma, P. *The Ultimate Guide to 12 Dimensionality Reduction Techniques*. 2018 [cited 08.10.2019]; Available from: https://www.analyticsvidhya.com/blog/2018/08/dimensionality-reduction-techniques-python/.
169. Comon, P., *Independent Component Analysis, a New Concept?* Signal Process., 1994. **36**(3): p. 287-314 [cited 08.10.2019]; Available from: https://hal.archives-ouvertes.fr/hal-00417283/document.
170. Draper, B.A., et al., *Recognizing Faces with PCA and ICA.* Comput. Vis. Image Underst., 2003. **91**(1-2): p. 115-137 [cited 08.10.2019]; Available from: https://www.sciencedirect.com/science/article/pii/S1077314203000778.
171. Lee, D.D. and H.S. Seung, *Algorithms for Non-Negative Matrix Factorization*, in *Proceedings of the 13th International Conference on Neural Information Processing Systems*. 2000, MIT Press: Denver, CO. p. 535-541. Available from: https://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf.
172. Lee, D. and H. Seung, *Learning the Parts of Objects by Non-Negative Matrix Factorization.* Nature, 1999. **401**: p. 788-91 [cited 08.10.2019]; Available from: https://www.researchgate.net/publication/12752937_Learning_the_Parts_of_Objects_by_Non-Negative_Matrix_Factorization.
173. scikit-learn. *Manifold Learning*. 2019 [cited 10.10.2019]; Available from: https://scikit-learn.org/stable/modules/manifold.html.
174. Roweis, S.T. and L.K. Saul, *Nonlinear Dimensionality Reduction by Locally Linear Embedding.* Science, 2000. **290**(5500): p. 2323-2326 [cited 10.10.2019]; Available from: https://science.sciencemag.org/content/sci/290/5500/2323.full.pdf.
175. Martinetz, T. and K. Schulten, *Topology Representing Networks.* Neural Networks, 1994. **7**(3): p. 507-522 [cited 09.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/0893608094901090.
176. Tenenbaum, J.B. and W.T. Freeman, *Separating Style and Content*, in *Advances in Neural Information Processing Systems 9*, M.C. Mozer, M.I. Jordan, and T. Petsche, Editors. 1997, MIT Press. p. 662-668 [cited 09.10.2019]; Available from: http://papers.nips.cc/paper/1290-separating-style-and-content.pdf.
177. Tenenbaum, J.B., V.d. Silva, and J.C. Langford, *A Global Geometric Framework for Nonlinear Dimensionality Reduction.* Science, 2000. **290**(5500): p. 2319-2323 [cited 10.10.2019]; Available from: https://science.sciencemag.org/content/sci/290/5500/2319.full.pdf.
178. Maaten, L.v.d. and G.E. Hinton. *Visualizing Data Using t-SNE*. 2008. [cited 10.10.2019]; Available from: http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf.
179. McInnes, L., J. Healy, and J. Melville *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. arXiv e-prints, 2018. [cited 29.08.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180203426M.
180. Bingham, E. and H. Mannila, *Random Projection in Dimensionality Reduction: Applications to Image and Text Data*, in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, ACM: San Francisco, California. p. 245-250. Available from: http://users.ics.aalto.fi/ella/publications/randproj_kdd.pdf.
181. Dong, Y. and N. Japkowicz. *Threaded Ensembles of Supervised and Unsupervised Neural Networks for Stream Learning*. 2016. Cham: Springer International Publishing [cited 09.11.2019]; Available from: https://link.springer.com/chapter/10.1007/978-3-319-34111-8_37.
182. Bourlard, H., *Auto-Association by Multilayer Perceptrons and Singular Value Decomposition*. 2000, IDIAP. Available from: http://publications.idiap.ch/downloads/reports/2000/rr00-16.pdf.
183. Chicco, D., P. Sadowski, and P. Baldi, *Deep Autoencoder Neural Networks for Gene Ontology Annotation Predictions.* ACM BCB 2014 - 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, 2014: p. 533-540 [cited 10.10.2019]; Available from: https://core.ac.uk/download/pdf/55252866.pdf.
184. Hinton, G.E. and R.R. Salakhutdinov, *Reducing the Dimensionality of Data with Neural Networks.* Science (New York, N.Y.), 2006. **313**: p. 504-7 [cited 10.10.2019]; Available from: https://www.cs.toronto.edu/~hinton/science.pdf.
185. Doersch, C. *Tutorial on Variational Autoencoders*. arXiv e-prints, 2016. [cited 10.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160605908D.
186. Makhzani, A., et al. *Adversarial Autoencoders*. arXiv e-prints, 2015. [cited 10.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151105644M.
187. Burda, Y., R. Grosse, and R. Salakhutdinov *Importance Weighted Autoencoders*. arXiv e-prints, 2015. [cited 10.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150900519B.
188. Makhzani, A. and B. Frey *k-Sparse Autoencoders*. arXiv e-prints, 2013. [cited 10.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2013arXiv1312.5663M.

189. Domingos, P., *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. 2018: Basic Books, Inc. 352 [cited 10.10.2019]; Available from: https://dl.acm.org/citation.cfm?id=3217330.
190. Zeng, N., et al., *Facial Expression Recognition via Learning Deep Sparse Autoencoders.* Neurocomputing, 2018. **273**: p. 643-649 [cited 10.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/S0925231217314649.
191. Vincent, P., et al., *Extracting and Composing Robust Features with Denoising Autoencoders*, in *Proceedings of the 25th international conference on Machine learning*. 2008, ACM: Helsinki, Finland. p. 1096-1103. Available from: https://www.researchgate.net/publication/221346269_Extracting_and_composing_robust_features_with_denoising_autoencoders.
192. Rifai, S., et al., *Contractive Auto-Encoders: Explicit Invariance During Feature Extraction*, in *Proceedings of the 28th International Conference on International Conference on Machine Learning*. 2011, Omnipress: Bellevue, Washington, USA. p. 833-840. Available from: https://icml.cc/2011/papers/455_icmlpaper.pdf.
193. Feng, F., et al., *Dimensionality Reduction of Hyperspectral Image with Graph-Based Discriminant Analysis Considering Spectral Similarity.* Remote Sensing, 2017. **9**: p. 323 [cited 15.10.2019]; Available from: https://www.researchgate.net/publication/315928445_Dimensionality_Reduction_of_Hyperspectral_Image_with_Graph-Based_Discriminant_Analysis_Considering_Spectral_Similarity.
194. Gewali, U., S. Monteiro, and E. Saber, *Machine Learning Based Hyperspectral Image Analysis: A Survey*. 2018 [cited 15.10.2019]; Available from: https://arxiv.org/abs/1802.08701.
195. Yan, X. and X.G. Su, *Linear Regression Analysis: Theory and Computing*. 2009: World Scientific Publishing Co., Inc. 348 [cited 14.10.2019]; Available from: https://www.researchgate.net/publication/267480780_Linear_Regression_Analysis_Theory_and_Computing.
196. Wikipedia. *Linear Regression*. 2019 [cited 14.10.2019]; Available from: https://en.wikipedia.org/wiki/Linear_regression.
197. Menard, S.W., *Applied Logistic Regression Analysis*. 2001: Sage Publications [cited 14.10.2019]; Available from: https://books.google.com/books/about/Applied_logistic_regression_analysis.html?id=zBgXAQAAMAAJ.
198. Gourieroux, C. and A. Monfort, *Asymptotic Properties of the Maximum Likelihood Estimator in Dichotomous Logit Models.* Journal of Econometrics, 1981. **17**(1): p. 83-97 [cited 10.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/0304407681900609.
199. Wikipedia. *Logistic Regression*. 2019 [cited 14.10.2019]; Available from: https://en.wikipedia.org/wiki/Logistic_regression.
200. Nelder, J.A. and R.W.M. Wedderburn, *Generalized Linear Models.* J. R. Stat. Soc. Ser. A, 1972. **19**: p. 92-100 [cited 10.11.2019]; Available from: https://www.researchgate.net/publication/287389446_Generalized_Linear_Models.
201. Czepiel, S. *Maximum Likelihood Estimation of Logistic Regression Models : Theory and Implementation*. [cited 14.10.2019]; Available from: https://czep.net/stat/mlelr.pdf.
202. Breiman, L., *Random Forests.* Machine Learning, 2001. **45**(1): p. 5-32 [cited 04.11.2019]; Available from: https://doi.org/10.1023/A:1010933404324.
203. Verikas, A., et al., *Electromyographic Patterns During Golf Swing: Activation Sequence Profiling and Prediction of Shot Effectiveness.* Sensors, 2016. **16**: p. 592 [cited 04.11.2019]; Available from: https://www.researchgate.net/publication/301638643_Electromyographic_Patterns_during_Golf_Swing_Activation_Sequence_Profiling_and_Prediction_of_Shot_Effectiveness.
204. Liaw, A. and M. Wiener, *Classification and Regression by RandomForest.* Forest, 2001. **23** [cited 14.10.2019]; Available from: https://www.researchgate.net/publication/228451484_Classification_and_Regression_by_RandomForest.
205. Breiman, L., *Bagging Predictors.* Machine Learning, 1996. **24**(2): p. 123-140 [cited 14.10.2019]; Available from: https://doi.org/10.1023/A:1018054314350.
206. scikit-learn. *sklearn.ensemble.RandomForestClassifier*. 2019 [cited 16.11.2019]; Available from: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.
207. Schapire, R., et al., *Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods*. Vol. 26. 1997. 322-330 [cited 14.10.2019]; Available from: https://www.researchgate.net/publication/221344849_Boosting_the_margin_A_new_explanation_for_the_effectiveness_of_voting_methods.
208. Song, L., P. Langfelder, and S. Horvath, *Random Generalized Linear Model: A Highly Accurate and Interpretable Ensemble Predictor.* BMC Bioinformatics, 2013. **14**(1): p. 5 [cited 14.10.2019]; Available from: https://doi.org/10.1186/1471-2105-14-5.
209. Lewis, N., *A Primer in Applied Regression Analysis*. 2005. p. 151-168 [cited 09.11.2019]; Available from: https://www.researchgate.net/publication/304728782_A_Primer_in_Applied_Regression_Analysis.
210. Shafrin, J. *What is a Pseudo R-squared?* 2016 [cited 14.10.2019]; Available from: https://www.healthcare-economist.com/2016/12/28/what-is-a-pseudo-r-squared/.
211. UCLA: Statistical Consulting Group. *FAQ: What Are Pseudo R-Squareds?* 2011 [cited 14.10.2019]; Available from: https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-what-are-pseudo-r-squareds/.
212. Halpin, B. *Pseudo-R2 is Pseudo*. 2016 [cited 14.10.2019]; Available from: http://teaching.sociology.ul.ie/bhalpin/wordpress/?p=365.
213. Shalizi, C. *Chapter 12 - Logistic Regression*. 2012 [cited 14.10.2019]; Available from: https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch12.pdf.

214. Morik, K. and U. Ligges. *Wissensentdeckung in Datenbanken*. 2013 [cited 14.10.2019]; Available from: https://www.statistik.uni-dortmund.de/fileadmin/user_upload/Lehrstuehle/Datenanalyse/Wissensentdeckung/Wissensentdeckung-Li-4_2x2.pdf.
215. Domencich, T. and D.L. McFadden, *Urban Travel Demand: A Behavioral Analysis*. 1975: North-Holland Publishing Co. [cited 14.10.2019]; Available from: https://eml.berkeley.edu/~mcfadden/travel.html.
216. Hensher, D.A. and P.R. Stopher. *Behavioural Travel Modelling*. 1979. [cited 14.10.2019]; Available from: https://books.google.com/books/about/Behavioural_Travel_Modelling.html?id=jqEOAAAAQAAJ.
217. Soheily-Khah, S. and Y. Wu, *A Novel Feature Engineering Framework in Digital Advertising Platform.* 2019. **10**: p. 21 [cited 02.11.2019]; Available from: https://www.researchgate.net/publication/334884247_A_Novel_Feature_Engineering_Framework_in_Digital_Advertising_Platform.
218. Pathak, A., M. Sehgal, and D. Christopher, *A Study on Fraud Detection Based on Data Mining Using Decision Tree.* International Journal of Computer Science Issues, 2011. **8** [cited 10.11.2019]; Available from: https://www.researchgate.net/publication/266052309_A_Study_on_Fraud_Detection_Based_on_Data_Mining_Using_Decision_Tree.
219. Cheng, Y., et al. *A Survey of Model Compression and Acceleration for Deep Neural Networks*. arXiv e-prints, 2017. [cited 15.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171009282C.
220. Nervana Systems. *Compressing Models - Pruning*. 2019 [cited 25.08.2019]; Available from: https://nervanasystems.github.io/distiller/pruning.html.
221. Setiono, R., *Extracting Rules from Pruned Neural Networks for Breast Cancer Diagnosis.* Artificial Intelligence in Medicine, 1996. **8**(1): p. 37-51 [cited 16.11.2019]; Available from: http://www.sciencedirect.com/science/article/pii/0933365795000194.
222. Cun, Y.L., J.S. Denker, and S.A. Solla, *Optimal Brain Damage*, in *Advances in neural information processing systems 2*, S.T. David, Editor. 1990, Morgan Kaufmann Publishers Inc. p. 598-605 [cited 18.10.2019]; Available from: http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf.
223. Hassibi, B., D.G. Stork, and G.J. Wolff. *Optimal Brain Surgeon: Extensions and Performance Comparisons*. in *NIPS 1993*. 1993. [cited 18.10.2019]; Available from: https://papers.nips.cc/paper/749-optimal-brain-surgeon-extensions-and-performance-comparisons.
224. Hanson, S. and L. Pratt, *Comparing Biases for Minimal Network Construction with Back-Propagation*. 1988. 177-185 [cited 18.10.2019]; Available from: https://papers.nips.cc/paper/156-comparing-biases-for-minimal-network-construction-with-back-propagation.
225. Gong, Y., et al. *Compressing Deep Convolutional Networks using Vector Quantization*. arXiv e-prints, 2014. [cited 19.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1412.6115G.
226. Han, S., et al., *Learning Both Weights and Connections for Efficient Neural Networks.* CoRR, 2015. **abs/1506.02626** [cited 01.08.2019]; Available from: http://arxiv.org/abs/1506.02626.
227. Molchanov, P., et al. *Pruning Convolutional Neural Networks for Resource Efficient Inference*. arXiv e-prints, 2016. [cited 18.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161106440M.
228. Li, H., et al. *Pruning Filters for Efficient ConvNets*. arXiv e-prints, 2016. [cited 18.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160808710L.
229. Zhu, M. and S. Gupta *To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression*. arXiv e-prints, 2017. [cited 25.08.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171001878Z.
230. Augasta, M.G. and T. Kathirvalavakumar, *Pruning Algorithms of Neural Networks — a Comparative Study.* Central European Journal of Computer Science, 2013. **3**: p. 105-115 [cited 18.10.2019]; Available from: https://link.springer.com/article/10.2478/s13537-013-0109-x.
231. Cheng, J., et al. *Recent Advances in Efficient Computation of Deep Convolutional Neural Networks*. arXiv e-prints, 2018. [cited 19.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180200939C.
232. Nervana Systems. *Weights Pruning Algorithms*. 2019 [cited 18.10.2019]; Available from: https://nervanasystems.github.io/distiller/algo_pruning.html.
233. Guo, Y., A. Yao, and Y. Chen *Dynamic Network Surgery for Efficient DNNs*. arXiv e-prints, 2016. [cited 18.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160804493G.
234. Narang, S., et al. *Exploring Sparsity in Recurrent Neural Networks*. arXiv e-prints, 2017. [cited 18.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170405119N.
235. Yu, D., et al. *Exploiting Sparseness in Deep Neural Networks for Large Vocabulary Speech Recognition*. in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012. [cited 18.10.2019]; Available from: https://ieeexplore.ieee.org/document/6288897.
236. Hu, H., et al. *Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures*. arXiv e-prints, 2016. [cited 18.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160703250H.
237. Brown, D.R. *Real-Time DSP Sampling, Quantization, Real-Time FIR Filtering*. ECE4703 2008 [cited 10.11.2019]; Available from: https://spinlab.wpi.edu/courses/ece4703_2008/lecture2.pdf.
238. Nervana Systems. *Quantization - Neural Network Distiller*. 2019 [cited 20.09.2019]; Available from: https://nervanasystems.github.io/distiller/quantization.html.

239. Dally, W. *High-Performance Hardware for Machine Learning*. Twenty-ninth Conference on Neural Information Processing Systems 2015 07.12.2015 [cited 21.10.2019]; Available from: https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.
240. Rastegari, M., et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. arXiv e-prints, 2016. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160305279R.
241. *IEEE Standard for Floating-Point Arithmetic.* IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019: p. 1-84.
242. Migacz, S. *8-bit Inference with TensorRT*. GPU Technology Conference 2017 2017 [cited 21.10.2019]; Available from: http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf.
243. Jacob, B., et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. arXiv e-prints, 2017. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171205877J.
244. Courbariaux, M., Y. Bengio, and J.-P. David *Training Deep Neural Networks with Low Precision Multiplications*. arXiv e-prints, 2014. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2014arXiv1412.7024C.
245. Google. *Building a Quantization Paradigm from First Principles*. gemmlowp: a small self-contained low-precision GEMM library 2019 [cited 21.10.2019]; Available from: https://github.com/google/gemmlowp/blob/master/doc/quantization.md.
246. Banner, R., et al. *Post-Training 4-Bit Quantization of Convolution Networks for Rapid-Deployment*. arXiv e-prints, 2018. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181005723B.
247. Krishnamoorthi, R., *Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper.* CoRR, 2018. **abs/1806.08342** [cited 29.08.2019]; Available from: http://arxiv.org/abs/1806.08342.
248. Seo, S. and J. Kim, *Efficient Weights Quantization of Convolutional Neural Networks Using Kernel Density Estimation Based Non-Uniform Quantizer.* Applied Sciences, 2019. **9**(12): p. 2559 [cited 21.10.2019]; Available from: https://www.mdpi.com/2076-3417/9/12/2559.
249. Choi, Y., M. El-Khamy, and J. Lee *Universal Deep Neural Network Compression*. arXiv e-prints, 2018. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180202271C.
250. Mishra, A. and D. Marr *Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy*. arXiv e-prints, 2017. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171105852M.
251. Bengio, Y., N. Léonard, and A. Courville *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. arXiv e-prints, 2013. [cited 23.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2013arXiv1308.3432B.
252. Wu, J., et al. *Quantized Convolutional Neural Networks for Mobile Devices*. arXiv e-prints, 2015. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151206473W.
253. Vanhoucke, V., A.W. Senior, and M.Z. Mao. *Improving the Speed of Neural Networks on CPUs*. 2011. [cited 21.10.2019]; Available from: https://ai.google/research/pubs/pub37631.
254. Gupta, S., et al. *Deep Learning with Limited Numerical Precision*. arXiv e-prints, 2015. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150202551G.
255. Zhou, S., et al. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. arXiv e-prints, 2016. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160606160Z.
256. Zhou, A., et al. *Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights*. arXiv e-prints, 2017. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170203044Z.
257. Mishra, A., et al. *WRPN: Wide Reduced-Precision Networks*. arXiv e-prints, 2017. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170901134M.
258. Choi, J., et al. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. arXiv e-prints, 2018. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180506085C.
259. Lin, X., C. Zhao, and W. Pan *Towards Accurate Binary Convolutional Neural Network*. arXiv e-prints, 2017. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171111294L.
260. Gysel, P., M. Motamedi, and S. Ghiasi *Hardware-Oriented Approximation of Convolutional Neural Networks*. arXiv e-prints, 2016. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160403168G.
261. Iandola, F.N., et al. *SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size*. arXiv e-prints, 2016. [cited 21.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160207360I.
262. Cao, X., et al., *Fast Hyperspectral Band Selection Based on Spatial Feature Extraction.* Journal of Real-Time Image Processing, 2018. **15**(3): p. 555-564 [cited 26.10.2019]; Available from: https://doi.org/10.1007/s11554-018-0777-9.
263. Medjahed, S.A. and M. Ouali, *Band Selection Based on Optimization Approach for Hyperspectral Image Classification.* The Egyptian Journal of Remote Sensing and Space Science, 2018. **21**(3): p. 413-418 [cited 27.10.2019]; Available from: http://www.sciencedirect.com/science/article/pii/S1110982317303101.
264. Qian, S.-E., *Dimensionality Reduction of Multidimensional Satellite Imagery.* SPIE Newsroom, 2011 [cited 26.10.2019]; Available from: https://pdfs.semanticscholar.org/0bab/4e4e7800c92e2a860c943ef1c209ed53d735.pdf.
265. Agarwal, A., et al. *Efficient Hierarchical-PCA Dimension Reduction for Hyperspectral Imagery*. in *2007 IEEE International Symposium on Signal Processing and Information Technology*. 2007. [cited 26.10.2019]; Available from: https://ieeexplore.ieee.org/document/4458191.

266. Koonsanit, K., C. Jaruskulchai, and A. Eiumnoh, *Band Selection for Dimension Reduction in Hyper Spectral Image Using Integrated InformationGain and Principal Components Analysis Technique.* International Journal of Machine Learning and Computing, 2012. **3**: p. 248-251 [cited 04.08.2019]; Available from: https://www.researchgate.net/publication/270772922_Band_Selection_for_Dimension_Reduction_in_Hyper_Spectral_Image_Using_Integrated_InformationGain_and_Principal_Components_Analysis_Technique.
267. Lin, Z., et al. *Spectral-Spatial Classification of Hyperspectral Image Using Autoencoders*. in *2013 9th International Conference on Information, Communications Signal Processing*. 2013. [cited 29.08.2019]; Available from: https://arxiv.org/abs/1511.02916v1.
268. Rajabi, R. and H. Ghassemian *Multilayer Structured NMF for Spectral Unmixing of Hyperspectral Images*. arXiv e-prints, 2015. [cited 26.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150601596R.
269. Jayaprakash, C., et al. *Randomized ICA and LDA Dimensionality Reduction Methods for Hyperspectral Image Classification*. arXiv e-prints, 2018. [cited 26.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180407347J.
270. Cheng, Q., P.K. Varshney, and M.K. Arora, *Logistic Regression for Feature Selection and Soft Classification of Remote Sensing Data.* IEEE Geoscience and Remote Sensing Letters, 2006. **3**(4): p. 491-494 [cited 26.10.2019]; Available from: https://ieeexplore.ieee.org/document/1715302.
271. Guo, Z., et al. *Semi-Supervised Hyperspectral Band Selection via Sparse Linear Regression and Hypergraph Models*. in *2013 IEEE International Geoscience and Remote Sensing Symposium - IGARSS*. 2013. [cited 26.10.2019]; Available from: https://ieeexplore.ieee.org/document/6723064.
272. Abdel-Rahman, E.M., F.B. Ahmed, and R. Ismail, *Random Forest Regression and Spectral Band Selection for Estimating Sugarcane Leaf Nitrogen Concentration Using EO-1 Hyperion Hyperspectral Data.* International Journal of Remote Sensing, 2013. **34**(2): p. 712-728 [cited 26.10.2019]; Available from: https://doi.org/10.1080/01431161.2012.713142.
273. He, Y. *A Curated List of Neural Network Pruning Resources*. Awesome Pruning 2019 [cited 25.10.2019]; Available from: https://github.com/he-y/Awesome-Pruning.
274. mrgloom. *Network Acceleration Methods*. 2019 [cited 25.10.2019]; Available from: https://github.com/mrgloom/Network-Speed-and-Compression.
275. chester256. *Papers for Deep Neural Network Compression and Acceleration*. 2018 [cited 25.10.2019]; Available from: https://github.com/chester256/Model-Compression-Papers.
276. *Compression Research Papers*. 2018 [cited 25.10.2019]; Available from: https://docs.google.com/spreadsheets/d/1n1xfLbt9QiqOZhSrIOdQNoOLLyozFdhCigdoXxF1cVk/edit#gid=627386401.
277. Choi, Y., M. El-Khamy, and J. Lee *Towards the Limit of Network Quantization*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161201543C.
278. Courbariaux, M., et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160202830C.
279. Courbariaux, M., Y. Bengio, and J.-P. David *BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151100363C.
280. Audebert, N., B. Le Saux, and S. Lefevre, *Deep Learning for Classification of Hyperspectral Data: A Comparative Review.* IEEE Geoscience and Remote Sensing Magazine, 2019. **7**(2): p. 159-173 [cited 01.08.2019]; Available from: https://ieeexplore.ieee.org/document/8738045.
281. Sharma, V.K., et al. *Hyperspectral CNN for Image Classification & Band Selection, with Application to Face Recognition*. 2016. [cited 02.08.2019]; Available from: https://pdfs.semanticscholar.org/a707/06f9f20a5b9c54fea99e02f93f14b2d87228.pdf.
282. Liu, B., et al., *A Semi-Supervised Convolutional Neural Network for Hyperspectral Image Classification.* Remote Sensing Letters, 2017. **8**(9): p. 839-848 [cited 02.08.2019]; Available from: https://doi.org/10.1080/2150704X.2017.1331053.
283. Boulch, A., N. Audebert, and D. Dubucq. *Autoencodeurs pour la visualisation d'images hyperspectrales*. in *XXV colloque Gretsi*. 2017. [cited 02.08.2019]; Available from: http://www.boulch.eu/files/2017_gretsi-autoencodeurs.pdf.
284. Santara, A., et al., *BASS Net: Band-Adaptive Spectral-Spatial Feature Learning Neural Network for Hyperspectral Image Classification.* IEEE Transactions on Geoscience and Remote Sensing, 2017. **55**(9): p. 5293-5301 [cited 02.08.2019]; Available from: http://ieeexplore.ieee.org/abstract/document/7938656/.
285. Cao, X., et al., *Hyperspectral Image Classification With Markov Random Fields and a Convolutional Neural Network.* IEEE Transactions on Image Processing, 2018. **27**(5): p. 2354-2367 [cited 02.08.2019]; Available from: https://ieeexplore.ieee.org/document/8271995.
286. Papers With Code. *Hyperspectral Image Classification on Indian Pines*. [cited 04.08.2019]; Available from: https://paperswithcode.com/sota/hyperspectral-image-classification-on-indian.
287. University of Tehran - Remote Sensing Laboratory. *Remote Sensing Datasets*. 2015 [cited 02.08.2019]; Available from: https://rslab.ut.ac.ir/data.
288. Computational Intelligence Group of Basque University. *Hyperspectral Remote Sensing Scenes*. 2014 [cited 04.08.2019]; Available from: http://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes.

289. Zar, J., *Biostatistical Analysis*. 1999, Prentice Hall: University of Michigan. p. 663 [cited 16.11.2019]; Available from: https://www.researchgate.net/publication/221959634_Biostatistical_analysis.
290. Du, R. *PyTorch Deep Compression*. 2018 [cited 02.08.2019]; Available from: https://github.com/larry0123du/PyTorch-Deep-Compression.
291. Chen, J., et al. *Self-Adaptive Network Pruning*. arXiv e-prints, 2019. [cited 31.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv191008906C.
292. Nervana Systems. *Neural Network Distiller by Intel AI Lab: A Python Package for Neural Network Compression Research*. 2019 [cited 04.08.2019]; Available from: https://github.com/NervanaSystems/distiller.
293. Nervana Systems. *Frequently Asked Questions (FAQ) - Quantization*. 2019 [cited 15.08.2019]; Available from: https://github.com/NervanaSystems/distiller/wiki/Frequently-Asked-Questions-(FAQ)#quantization.
294. Jacob, G. *Export Quantized Models with ONNX*. 2019 [cited 15.08.2019]; Available from: https://github.com/NervanaSystems/distiller/issues/255.
295. Masood, A. and A. Hashmi, *Cognitive Computing Recipes - Artificial Intelligence Solutions Using Microsoft Cognitive Services and TensorFlow*. 2019, USA: Apress [cited 09.11.2019]; Available from: https://link.springer.com/content/pdf/10.1007/978-1-4842-4106-6.pdf.
296. Yang, R. *ONNX v1.5.0*. 2019 [cited 20.08.2019]; Available from: https://github.com/onnx/onnx/releases.
297. Microsoft. *Convert ML models to ONNX with WinMLTools*. 2019 [cited 15.08.2019]; Available from: https://docs.microsoft.com/en-us/windows/ai/windows-ml/convert-model-winmltools#quantize-onnx-model.
298. Microsoft. *WinMLRunner Tool*. 2019 [cited 20.08.2019]; Available from: https://github.com/microsoft/Windows-Machine-Learning/tree/master/Tools/WinMLRunner.
299. TensorFlow. *Using the SavedModel Format*. 2019 [cited 02.11.2019]; Available from: https://www.tensorflow.org/guide/saved_model.
300. Zmora, N. *Thinning FC Layers*. 2018 [cited 13.08.2019]; Available from: https://github.com/NervanaSystems/distiller/issues/73.
301. Papers With Code. *HSI-CNN: A Novel Convolution Neural Network for Hyperspectral Image*. 2018 [cited 13.09.2019]; Available from: https://paperswithcode.com/paper/hsi-cnn-a-novel-convolution-neural-network.
302. PyTorch. *Quantization in Glow*. 2019 [cited 20.08.2019]; Available from: https://github.com/pytorch/glow/blob/master/docs/Quantization.md.
303. Rotem, N., et al., *Glow: Graph Lowering Compiler Techniques for Neural Networks.* CoRR, 2018. **abs/1805.00907** [cited 20.08.2019]; Available from: http://arxiv.org/abs/1805.00907.
304. Cao, X. *Hyperspectral Image Classification Using Markov Random Field and a Convolutional Neural Network*. 2018 [cited 12.09.2019]; Available from: https://github.com/xiangyongcao/CNN_HSIC_MRF.
305. Essig, M. *Pixelwise Classification with Keras - Hyperspectral Imaging - Indian Pines Dataset*. 2018 [cited 12.09.2019]; Available from: https://github.com/MeryllEssig/M2-DeepLearning.
306. Whetton, B. *Keras-Surgeon - Pruning and Other Network Surgery for Trained Keras Models*. 2018 [cited 21.09.2019]; Available from: https://github.com/BenWhetton/keras-surgeon.
307. TensorFlow. *Post-training weight quantization*. 2019 [cited 29.08.2019]; Available from: https://www.tensorflow.org/lite/performance/post_training_quant.
308. saketd403. *Visualising Image Classification Models and Saliency Maps*. 2018 [cited 12.09.2019]; Available from: https://github.com/saketd403/Visualising-Image-Classification-Models-and-Saliency-Maps.
309. Keras-vis. *visualize_saliency*. [cited 12.09.2019]; Available from: https://raghakot.github.io/keras-vis/vis.visualization/#visualize_saliency.
310. Anh, H.N. *Implementations of Some Popular Saliency Maps in Keras*. 2018 [cited 12.09.2019]; Available from: https://github.com/experiencor/deep-viz-keras.
311. Ming-Hsuan, Y. *Face Recognition Using Extended Isomap*. in *Proceedings. International Conference on Image Processing*. 2002. [cited 29.10.2019]; Available from: https://ieeexplore.ieee.org/document/1039901.
312. Chen, Y., et al., *Deep Learning-Based Classification of Hyperspectral Data.* Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of, 2014. **7**: p. 2094-2107 [cited 29.08.2019]; Available from: https://ieeexplore.ieee.org/document/6844831.
313. Zhan, Y., et al. *A New Hyperspectral Band Selection Approach Based on Convolutional Neural Network*. in *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. 2017. [cited 29.10.2019]; Available from: https://ieeexplore.ieee.org/document/8127792.
314. Hazelwood, K., et al. *Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective*. in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018. [cited 18.11.2019]; Available from: https://ieeexplore.ieee.org/document/8327042.
315. Ma, X., et al. *PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-Time Execution on Mobile Devices*. arXiv e-prints, 2019. [cited 07.11.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190905073M.
316. Anwar, S., K. Hwang, and W. Sung *Structured Pruning of Deep Convolutional Neural Networks*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151208571A.
317. Li, F., B. Zhang, and B. Liu *Ternary Weight Networks*. arXiv e-prints, 2016. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160504711L.
318. Zhu, C., et al. *Trained Ternary Quantization*. arXiv e-prints, 2016. [cited 20.09.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161201064Z.

319. TPcoding. *tflite Can't ResizeInputTensor Size*. 2018 02.07.2019 [cited 28.08.2019]; Available from: https://github.com/tensorflow/tensorflow/issues/22377.
320. Chaudhry, S. *Can't Allocate Memory for the Interpreter in tflite*. 2018 [cited 16.09.2019]; Available from: https://github.com/tensorflow/tensorflow/issues/19982.
321. TensorFlow. *Post-training integer quantization*. 2019 [cited 29.08.2019]; Available from: https://www.tensorflow.org/lite/performance/post_training_integer_quant.
322. Cai, Y., X. Liu, and Z. Cai *BS-Nets: An End-to-End Framework For Band Selection of Hyperspectral Image*. arXiv e-prints, 2019. [cited 07.11.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190408269C.
323. Tian, Q., T. Arbel, and J.J. Clark *Task-Specific Deep LDA Pruning of Neural Networks*. arXiv e-prints, 2018. [cited 07.11.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180308134T.
324. Ozbulak, U. *PyTorch Implementation of Convolutional Neural Network Visualization Techniques*. 2019 [cited 12.09.2019]; Available from: https://github.com/utkuozbulak/pytorch-cnn-visualizations.
325. Kim, B. *TensorFlow Tutorial for Various Deep Neural Network Visualization Techniques*. 2018 [cited 12.09.2019]; Available from: https://github.com/1202kbs/Understanding-NN.
326. *On Iterative Neural Network Pruning, Reinitialization, and the Similarity of Masks*. in *Submitted to International Conference on Learning Representations*. 2020. [cited 01.11.2019]; Available from: https://openreview.net/forum?id=B1xgQkrYwS.
327. Marcos, D., S. Lobry, and D. Tuia *Semantically Interpretable Activation Maps: What-Where-How Explanations Within CNNs*. arXiv e-prints, 2019. [cited 31.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190908442M.
328. He, H. *The State of Machine Learning Frameworks in 2019*. 2019 [cited 19.10.2019]; Available from: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.
329. Wen, W., et al. *Learning Structured Sparsity in Deep Neural Networks*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160803665W.
330. Liu, Z., et al. *Rethinking the Value of Network Pruning*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181005270L.
331. Wang, C., et al. *EigenDamage: Structured Pruning in the Kronecker-Factored Eigenbasis*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190505934W.
332. Liu, Z., et al., *Frequency-Domain Dynamic Pruning for Convolutional Neural Networks*, in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, Curran Associates Inc.: Montréal, Canada. p. 1051-1061. Available from: https://papers.nips.cc/paper/7382-frequency-domain-dynamic-pruning-for-convolutional-neural-networks.pdf.
333. Dong, X., S. Chen, and S. Jialin Pan *Learning to Prune Deep Neural Networks via Layer-Wise Optimal Brain Surgeon*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170507565D.
334. See, A., M.-T. Luong, and C.D. Manning *Compression of Neural Machine Translation Models via Pruning*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160609274S.
335. He, Y., et al. *Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181100250H.
336. Lin, S., et al. *Towards Optimal Structured CNN Pruning via Generative Adversarial Learning*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190309291L.
337. Ding, X., et al. *Centripetal SGD for Pruning Very Deep Convolutional Networks with Complicated Structure*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190403837D.
338. Lemaire, C., A. Achkar, and P.-M. Jodoin *Structured Pruning of Neural Networks with Budget-Aware Regularization*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181109332L.
339. Molchanov, P., et al. *Importance Estimation for Neural Network Pruning*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190610771M.
340. Zhao, C., et al. *Variational Convolutional Neural Network Pruning*. in *CVPR*. 2019. [cited 25.10.2019]; Available from: http://openaccess.thecvf.com/content_CVPR_2019/papers/Zhao_Variational_Convolutional_Neural_Network_Pruning_CVPR_2019_paper.pdf.
341. Gao, X., et al. *Dynamic Channel Pruning: Feature Boosting and Suppression*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181005331G.
342. Lee, N., T. Ajanthan, and P.H.S. Torr *SNIP: Single-Shot Network Pruning based on Connection Sensitivity*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181002340L.
343. Liu, L., et al. *Dynamic Sparse Graph for Efficient Deep Learning*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181000859L.
344. Ding, X., et al. *Approximated Oracle Filter Pruning for Destructive CNN Width Optimization*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190504748D.
345. Ye, J., et al. *Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180200124Y.

346. Zhuang, Z., et al. *Discrimination-Aware Channel Pruning for Deep Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181011809Z.
347. He, Y., et al. *AMC: AutoML for Model Compression and Acceleration on Mobile Devices*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180203494H.
348. Dubey, A., M. Chatterjee, and N. Ahuja *Coreset-Based Neural Network Compression*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180709810D.
349. Mallya, A. and S. Lazebnik *PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171105769M.
350. Yu, R., et al. *NISP: Pruning Networks Using Neuron Importance Score Propagation*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv171105908Y.
351. He, Y., et al. *Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180806866H.
352. Lin, J., et al., *Runtime Neural Pruning*, in *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, Curran Associates Inc.: Long Beach, California, USA. p. 2178-2188. Available from: https://papers.nips.cc/paper/6813-runtime-neural-pruning.pdf.
353. Yang, T.-J., Y.-H. Chen, and V. Sze *Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161105128Y.
354. He, Y., X. Zhang, and J. Sun *Channel Pruning for Accelerating Very Deep Neural Networks*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170706168H.
355. Yu, J., et al. *Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism*. in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017. [cited 25.10.2019]; Available from: https://ieeexplore.ieee.org/document/8192500.
356. Zhou, H., J.M. Alvarez, and F. Porikli. *Less Is More: Towards Compact CNNs*. 2016. Cham: Springer International Publishing [cited 25.10.2019]; Available from: https://link.springer.com/chapter/10.1007/978-3-319-46493-0_40.
357. Frankle, J. and M. Carbin *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180303635F.
358. Neklyudov, K., et al. *Structured Bayesian Pruning via Log-Normal Multiplicative Noise*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170507283N.
359. Lebedev, V. and V. Lempitsky *Fast ConvNets Using Group-Wise Brain Damage*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv150602515L.
360. Mehta, D., K. In Kim, and C. Theobalt *On Implicit Filter Level Sparsity in Convolutional Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv181112495M.
361. Li, J., et al. *OICSR: Out-In-Channel Sparsity Regularization for Compact Deep Neural Networks*. arXiv e-prints, 2019. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2019arXiv190511664L.
362. Liu, Z., et al. *Learning Efficient Convolutional Networks through Network Slimming*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170806519L.
363. Chen, C., et al. *Constraint-Aware Deep Neural Network Compression*. 2018. Cham: Springer International Publishing [cited 25.10.2019]; Available from: http://www.sfu.ca/~ftung/papers/constraintaware_eccv18.pdf.
364. Zhang, T., et al. *A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180403294Z.
365. Carreira-Perpinan, M.A. and Y. Idelbayev. *"Learning-Compression" Algorithms for Neural Net Pruning*. in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018. [cited 25.10.2019]; Available from: https://ieeexplore.ieee.org/document/8578988.
366. Aghasi, A., et al. *Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161105162A.
367. Peng, H., et al. *Collaborative Channel Pruning for Deep Networks*. in *ICML*. 2019. [cited 25.10.2019]; Available from: https://www.semanticscholar.org/paper/Collaborative-Channel-Pruning-for-Deep-Networks-Peng-Wu/f628234b1bec25cfc182eb959430692ce3a57b7e.
368. Huang, Z. and N. Wang *Data-Driven Sparse Structure Selection for Deep Neural Networks*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170701213H.
369. Luo, J.-H., J. Wu, and W. Lin *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170706342L.
370. Tung, F. and G. Mori, *CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization*. 2018. 7873-7882 [cited 25.10.2019]; Available from: https://www.researchgate.net/publication/329748916_CLIP-Q_Deep_Network_Compression_Learning_by_In-parallel_Pruning-Quantization.
371. Liu, X., et al. *Efficient Sparse-Winograd Convolutional Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180206367L.
372. Zhang, H., et al. *The ZipML Framework for Training Models with End-to-End Low Precision: The Cans, the Cannots, and a Little Bit of Deep Learning*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161105402Z.
373. Shin, S., K. Hwang, and W. Sung, *Fixed-Point Performance Analysis of Recurrent Neural Networks.* IEEE Signal Processing Magazine, 2015. **32**: p. 158-158 [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015ISPM...32..158.

374. Cai, Z., et al. *Deep Learning with Low Precision by Half-Wave Gaussian Quantization*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170200953C.
375. Ding, Y., et al. *On the Universal Approximability and Complexity Bounds of Quantized ReLU Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180203646D.
376. Dettmers, T. *8-Bit Approximations for Parallelism in Deep Learning*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151104561D.
377. Judd, P., et al. *Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151105236J.
378. Hashemi, S., et al. *Understanding the Impact of Precision Quantization on the Accuracy and Energy of Neural Networks*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161203940H.
379. Molchanov, D., A. Ashukha, and D. Vetrov *Variational Dropout Sparsifies Deep Neural Networks*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170105369M.
380. Ullrich, K., E. Meeds, and M. Welling *Soft Weight-Sharing for Neural Network Compression*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170204008U.
381. Louizos, C., K. Ullrich, and M. Welling *Bayesian Compression for Deep Learning*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170508665L.
382. Lin, J.-H., et al. *Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170704693L.
383. Yang, H., et al. *BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170509864Y.
384. He, Z., B. Gong, and D. Fan *Optimize Deep Convolutional Neural Network with Ternarized Weights and High Accuracy*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180707948H.
385. McDonnell, M.D. *Training Wide Residual Networks for Deployment Using a Single Bit for Each Weight*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180208530M.
386. Hou, L., Q. Yao, and J.T. Kwok *Loss-aware Binarization of Deep Networks*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161101600H.
387. Gudovskiy, D.A. and L. Rigazio *ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks*. arXiv e-prints, 2017. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2017arXiv170602393G.
388. Bagherinezhad, H., M. Rastegari, and A. Farhadi *LCNN: Lookup-Based Convolutional Neural Network*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv161106473B.
389. Hubara, I., et al. *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*. arXiv e-prints, 2016. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2016arXiv160907061H.
390. Wu, S., et al. *Training and Inference with Integers in Deep Neural Networks*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180204680W.
391. Polino, A., R. Pascanu, and D. Alistarh *Model Compression via Distillation and Quantization*. arXiv e-prints, 2018. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2018arXiv180205668P.
392. Kim, Y.-D., et al. *Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications*. arXiv e-prints, 2015. [cited 25.10.2019]; Available from: https://ui.adsabs.harvard.edu/abs/2015arXiv151106530K.

8. List of Figures

Figure 1: Hyperspectral reflectance profile illustration of Burger and Gowen [7]. Among the bands captured by the sensor at different wavelengths λ, we choose the band with the wavelength λi and observe the reflectance profile for the pixel at the position xi, yi. The image plane at λi translates into exactly one dot in the reflectance profile. The hyperspectral image cube is comprised of two spatial dimensions (X and Y) and one spectral dimension (λ) [7]. 8

Figure 2: Functional taxonomy of HSI analysis goals defined by Chang [36]. Classification is portrayed as the intersection of discrimination and detection. Together with quantification, it is a common aim of HSI analysis [7]. Target classification is more specific than detection because a detector "may not be able to classify the targets it detected" (e.g., an anomaly detector). On the same note, a target classifier which classifies vehicles as wheeled or tracked is not necessarily able to identify a jeep and a truck, which shows the difference in granularity between classification and identification [36]. 10

Figure 3: Mathematical illustration of how the output of a neuron propagated to the connected neuron in the subsequent layer is calculated. For the k-th neuron and $i = 1, \ldots, m$, this involves calculating the weighted sum vk of weights wki from input signals xi, adding the bias bk and using the activation function σ to obtain the output $yk = \sigma vk$. Figure adjusted from [42]. 12

Figure 4: Illustration of max and average pooling with their challenges. If we choose the wrong pooling type, the line, whose grayscale values we see in the matrices, ceases to be visible. Yu et al. propose a mixed pooling in their paper as an alternative to max and average pooling and one of many pooling implementation types, which is a "stochastic procedure which randomly employs the local max pooling and average pooling methods when training CNNs" [48, 49]. Figure combined from [48] and [49]. 13

Figure 5: Li et al.'s illustration of 2D (a) and 3D (b) convolution operations (left-hand side: before convolution; right-hand side: after convolution). The 2D convolutional kernel convolves the input so that spatial features are captured. In contrast, 3D cubes are used for 3D convolutional operations in an effort to extract spatiospectral features [51]. 14

Figure 6: Hyperspectral image cube for the Pavia University scene to illustrate the hyperspectral input patch for a CNN, depending on the architecture used, adjusted from [38]. 1D CNNs focus on spectral features with their 1D convolutional layers, so they are only given a one-dimensional hyperspectral pixel of a depth as an input, the size of which is equal to the $number_of_bands$ (this will translate to a two-dimensional tensor in PyTorch). Conversely, 2D spatial input patches for all bands of the combined size of $width \times height \times number_of_bands$ are used by 2D CNNs (for spatial feature recognition) and 3D CNNs (with the potential of spatiospectral exploration of the input patch with 3D convolutional kernels), which will both mean five-dimensional tensors in PyTorch as we have seen in DeepHyperX [38]. 15

Figure 7: Three processes used for the task of exploring a neural network architecture are shown in Wistuba et al.'s illustrations [76]. For reinforcement learning, the agent aims to maximize its reward, i.e. a high accuracy, by choosing a neural network, making an observation from the environment and adjusting accordingly. Surrogate models act to minimize a cost function (often least squares) so that a promising candidate is chosen, evaluated, and the surrogate updated, which is repeated until a convergence criterion is reached. In the evolutionary algorithm context, parents from a neural network population are selected, mutated and evaluated or dropped, depending on the performance reached [76]. 16

Figure 8: Side-by-side comparison of the gradient descent and SGD algorithms, adjusted from [42]. The fact that SGD operates on batches instead of the entire training dataset significantly improves the runtime, which is why it is more common to use [81]. Because of these different algorithm philosophies, the meaning of the N parameter varies. 17

Figure 9: Illustration of different visualization methods for activations heavily inspired by, but arranged differently than [46]. Having defined what we mean by fi and Ri, the forward pass and backpropagation should seem familiar (concept explained in 2.2.2). As new approaches, deconvnet only passes through positive values during the backward pass, while the backpropagation variant "guided backpropagation" additionally respects ReLU's zeros from the forward pass in the sense that they remain during the guided backpropagation process. For Springenberg et al., "guided backpropagation produces cleaner visualizations than the 'deconvnet' approach" [46]. 20

Figure 10: Structure of a GRU cell and mathematical expression, as depicted by Alom et al. [42]. The state of the GRU at the time t is represented by ht, which is calculated using the previous state $ht - 1$ and an auxiliary input xt. The other variables show the intermediate calculations, where σ stands for the activation function used [42]. 22

Figure 11: Disambiguation of AI-related terms by Sze et al. [126]. Any ANN (CNN, RNN) falls under deep learning, whereas SVM and k-NN are machine learning methods. Spiking computing shows that not all brain-inspired methods are neural networks, just like not all AI-related terms automatically fall under machine learning [126]. 22

Figure 12: Separation variants of data points with $X1$ and $X2$ components illustrated by James et al. [93]. Maximum-margin classifiers are sensitive to outliers like the blue dot added in the right picture. If we allow the outlier to be misclassified so that we arrive at a support vector classifier, the $2 - 1$-dimensional hyperplane (a line) would have a desirable larger margin, which shows the practical implications of the bias-variance tradeoff [93]. 23

Figure 13: Side-by-side comparison of average compressed data rates for segmented and full images in bits per sample for several instruments in both hyperspectral and multispectral scenarios using the image compression standards and techniques in the columns, where "[b]etter performance [i.e. fewer bits per sample necessary] is indicated in green" [148]. The authors explain their results in the following way: "For hyperspectral images, full image compression always performs as well as or better than segmented compression. For multispectral images, where segment size is selected to be larger, this performance penalty is less apparent, and in some cases segmentation even provides a small benefit." [148] 28

Figure 14. Visualization of LLE calculation steps by Roweis and Saul [174]. LLE does not care about calculating pairwise distances, but instead assumes that each data point Xi and its neighbors are located on a locally linear patch. This is because Xi is linearly reconstructed using its neighbors Xk and Xj and corresponding weights Wik and Wij. With the low-dimensional embedding vectors Yk and Yj, Xi can be mapped to Yi [174]. 32

Figure 15: Simplified illustration of random forest by Verikas et al. [203]. Multiple decision trees $treei$ structurally possibly different from one another might arrive at different outcomes ki for the input X, $i = 1, \dots, B$, but in the end, they are combined to k. This can be done using averaging for regression purposes, or with plurality voting for classification tasks [203]. 35

Figure 16: Own visual description of the iterative pruning procedure on a neuron-level basis heavily inspired by Molchanov et al.'s [227]. For each iteration, the pruning criteria determine the least valuable neurons, which are removed, followed by typically a few epochs of retraining [227]. 38

Figure 17: Migacz's illustration of saturation and its advantages in the process of quantization [242]. In this example, values below the negative threshold $-T$ are considered outliers and mapped to the lowest value -127 when using saturation. One chooses T for a higher precision of the targeted values, i.e. between $-T$ and $+T$, compared to the precision between $-max$ and $+max$, which comes at the cost of not expressing the magnitude of values between $-max$ and $-T$ as well as $+T$ and $+max$, which is the compromise entailed by saturation [242]. 42

Figure 18: Iterative quantization as performed by Seo and Kim and Choi et al. by retraining the codebook can be applied for convolutional filters in a neural network. Retraining the codebook is often done in papers since it "can provide substantial accuracy improvements at lower bitwidths" [247]. We see that two bits would suffice for the representation of floating-point numbers in this example [248, 249]. Figure from [248]. 42

Figure 19: Krishnamoorthi's comparison of the quantization types post-training quantization and training-aware quantization for the TensorFlow-API [247]. The TOCO converter transforms a `pb` graph file into a usually significantly smaller `tflite` file, which can be deployed on mobile platforms. Activation quantization requires calibration data so the activations can be calculated and analyzed for the upcoming quantization; training-aware quantization takes part in the training process [247]. 43

Figure 20: Table of models supported by DeepHyperX from the very beginning. The variety of architectures and ideas seen in the table allows us to compare the suitability of the models for the hyperspectral datasets when comparing metrics, especially accuracy metrics, later on, as well as the suitability of model compressions for different architectures. 51

Figure 21: Table of models which we added to DeepHyperX since these are the models that achieve the highest accuracies on the IndianPines dataset [286]. This suffices as a reason to include the models so that we can examine how suitable the ideas, architectures and hyperparameters are in practice when conducting the experiments. 51

Figure 22: Table of datasets supported by DeepHyperX. While IndianPines, PaviaC, PaviaU, KSC and Botswana were already integrated in the tool, we expanded it to support the datasets Salinas, SalinasA, Samson, JasperRidge-198, JasperRidge-224, Urban-162, Urban-210, China and USA as well by implementing adequate data loaders. The AVIRIS sensor captures a center wavelength range from 400 to 2500nm with 10nm width, just like the EO-1 satellite, while ROSIS' center spectral range is 430-860nm with 4nm width [287, 288]. The corrected versions of IndianPines, SalinasA and Salinas are used, which means that water absorption bands have been removed by [288]. 52

Figure 23: The pruning method outlined and illustrated by Han et al. consists of the three-step-pipeline in the picture [226]. Once connections are pruned, retraining is essential to avoid a dramatic loss of accuracy [226]. On the right-hand side, an example for a pruned network is shown. Once all connections to and from a neuron are pruned, we consider the neuron pruned as it is unreachable. 54

Figure 24: Model accuracies for the IndianPines dataset (n=6, CI=95%). We see the split of models performing reasonably well at around 80% OA vs. ones that do not (30% OA roughly). The size of the CI of hamida, li and mou disqualifies these models from the race, while he and cao perform the best with regard to accuracy. 59

Figure 25: RAM and VRAM usages of the models on the IndianPines dataset (n=6; respective VRAM usages remain constant among the runs; for RAM usage, CI=95%). While all models consume roughly the same amount of RAM, we do not observe a correlation between VRAM usage and any of the accuracy metrics in Figure 24, with luo and nn consuming the most VRAM. The roy model asked for too much VRAM to be executed. 59

Figure 26: Time consumptions of the models on the IndianPines dataset (n=6, CI=95%). We generally see a strong trend for the neural networks that the higher the training time, the higher the inference time and the other way around. There is no correlation between either time and any accuracy metric from Figure 24, though. 60

Figure 27: Model sizes on the IndianPines dataset (no variation, so no CIs). We see a strong correlation between model sizes and the VRAM usages from Figure 25. This makes sense because the tensors that are passed through the model are stored in VRAM so that the model can be executed on GPU instead of CPU for inference (because it is general knowledge that this is faster and the usual thing to do for neural networks, following appropriate experiments [66]). Moreover, the model sizes and time consumptions from Figure 26 also seem similar regarding the position of the data points, but that is just a general correlational trend that does come with exceptions, such as luo vs. sharma and santara vs. cao. 60

Figure 28: RAM and VRAM usages of the models for the PaviaC dataset (n=6; VRAM usages remain constant among the runs; for RAM usage, CI=95%). We witness the same pattern as in Figure 25 for VRAM consumption, which backs up our claim that the relative distributions of VRAM consumptions do not or only hardly vary across different datasets. The sharma model is not feasible to use for this dataset due to excessive runtime. 61

Figure 29: Time consumptions of the models for the KSC dataset (n=6, CI=95%). The same pattern as with the IndianPines dataset in Figure 26 can be observed, which shows us that it makes sense to think of training and inference times as parameters bound to the model rather than a dataset. 61

Figure 30: OAs of the santara model after applying the best-performing feature extraction techniques in terms of OA on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). PCA constantly scores OAs over 95% (works reliably), while with a growing number of components, the OAs for LLE and NMF grow until the dent at 110 remaining components. Even for just two components, PCA achieved 95.36% OA (not shown) and only at a single component did the OA disappoint (52.19%). LLE only works at all until 100 components. Almost all of the visible OAs lie above the reference value of 77%. .. 64

Figure 31: OAs of the santara model after applying feature selection techniques on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). All three techniques perform better when more components remain according to their roughly linear trends, but even then, none of them come close to the reference OA of 77%, with RandomForest and LogisticRegression crossing LinearRegression's downwards trend at 60 components. Beyond certain component numbers, the techniques performed too unreliably to justify including the values of, e.g., LinearRegression and RandomForest at 70 components and more (grave fluctuations between 5% and 20% OA). 65

Figure 32: OAs of the cao model after applying feature selection techniques on the IndianPines dataset (CIs not shown for the sake of visibility, n=6). The OAs vary a lot for linear regression, but this technique overall reaches the same OAs as random forest. Surprisingly, the more bands we select with logistic regression, the worse the OA gets, but the fact it surpasses the reference OA at 10 and 30 components we think is remarkable, although the high OAs come at the cost of an uncertainty we did not have for feature extraction techniques like PCA to this extent. 66

Figure 33: OA for the connection-wise threshold-based pruned cao model (n=6, CI=95%). Despite having a high base OA, the OA after pruning crosses the line at about 40% pruning percentage and the OAs we received at a lower pruning percentage are better than the OA of the original model. We see an approximately linear decrease until 77% and a sharp decline in OA from 93% pruning onwards. 68

Figure 34: OA for the connection-wise threshold-based pruned hu model (n=6, CI=95%). Just like in Figure 33, we can identify three phases: an OA similar to the reference model between 0% and 38%, a slight decline afterwards until around 90% and an unpredictable decline of OA in the end, which makes the usage of a hu model pruned to a percentage greater than 90% unreliable. 68

Figure 35: OA for the connection-wise threshold-based pruned santara model (n=6, CI=95%). The increasing steepness of the OA decline after 40% is most reminiscent of the trend outlined by Han et al. from the models we have pruned and analyzed. 69

Figure 36: OA for the connection-wise threshold-based pruned luo model (n=6, CI=95%). To highlight how much we could prune before an OA loss worth mentioning occurs, we start plotting the data points at 97% in this figure. Since the number of parameters is primarily caused by the linear layer having more than 2,500 times more parameters than the other two convolutional layers and the other linear layer combined, it is pleasant to see that an overwhelming majority of these parameters can be safely pruned from the network without any negative consequence to OA. This confirms the finding of Han et al. that linear layers can be pruned more than convolutional layers [226]. Between 0% and 97% pruning, most OAs surpassed the reference OA with few exceptions, but deviations in both ways are around 1% from the reference OA. 70

Figure 37: OA for the channel-wise pruned luo model using the $l1$-norm (n=6, CIs not shown for the sake of visibility). With wide ranges of reaching 45-50% OA visible, channel pruning is beneficial for a good OA until it begins to drop at 65% combined pruning of the two convolutional layers increasingly rapidly. The structure of the curve is much clearer than in the fine-grained pruning shown in Figure 33, Figure 34, Figure 35 and Figure 36. 72

Figure 38: OAs for the cao, he, hu, luo and santara models having been quantized after training, in this case, with the same number of bits for all quantizable model components (n=6, CI=95%). This figure highlights that it is a bad idea to quantize activations, weights and accumulators with the same number of bits as the highest OA we reach in this diagram is 16% (compared to much higher base accuracies as shown in Figure 39 - not shown here to keep these low accuracies visible), rendering all models unusable in practice. Contrasting the models, we see that cao suffers the least in terms of OA, although the differences are marginal. 74

Figure 39: OAs for the quantized cao, he, hu, luo and santara models (n=6, CI=95%). All five models have been quantized with 8 bits for activations and 16 bits for weights. The significance of the variation of bits used to quantize the accumulators is displayed in this figure, which teaches us that it is a bad idea to quantize the accumulators at all (the reference OA models use 32 bits for representing activations, weights and accumulators respectively), considering the enormous OA differences across the board. The combination of 8/16/32 bits for activations/weights/accumulators gets us the best accuracies for the whole quantization measurement, in most cases even surpassing the reference OAs. 74

Figure 40: OAs for the quantized cao, he, hu, luo and santara models (n=6, CI=95%). All five models have been quantized with 16 bits for the accumulators. This figure shows that should we decide to quantize the accumulators to 16 bits (e.g., because model size was especially important to us), the only three promising results we will get unless we use (32,32) for (activations, weights) are (4,4), (4,8) and (8,4). All other combinations for all analyzed models always produced accuracies about as low as the ones we see for (8,8), i.e. around 10%. Even for these three combinations, we still lose a lot of OA, so we do not advise to quantize the accumulators. 75

Figure 41: Relative model sizes remaining after post-training quantization of ONNX model files with WinMLTools. The greater the original model's size, the higher the number of parameters, the greater their impact on the calculated remaining model size fraction. It is a value pushed down by the quantized parameters for larger models, which becomes more apparent for 8 bits than it is visible for 16 bits. 76

Figure 42: OAs of the models by their fine-grained pruning percentages for five component numbers. PCA was used for feature extraction as a result of the conclusions of 4.2.2. Compared to the reference OA (no band selection technique, number of bands stays at the 200 for the IndianPines dataset), high accuracies are retained longer and the trend is visible much more clearly. Until around 60%, all accuracies of the PCA-X curves for $X \in 40,70,100,140,170$ are very close to or precisely 100%. After that, using more PCA components leads to a shallower accuracy descent. 79

Figure 43: Relative model size savings with regard to the number of components used for the band selection technique, with which the model was generated. hu's model size is expressed in four rays getting increasingly shallower (due to ceil functions determining the two output dimensions of Conv1d and MaxPool1d), sometimes surpassing its reference size, while the other models' sizes scale linearly with the number of components. The absolute reference model sizes are 1.14MB (he), 309.94KB (hu), 396.95MB (luo), 10.44MB (santara) and 4.56MB (cao). The luo model cannot be generated for 40 (and fewer) components because the kernel size of its second convolutional layer would have been greater than the actual input size, which is not allowed. The same reasoning holds true for he and 10 components. 80

Figure 44. OAs of the models depending on the fine-grained pruning percentages for the five component numbers with PCA as the band selection technique. **hu's reference OA is twice as worse as he's, which shows the potential of raising the OA by this pipeline more clearly**: until around 60% pruning, the mean OA of all PCA-X curves for $X \in 40,70,100,140,170$ is almost double the reference OA. Only then do the OAs begin to decline and the difference between the Reference-OA and the PCA-X curves begins to shrink. 81

Figure 45: OAs of the models for LinearRegression with 50 components followed by fine-grained pruning (best viewed in color). The accuracies of he in particular are all over the place, they do not surpass the reference OA at any time. While the luo OAs remain constant and we observe the same curve type for hu as in Figure 44 with PCA, the other accuracy curves only vaguely follow a recognizable pattern. This unpredictability / chaos combined with a low OA is characteristic for the feature selection algorithms analyzed, of which LinearRegression represents the trend we encountered well. 82

Figure 46: OAs for the NMF-70 - (fine-grained $l1$-based) pruning - (post-training) quantization pipeline for three (activations, weights, accumulators) bit combinations compared to the NMF-70-pruning reference OA curve. The values are often so close to each other that no matter which line style we chose, we would not be able to see all four values. That means that not much accuracy is lost here due to the added quantization, and the amounts of OA lost for the three curves mostly correspond to our findings from 4.2.3.3. The results for other component numbers in $X \in 40,70,100,140,170$ for NMF-X reflect the same findings, as does PCA-X for that matter. 83

Figure 47: Gradient-based saliency maps for 0 to 85% APoZ-based channel pruning for bands 0-90 in steps of 10 for PCA on the biases of the `dense_1` layer. The features at 0% are still very visible at 40%, which makes sense because until 40%, the OA can be maintained. The more we prune afterwards, the more distortion we start to witness: the position and value of the gradients progressively shifts. At 85%, the gradients have become particularly low. 85

Figure 48: Gradient-based saliency maps for 0 to 85% APoZ-based channel pruning for bands 0-90 in steps of 10 for NMF on the biases of the `dense_1` layer. In contrast to Figure 47, features remain even more visible at higher pruning percentages, i.e. the pixel maps do not change as much as in Figure 47 for more pruning – likely attributed to NMF's non-negativity constraint, as we think. Gradients at individual pixels get darker for a higher pruning percentage, often brightening up the surrounding area (e.g., bands 0 and 20 for 70 vs. 85% pruning). The input appears less bright than PCA's. 85

Figure 49: Activation maps for bands 0 to 90 in increments of 10 from left to right for any pruning percentage and layer generated with the `visualize_activation` method. We see that entirely black patches as the best model input for no band selection may or may not remain black when we use band selection. For example, bands 30, 40 and 60 seemed particularly relevant for NMF with 100 bands, whereas PCA shows additional interest in the bands 50, 70 and 90 (at least more obviously than NMF). 86

Figure 50: Saliency maps obtained through guided backpropagation for bands 0 to 90 in increments of 10 from left to right for no pruning. On occasion, the weights and biases for a chosen layer might look similar if we look at dots for very high gradients or line structures (e.g., the `dense_1` layer), but this very well might be coincidence. 87

Figure 51: Table summarizing the metrics measured for the compressions shown. Inference time measurements do not make sense in PyTorch, but considering Intel Distiller's lack of ONNX export for quantized models and Distiller being the only suite suitable for us to support fine-grained quantization, we could not export and convert the quantized model to any other language. However, we did implement our minimal Keras implementation to measure these metrics, but that is only possible for models whose structures are possible to express in Keras, which is very difficult for, e.g., tensor splitting. PyTorch's lack of physical model size reduction, on the other hand, causes inference times not to shrink after (fine-grained) pruning, so there is no point in conducting in-depth inference time measurements in this case. The color scheme green/yellow/orange denotes whether we could achieve results / experienced minor or circumventable difficulties / were stuck due to force majeure. 88

Figure 52: Number of articles about deep learning for HSI per year with regard to the application domains (RS = remote sensing), where "[t]he last column comprises published and in-press papers found up to 31 January 2019" [13]. We see an almost exponential increase of articles, which is particularly visible between 2016 and 2018, being mostly attributed to the popularity increase of remote sensing classification, but also due to new HSI applications like food and agriculture [13]. ... 89

Figure 53: Table showing the numbers of components as a subset of $A = 10,20, \dots ,190$ for which the achieved OA surpasses the respective reference OA; techniques that are not shown, but used in the experiment, never managed to surpass a reference OA, especially including LinearRegression and RandomForest, but also other exotic techniques. Note that LLE did not work for more than 100 components due to its component requirements. The color scheme green/yellow/orange denotes how satisfying the results are in terms of a high/mediocre/bad reference OA and variability of the band selection technique in terms of the choice of the number of components. 91

Figure 54: Visualization of the he model with hiddenlayer. The fact that tensors are split into four parts and added twice is characteristic for the model. 127

Figure 55: Visualization of the hu model with hiddenlayer. Judging from the architecture, this is our simplest and most linear model. 127

Figure 56: Visualization of the luo model with hiddenlayer. The nodes between "Shape" and "Concat" on the right-hand side are an extensive visualization for a reshape operation, which is key to the model. 127

Figure 57: Visualization of the cao model with hiddenlayer. Unlike the one-dimensional hu model, this is an easy example for a two-dimensional model. 128

Figure 58: Excerpt of the santara model visualization with hiddenlayer. After the Conv2d layer, tensors are split. They pass through different one-dimensional convolutional layers before they are concatenated and go through the linear layers in the end. The tensor splitting gimmick characteristic for this model makes the visualization complex. 128

Figure 59: Gradient-based saliency maps after applying NMF (left) / PCA (right) for the cao model's first four trainable weights on IndianPines. They generalize our observations from 4.3.1 by showing they are independent from the trainable weight used. The bands shown, for which the saliency maps were calculated, are bands 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 from left to right for both NMF-100 and PCA-100. Of the many hyperspectral patches (e.g., 4,500 samples) of the dimension `patch_size * patch_size * number_of_bands`, the first such patch was consistently chosen so the results could be compared. 129

Figure 60: Gradient-based saliency maps after applying NMF (left) / PCA (right) for the cao model's last five trainable weights on IndianPines except for the `dense_1`: biases (they have been shown already in Figure 47 and Figure 48). They generalize our observations from 4.3.1 by showing they are true regardless of the trainable weight used. 130

Figure 61: Visualizations of the IndianPines dataset. From left to right: ground truth, a typical training ground truth, a typical testing ground truth, an RGB visualization of the dataset using the bands (43,21,11). Training and testing ground truth add up to the leftmost picture, where the 80%-20% sampling of pixels is performed randomly. The colors of the first three pictures stand for the 16 IndianPines classes (plus the black color to represent unclassified territory). 130

Figure 62: Visdom visualization that shows the reflectances (y axis) of the 16 classes of IndianPines (i.e. without the "Unclassified" class) depending on the number of the band (x axis). The key is visible on the right-hand side. We see several peaks of reflectance for several class curves being close to one another at the band numbers 50, 60 and 120. The most difference in reflectance, i.e. a presumably high discriminative ability, which could be well used to distinguish between the classes in HSI classification tasks, can be found at low band numbers, especially the first 50 bands. 131

Figure 63: A confusion matrix provided by DeepHyperX's Visdom integration. The descriptions on the left depict the actual class of the pixel, while the labels on the bottom denote the class which a pixel is classified as. In this example, we see that the overwhelming majority of classifications are on the diagonal and only a few unclassified outliers ("Undefined") exist, which could be expressed in the form of good accuracy metrics if desired, although this compact representation would come at the cost of losing the potential of witnessing the distribution of false positives and false negatives, which we could

do in the matrix (including the calculation of precision and recall metrics as a result). In any case, many samples on the diagonal of the confusion matrix as shown here denote a high classification accuracy. 131

Figure 64: Visualization of the quantized `he.pb` model in TensorBoard. Seeing the nodes, future work could consider the calculations made by the nodes and analyze how the quantization of activations/weights/accumulators to a certain number of bits influences these on a low-level basis. 132

Figure 65: Architecture considerations for compression-related experiments in chapter 4.2. Core components include the hyperspectral models and datasets as well as band selection covered by DeepHyperX, `iterative_pruning.py` implementing fine-grained pruning, Intel Distiller allowing for coarse-grained pruning and component-specific quantization variants, as well as other quantization possibilities for the sake of calculating the model size reductions, seeing the nodes in a visualization and obtaining a model suitable to be used for an Android app, notwithstanding the considerable challenges linked with the computational restrictions of mobile devices. 133

Figure 66: Architecture considerations for visualization-related experiments in chapter 4.3. Once a hyperspectral framework is built in Keras and an appropriate model implemented, channel pruning like we have seen in Intel Distiller can be connected, but this time, with APoZ as the pruning criterion (because such a code fragment has been available and we have worked with it before). The fact that Keras is a high-level TensorFlow-API helps making the quantization process more straightforward than in PyTorch and a matter of few lines of code. Keras-vis offers a variety of inbuilt visualization methods one could use, but for more specific variations like in our case, one could consider own implementations. 133

Figure 67: Model accuracies for the PaviaC dataset (n=6, CI=95%). OA, AA and Kappa are so high in general that this dataset would have been a bad foundation for further experiments. 134

Figure 68: Model accuracies for the Botswana dataset (n=6, CI=95%). The extreme disparities between well and badly performing models discouraged us from choosing this dataset. The exorbitantly large CIs for hamida and luo are conspicuous. 134

Figure 69: Model accuracies for the Urban-210 dataset (n=6, CI=95%). OA is so much greater than AA, which in turn is greater than Kappa, that this fact that there must be classes difficult to classify discouraged us from choosing this dataset. 135

Figure 70: Model accuracies for the USA dataset (n=6, CI=95%). The phenomenon of Figure 69 is even more prevalent here, with OA amounting to more than triple the AA at times. The dataset being rather unknown to the scientific community also contributed to us not choosing this dataset. 135

Figure 71: Inference times for the cao model in the Keras implementation depending on the APoZ channel-based pruning percentage (n=**5** because sufficed for statistical significance, CIs not shown for visibility). The reference times (no prior band selection) surpass the ones after PCA / NMF by roughly 20%. The more is pruned, the closer the PCA and NMF times get. Regardless, we can clearly see that pruning helps to improve inference times, which motivates further research. 136

Figure 72: Inference times for the cao model in the Keras implementation after quantization depending on the APoZ channel-based pruning percentage (n=**5** because sufficed for statistical significance, CIs not shown for visibility). The inference times are about 20 times as high as in Figure 71, which may be attributed to the fact that the way we needed to do inference for the `tflite` model (`interpreter.invoke` in a loop for every patch) is not as efficient as the `model.predict` method for `h5` (which considers all patches and is not available for `tflite`). Regardless, we see linear a decrease for all three cases, where the PCA and NMF curves converge the more we prune. 136

Figure 73: Comparison of inference time savings for pruning alone vs. for pruning and post-training quantization depending on the APoZ channel-based pruning percentage (no prior band selection applied; n=**5** because sufficed for statistical significance, CIs not shown for visibility). Both curves show that the inference times decrease, but the pruning data is much more scattered, whereas the linear trend is clearly observable for quantization. We should also keep in mind that while these relative inference time savings sound nice, the absolute values shown in Figure 71 and Figure 72 reveal that the quantization inference times are about 20 times as high as the pruning inference times, as described for these figures. .. 137

Figure 74: Model sizes of the cao model implemented in Keras after pruning with optional quantization on top after applying (or not applying) the band selection techniques PCA or NMF, depending on the APoZ channel-based pruning percentage. Quantization does account for a huge model size reduction (3MB), whereas the curves do not react that strongly to pruning anymore if we decide to quantize. A band selection technique saves us space. All trends observed scale linearly with the pruning percentage of the model. 137

9. Appendix

9.1 Further Related Work

This is an overview of related work not listed in 2.3 and 3.3, categorized in pruning, quantization and their respective subtopics. It is not an exhaustive list; we have picked the papers as representatives for thinking outside the box, constituted by our experiment and the background required to understand it, as well as other thematically similar work.

9.1.1 Pruning

Weight Pruning

EigenDamage: Structured Pruning in the Kronecker-Factored Eigenbasis [331]
- *Problem*: reduce inference time
- *Action*: formulate network based on Kronecker-factored eigenbasis, followed by applying Hessian-based pruning with approximately independent weights
- *Result*: 10x model size reduction, 8x FLOPS reduction on wide ResNet32

Frequency-Domain Dynamic Pruning for Convolutional Neural Networks [332]
- *Problem*: exploit spatial correlations for network compression and acceleration
- *Action*: use dynamic pruning scheme for pruning frequency-domain coefficients iteratively: "frequency bands are pruned discriminatively, given their different importance on accuracy"
- *Result*: 8.4x compression ratio, 9.2x inference speedup for ResNet-110, better accuracy than reference model on CIFAR-10 dataset

Learning to Prune Deep Neural Networks via Layer-Wise Optimal Brain Surgeon [333]
- *Problem*: existing pruning methods need lots of retraining or do not compress much
- *Action*: propose a layer-wise pruning of parameters bounded by a linear combination of reconstructed errors at each layer
- *Result*: less retraining required to obtain well-performing neural network, as proven by experiments with LeNet-300-100 and LeNet-5 on MNIST, CIFAR-Net on CIFAR-10, AlexNet and VGG-16 on the ImageNet ILSVRC-2012 dataset

Dynamic Network Surgery for Efficient DNNs [233]
- *Problem*: structurally complex neural networks
- *Action*: perform **on-the-fly connection pruning with connection splicing**
- *Result*: LeNet-5 compressed by 108x, AlexNet by 17.7x, both on MNIST dataset

Compression of Neural Machine Translation Models via Pruning [334]
- *Problem*: overparameterization of neural networks, which means a large storage size
- *Action*: propose class-blind, class-uniform and class-distribution pruning schemes, which compute pruning thresholds differently
- *Result*: 40% pruning of neural machine translation model with 200 million parameters; can surpass original performance even with an 80%-pruned model

Filter Pruning

Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration [335]
- *Problem*: requirements for norm-based pruning, i.e. large norm deviation of filters and small minimum norm, are not always met
- *Action*: propose filter pruning method that does not need the two requirements above, where redundant filters are pruned, not the ones with less importance according to a low norm
- *Result*: 52% FLOPS on ResNet-110 on CIFAR-10 / 42% FLOPS on ResNet-101 on ILSVRC-2012 reduced while generally maintaining accuracy

Towards Optimal Structured CNN Pruning via Generative Adversarial Learning [336]

- *Problem*: the popular iterative pruning of layers may be computationally intensive and not yield optimal results
- *Action*: pruning approach that jointly prunes filters and other structures end-to-end
- *Result*: for example, ResNet-50 achieves 10.88% Top-5 error and results in 3.7x speedup on ILSVRC-2012

Centripetal SGD for Pruning Very Deep Convolutional Networks with Complicated Structure [337]

- *Problem*: redundancy of CNNs
- *Action*: **approximate filters** so that they become identical for network slimming, so that the redundant filters can be removed with no performance loss; use centripetal SGD (C-SGD) to train several filters at once and map them to a single parameter point
- *Result*: promising results on CIFAR-10 and ImageNet; evidence that redundant CNN trained with C-SGD performs better than normally trained counterpart with same width

Structured Pruning of Neural Networks with Budget-Aware Regularization [338]

- *Problem*: size and inference speed are not controlled directly, although these parameters are often the target of pruning methods
- *Action*: use **budgeted regularized pruning framework** based on learnable masking layer, a novel objective function and knowledge distillation
- *Result*: more accurate, less computationally expensive CNNs; accuracy drop incurred by severe pruning is prolonged

Importance Estimation for Neural Network Pruning [339]

- *Problem*: computation, energy and memory transfer costs of neural networks
- *Action*: **attribute loss contributions to neurons** using Taylor expansions (first and second-order), iteratively removing neurons with the smallest scores; no sensitivity analysis required, works for all layer types
- *Result*: >93% contribution correlation on ImageNet compared to reliable estimate; 40% FLOPS reduction, 30% fewer parameters on ResNet-101 with broadly the same accuracy

Variational Convolutional Neural Network Pruning [340]

- *Problem*: deterministic value-based pruning is improper and unstable
- *Action*: **Bayesian channel pruning** to estimate channel saliency (newly introduced parameter) for better computation efficiency; such a layer does not require special network design
- *Result*: 40% fewer channels of ResNet-50 on ImageNet dataset; up to 74% size reduction on CIFAR-10 for different models

Rethinking the Value of Network Pruning [330]

- *Problem*: misconceptions about the promises of pruning exist
- *Action*: evaluate common beliefs about pruning based on past research
- *Result*: observations: finetuning a pruned model is at most as good as training it with randomly initialized weights or using Frankle and Carbin's lottery ticket strategy; **applying pruning algorithms that are not network-independent is equivalent to training said architecture from scratch; the pruned architecture found is more important than a set of allegedly important weights**

Dynamic Channel Pruning: Feature Boosting and Suppression [341]

- *Problem*: running CNNs is computationally costly and requires lots of memory
- *Action*: salient convolutional channels can be predicted in advance and amplified to skip other unimportant input and output channels at runtime, using the feature boosting and suppression (FBS) technique; in contrast to pruning, **FBS does not change network structures, but accelerates convolution**
- *Result*: 5x computational savings for VGG-16, 2x for ResNet-18 on the ImageNet dataset

SNIP: Single-Shot Network Pruning Based on Connection Sensitivity [342]

- *Problem*: iterative pruning undermines the utility of large neural networks
- *Action*: prune network once prior to training with a connection sensitivity criterion, then train it normally
- *Result*: tiny architecture of CNNs, residual and recurrent neural networks for MNIST, CIFAR-10 and Tiny-ImageNet datasets

Dynamic Sparse Graph for Efficient Deep Learning [343]
- *Problem*: training and inference consume time and memory; research focuses only on inference
- *Action*: use dynamic and sparse graph structure, which activates only few neurons
- *Result*: 1.7-4.5x memory saving, 2.3-4.4x operation reduction on various benchmarks

Approximated Oracle Filter Pruning for Destructive CNN Width Optimization [344]
- *Problem*: neural networks are computationally expensive and design considerations are difficult; pruning takes a lot of time
- *Action*: approximated oracle filter pruning (AOFP) searches for least important filters using binary search, masks out filters randomly, accumulates errors and finetunes the network; simultaneous pruning possible with AOFP
- *Result*: can prune deep CNNs acceptably fast without heuristic knowledge or redesigning it

Rethinking the Smaller-Norm-Less-Informative Assumption in Channel Pruning of Convolution Layers [345]
- *Problem*: pruning reduces computational complexity, but assumption that small-norm parameter or feature is unimportant turns out to be problematic
- *Action*: new pruning technique simplifies channel-to-channel computation graph without needing to sparsify high-dimensional tensors with no apparent benefit: **use stochastic training to make output channels deliver constant values, remove these channels and adjust biases to make up for it**
- *Result*: competitive performance of this new benchmark

Discrimination-Aware Channel Pruning for Deep Neural Networks [346]
- *Problem*: training reduced network from scratch is computationally expensive, minimizing reconstruction error ignores discriminative power of channels
- *Action*: discrimination-aware channel pruning prunes only channels that contribute to discriminative power and **adds losses to layers**, which are used for selecting purposes together with the reconstruction error; iterative channel selection and parameter optimization algorithm proposed
- *Result*: channels of ResNet-50 can be pruned by 30% on ILSVRC-12 with better accuracy

AMC: AutoML for Model Compression and Acceleration on Mobile Devices [347]
- *Problem*: model compression techniques rely on hand-crafted heuristics and rule-based policies; domain experts need to determine tradeoff between size, speed and accuracy
- *Action*: **use reinforcement learning to automatically find model compression policy**
- *Result*: learning-based compression policy achieves better results than rule-based alternative: VGG-16 has 4x FLOPS reduction on ImageNet dataset, MobileNet experienced 1.81x inference time speedup on Android phone

Coreset-Based Neural Network Compression [348]
- *Problem*: iterative pruning requires retraining and is complicated to implement
- *Action*: use redundancies in coreset representations of filters for pruning; no retraining necessary, easy implementation
- *Result*: state-of-the-art (July 2018) compression performance for many CNN architectures; in combination with quantization and Huffman coding, 832x smaller memory footprint for AlexNet and inference time reductions

PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning [349]
- *Problem*: neural networks **trained for multiple tasks** using proxy losses result in forgetting a task
- *Action*: use neural network for multiple tasks by iterative pruning and network retraining; **learn new tasks with freed redundant parameters**
- *Result*: better robustness reached against forgetting; VGG-16 network trained for three tasks has accuracies comparable to the network trained for only one task

NISP: Pruning Networks Using Neuron Importance Score Propagation [350]

- *Problem*: existing pruning methods ignore error propagation in deep network (i.e. among multiple layers, not just the subsequent layer)
- *Action*: add a final response layer as the second-to-last layer for classification, based on which to optimize the binary integer optimization problem of pruning, using feature ranking techniques; **neuron importance score propagation propagates importance scores of final responses to every neuron**, according to which one-shot pruning followed by finetuning can be applied
- *Result*: significant acceleration and compression

Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks [351]

- *Problem*: existing filter pruning methods lack a large model capacity and are dependent on the pretrained model
- *Action*: soft filter pruning (SFP) allows filter update after pruning, which means a larger optimization space and enables SFP to train from scratch and prune simultaneously
- *Result*: 42% FLOPS reduced for ResNet-101 on ILSCRC-2012 dataset

Runtime Neural Pruning [352]

- *Problem*: existing pruning methods produce fixed model for deployment
- *Action*: runtime neural pruning **prunes neural network dynamically at runtime** in bottom-up, layer-by-layer approach modeled **as a Markov decision process with reinforcement learning** for training: agent judges importance of convolutional kernels and conducts channel-wise pruning
- *Result*: better tradeoff between speed and accuracy reached, especially with large pruning rates for simple pictures

Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning [353]

- *Problem*: CNNs consume a lot of energy because they are computationally complex
- *Action*: energy-aware layer-by-layer weight pruning algorithm prunes and retrains, whereafter finetuning with backpropagation takes place
- *Result*: AlexNet consumes 3.7x less energy, GoogLeNet 1.6x on ImageNet ILSVRC 2014 dataset

Channel Pruning for Accelerating Very Deep Neural Networks [354]

- *Problem*: neural networks are complex, which makes inference slow
- *Action*: propose iterative two-step pruning algorithm with LASSO channel selection
- *Result*: 5x faster inference for VGG-16, 2x for ResNet and Xception respectively

Other

Partial Order Pruning: For Best Speed/Accuracy Trade-Off in Neural Architecture Search [77]

- *Problem*: most architecture search approaches only want high performance, not focusing on good accuracy; need better tradeoff
- *Action*: partial order pruning uses partial order assumption to search for accuracies with best speed/accuracy tradeoff; propose Dongfeng (DF) networks that excel in both criteria
- *Result*: state-of-the-art (March 2019) speed/accuracy tradeoff of the proposed DF real-time segmentation networks

Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism [355]

- *Problem*: neural networks are energy-consuming; sparsity due to weight pruning can hurt overall performance and encoding can incur storage overhead
- *Action*: **customize pruning to underlying hardware** by matching pruned structure to data-parallel hardware organization; **depending on the parallelism potential of the hardware, different pruning types (weight and node pruning) are realized together**, but one type may be applied more than the other type
- *Result*: roughly twofold speedup compared to traditional weight pruning while significantly reducing model sizes

9.1.1.1 Sparsity Constraints and Regularization

These works train compact CNNs with sparsity constraints or focus on sparsity regularization.

Weight Pruning

Less Is More: Towards Compact CNNs [356]

- *Problem*: millions of CNN parameters
- *Action*: use **sparse constraints during training** in the objective function to reduce number of neurons and memory footprint
- *Result*: compact CNN contains 30% of original neurons

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks [357]

- Problem: identify subnetwork of network to be trained from scratch as opposed to iterative pruning
- Action: lottery ticket hypothesis: "dense, randomly-initialized, feed-forward networks contain **subnetworks** (winning tickets) that—when trained in isolation— **reach test accuracy comparable to the original network** in a similar number of iterations"
- Result: saved 80-90% storage for CIFAR-10 and MNIST respectively

Structured Bayesian Pruning via Log-Normal Multiplicative Noise [358]

- *Problem*: sparsity induced by Bayesian dropout is unstructured
- *Action*: proposed Bayesian model takes computational structure into account, injects noise into neurons while keeping weights unregularized; an example of Bayesian pruning, i.e. **probabilistic finetuning** of parameters like network weights
- *Result*: acceleration on numerous architectures

Filter Pruning

Fast ConvNets Using Group-Wise Brain Damage [359]

- *Problem*: want to speedup convolutional layers
- *Action*: perform group-wise filter pruning; reduce convolutions to matrix multiplications
- *Result*: speedups reached using group-sparsity regularizers

Learning Structured Sparsity in Deep Neural Networks [329]

- *Problem*: computational complexity of deep neural networks
- *Action*: regularize structures using structured sparsity learning to either learn a compact structure from a bigger ANN to reduce computation cost, or accelerate ANN evaluation by obtaining hardware-friendly structured sparsity, or regularize ANN structure for improved classification accuracy
- *Result*: 3.1x speedup of AlexNet's convolutional layer computations; 20 layers of ResNet on CIFAR-10 can be reduced to 18

On Implicit Filter Level Sparsity in Convolutional Neural Networks [360]

- *Problem*: no deep insight into filter level sparsity
- *Action*: investigate filter level sparsity for CNNs with batch normalization, ReLU, adaptive gradient descent and l_2 regularization or weight decay; analyze training strategies
- *Result*: practical training strategy suggestions; insight into how sparsity can be used for neural network speedup and filter level sparsity in general

OICSR: Out-In-Channel Sparsity Regularization for Compact Deep Neural Networks [361]

- *Problem*: correlations between consecutive layers are omitted by many channel pruning works
- *Action*: view subsequent layer (including statistics and correlations) together using Out-In-Channel Sparsity Regularization, resulting in the design of a **global greedy channel pruning algorithm**
- *Result*: reduced FLOPS by 37.2% for ResNet-50 on ImageNet-1K

Learning Efficient Convolutional Networks Through Network Slimming [362]

- *Problem*: high computational cost of CNNs; need to reduce model size, memory usage and number of operations
- *Action*: lower all three metrics simultaneously by enforcing channel-level sparsity during training
- *Result*: 20x model size reduction and 5x fewer computing operations for VGGNet on CIFAR-10

Structured Pruning of Deep Convolutional Neural Networks [316]

- *Problem*: pruning results in irregular network connections, hindering representation and efficient parallel computation
- *Action*: use channel-wise per-kernel and intra kernel strided structured sparsity using **particle filtering**: importance of particle for convolutional filter is assigned by computing misclassification rate with corresponding connectivity pattern
- *Result*: intra kernel strided sparsity can reduce size of kernel and feature map tensors; together with quantization on top, storage of neural network is significantly reduced

9.1.1.2 Pruning as an Optimization Problem

These works view pruning as an optimization problem and propose different strategies to deal with it.

Weight Pruning

Constraint-Aware Deep Neural Network Compression [363]

- *Problem*: applications require certain performance constraints, e.g., fast inference for self-driving cars. Need to compress neural network with respect to the constraint
- *Action*: stated compression learning problem as constrained Bayesian optimization problem, introduced solving strategy (“cooling” / “annealing”)
- *Result*: successful experiments on the ImageNet dataset

A Systematic DNN Weight Pruning Framework Using Alternating Direction Method of Multipliers [364]

- *Problem*: iterative pruning lacks guarantees on weight reduction ratio and convergence time
- *Action*: state weight pruning as optimization problem, use **alternating direction method of multipliers** for systematic weight pruning to split the problem into two, solve using SGD / analytically respectively
- *Result*: fast convergence rate; weight reduction without accuracy loss: 71.2x for LeNet-5 on MNIST, 21x for AlexNet on ImageNet; five times less computation in convolutional layers

“Learning-Compression” Algorithms for Neural Net Pruning [365]

- *Problem*: compress large neural networks to run them on mobile devices
- *Action*: propose a generic algorithm to solve pruning optimization problem by **alternating learning and compression phases**
- *Result*: “state-of-the-art” (June 2018) pruning in LeNet and ResNet variations

Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee [366]

- *Problem*: reduce overfitting and model redundancy
- *Action*: layer-wise pruning by solving optimization problem, providing parallel and cascade algorithm versions, including mathematical analysis
- *Result*: “[f]or a single layer taking independent Gaussian random vectors of length N as inputs” and at most s non-zero weights per node describing the network response, “**weights can be learned from $\mathcal{O}(s * \log(N))$ samples**”

Filter Pruning

Collaborative Channel Pruning for Deep Networks [367]

- *Problem*: computational overhead of neural networks
- *Action*: **collaborative** channel pruning exploits **inter-channel dependency** to prune channels; channel selection problem is stated as an optimization problem
- *Result*: higher classification accuracy than other state-of-the-art (September 2019) pruning methods

Data-Driven Sparse Structure Selection for Deep Neural Networks [368]

- *Problem*: computational complexity of neural networks
- *Action*: learn and prune models end-to-end based on scaling factor, sparsity regulations and solving an optimization problem using the accelerated proximal gradient method; can prune CNN by forcing some factors to be zero; only need one training pass (as opposed to iterative pruning)
- *Result*: promising results with adaptive depth and width selection

ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression [369]

- *Problem*: existing filter-level pruning methods derive filter importance from current layer statistics
- *Action*: accelerate and compress CNNs in both training and inference stages; view filter pruning as an optimization problem, **where filters are pruned based on statistics of next layer, not current one**
- *Result*: 3.31x FLOPS reduction and 16.63x compression for VGG-16 on ILSVRC-12 dataset; 50% reduction of FLOPS and model size for ResNet-50 on the same dataset

9.1.1.3 Combinations of Compressions

Analogous to Han et al.'s deep compression pipeline [12], pruning can be combined with other compression techniques to compress the model even more:

CLIP-Q: Deep Network Compression Learning by In-Parallel Pruning-Quantization [370]

- *Problem*: realize a more efficient utilization of computation resources and recover from premature pruning errors
- *Action*: jointly **perform pruning and weight quantization in parallel** with finetuning
- *Result*: compressions of AlexNet by 51-fold, GoogleLeNet by 10-fold, ResNet-50 by 15-fold with same accuracy as the respective uncompressed network

Efficient Sparse-Winograd Convolutional Neural Networks [371]

- *Problem*: neural networks are computationally intensive and not energy-efficient; multiplications for convolutions dominate energy usage
- *Action*: use combination of weight pruning and filtering by applying Winograd transform: move ReLU activations into Winograd domain and (aggressively) prune weights in Winograd domain for the sake of sparsity
- *Result*: 10.4x / 6.8x / 10.8x fewer multiplications on CIFAR-10, CIFAR-100 and ImageNet datasets

9.1.2 Quantization

We will split the references into the categories of general quantization, binarized neural networks (i.e. 1-bit quantizations), operation modifications and combinations of compressions.

The ZipML Framework for Training Models with End-To-End Low Precision: The Cans, the Cannots, and a Little Bit of Deep Learning [372]

- *Problem*: **can we guarantee metrics for models trained end-to-end with low precision, e.g., order-of-magnitude speedups**?
- *Action*: propose double sampling framework, which allows for low precision training with no bias; develop variance-optimal stochastic quantization strategy
- *Result*: FPGA prototype is 6.5x faster with framework compared to FP32; stochastic quantization saves 1.7x data movement; **question is true for linear models, and for non-linear models, bias can be controlled**

Fixed-Point Performance Analysis of Recurrent Neural Networks [373]

- *Problem*: RNNs implementations are complex; need to analyze fixed-point performance
- *Action*: use iterative quantization method, studying sensitivity in each layer; get the fixed-point optimization results
- *Result*: insight into fixed-point performance optimization of RNNs

Deep Learning with Low Precision by Half-Wave Gaussian Quantization [374]

- *Problem*: **need to approximate non-linearity** of binary quantization, especially tanh and ReLU
- *Action*: **half-wave Gaussian quantizer** is used for approximation, exploiting network statistics; use multiple piece-wise backward approximators against gradient mismatch
- *Result*: resulting network with 1-bit binary weights and 2-bit quantized activations shows closer performance to full precision networks than other low-precision networks

On the Universal Approximability and Complexity Bounds of Quantized ReLU Neural Networks [375]

- *Problem*: want to study representational power of quantized neural networks from the standpoint of complexity theory
- *Action*: provide upper bounds on the number of weights and their bitwidth and memory size for a given approximation error bound
- *Result*: **for an approximation error bound of ε, one needs no more than $\mathcal{O}\left(\log^5(1/\varepsilon)\right)$ times the weights of an unquantized network in the quantized network**

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [243]

- *Problem*: computational cost of neural networks
- *Action*: use integer-only quantization; perform training which preserves end-to-end model accuracy post quantization
- *Result*: improved tradeoff between accuracy and on-device latency

SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size [261]

- *Problem*: smaller CNNs can be deployed on mobile devices
- *Action*: design well-performing small CNN architecture based on AlexNet, but not as computationally complex
- *Result*: SqueezeNet has 50x fewer parameters than AlexNet and can be compressed to less than 0.5MB with model compression techniques (510x smaller than AlexNet)

8-Bit Approximations for Parallelism in Deep Learning [376]

- *Problem*: bottlenecks in communication bandwidth complicate speedup through parallelism
- *Action*: approximate by compressing 32-bit gradients and non-linear activations to 8 bits respectively
- *Result*: **2x data transfer speedup** compared to 32-bit parallelism: can obtain 50x speedup on 96 GPUs compared to 23x for 32 bits

Hardware-Oriented Approximation of Convolutional Neural Networks [260]

- *Problem*: high computational complexity of CNNs
- *Action*: hardware-oriented model approximation with Ristretto framework: analyzes CNN's numerical resolution for weights and outputs of convolutional and fully connected layers; can use fixed-point arithmetic instead of floating-point and finetune the network
- *Result*: can condense CaffeNet and SqueezeNet to 8-bit

Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets [377]

- *Problem*: want to investigate effect of reduced precision on classification accuracy: **layer precision tuning can save energy and improve performance**
- *Action*: **different layers use different precisions**: propose method for finding a low precision configuration for the network with acceptably high accuracy
- *Result*: precision may vary within a network; results show that data footprint for intermediate data of the used CNNs can be reduced by 74%

DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients [255]

- *Problem*: want to train neural networks with low-bitwidth parameters
- *Action*: propose DoReFa-Net to train low-bitwidth weights and activations with low-bitwidth gradients, which are stochastically quantized during backpropagation; convolutions can operate on low bitwidth kernels and activations, so DoReFa-Net can accelerate both training and inference
- *Result*: DoReFa-Net AlexNet on ImageNet can get comparable accuracy than original with 1-bit weights, 2-bit activations and 6-bit gradients

Understanding the Impact of Precision Quantization on the Accuracy and Energy of Neural Networks [378]
- *Problem*: **complexity and power consumption** of CNNs
- *Action*: perform a study of different bit-precisions in neural networks and the effect of precision scaling on accuracy and hardware metrics
- *Result*: **decreases of network accuracy are modest compared to design benefits gained by limited bit precision**

Variational Dropout Sparsifies Deep Neural Networks [379]
- *Problem*: want to investigate variational dropout
- *Action*: propose extension for unbounded dropout rates to reduce variance of gradient estimator, which leads to very sparse results
- *Result*: 280x fewer parameters for LeNet architectures, 68x for VGG-like networks

Soft Weight-Sharing for Neural Network Compression [380]
- *Problem*: neural networks are computationally expensive, requiring lots of memory and energy
- *Action*: use soft weight-sharing variant to achieve both quantization and retraining in one retraining procedure
- *Result*: competitive compression rates

Bayesian Compression for Deep Learning [381]
- *Problem*: make neural networks computationally efficient through compression
- *Action*: use Bayesian strategy for pruning, i.e. sparsity inducing priors; in particular, use hierarchical priors for neuron pruning and determine optimal fixed-point precision with posterior uncertainties
- *Result*: the two methods contribute greatly to good compression, speed and energy efficiency

9.1.2.1 Binarized Neural Networks

Binarization is a special case of quantization where one decides to represent some parameter (e.g., weights, activations, accumulators) with just 1-bit. We will include ternary quantization (i.e. the representation with 2 bits) here as this method is also targeted towards using very few bits for parameter representation. Perhaps counterintuitively, researchers have shown that even binarization can result in acceptable accuracies (and, obviously, compression rates), putting forward their respective ways of dealing with this constraint:

Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration [382]
- *Problem*: CNNs are costly in terms of computation and memory usage
- *Action*: **use binarized CNN with separable filters, which applies SVD**
- *Result*: binarized CNN with separable filters accelerator saves 17% memory and reduces execution time by 31.3%

BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet [383]
- *Problem*: improve the efficiency and lower the energy consumption of CNNs
- *Action*: implement BMXNet library, which supports both XNOR-Networks and quantized neural networks; layers can be seamlessly applied with other library components
- *Result*: efficiency and effectiveness of the library implementation is validated in experiments

Optimize Deep Convolutional Neural Network with Ternarized Weights and High Accuracy [384]
- *Problem*: model size and computational cost of deep CNNs
- *Action*: statistical weight scaling and residual expansion methods
- *Result*: 16x model compression of ternarized ResNet-32/44/56 on CIFAR-10 dataset; when adding residual expansion on top, 8x model compression rate

Trained Ternary Quantization [318]
- *Problem*: high energy consumption of deep neural networks, difficult to deploy on mobile devices
- *Action*: use trained ternary quantization
- *Result*: 16x smaller than full-precision models

Training Wide Residual Networks for Deployment Using a Single Bit for Each Weight [385]

- *Problem*: fast and energy-efficient deployment of deep neural networks on resource-constrained embedded hardware
- *Action*: apply scaling factors equal to layer-specific standard deviations
- *Result*: only 10MB parameter memory for 1-bit weight models on CIFAR-10, CIFAR-100 and ImageNet

Loss-Aware Binarization of Deep Networks [386]

- *Problem*: existing network binarizations ignore effect of binarization on the loss for matrix approximations
- *Action*: **use Newton algorithm with diagonal Hessian approximation** to minimize the loss with regard to binarized weights
- *Result*: binarization scheme outperforms existing binarization methods; it is more robust for wide and deep networks

9.1.2.2 Operation Modifications

Multiplication operations, many of which are contained in convolutional layers, have turned out to be computationally costly operations, so there is work dedicated to replacing them with other operations [387, 388] or using a lower precision variant [244, 389, 390]:

Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations [389]

- *Problem*: reduce power consumption of quantized neural networks
- *Action*: **train** very low precision networks, e.g., binary neural networks **at runtime**, using the training to compute gradients; replace arithmetic operations with bit-wise operations; propose binary matrix multiplication GPU kernel to run the MNIST QNN 7 times faster
- *Result*: 1-bit weights and 2-bit activations for AlexNet result in 51% accuracy on the ImageNet dataset; quantized RNNs used 4-bits for gradients instead of 32-bits, achieving comparable accuracy on the Penn Treebank dataset

ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks [387]

- *Problem*: multiplication operations are costly, need to reduce computational cost
- *Action*: propose ShiftCNN architectures, **performing only shifts and additions and precomputing convolutions**; number of multiplications is decreased by at least two orders of magnitude
- *Result*: 4x smaller power consumption of converted state-of-the-art (June 2017) CNNs on ImageNet than conventional 8-bit fixed-point architectures

Training and Inference with Integers in Deep Neural Networks [390]

- *Problem*: training and inference for low-bitwidth integers has not been demonstrated simultaneously
- *Action*: **discretize training and inference by shifting weights, activations, gradients and errors** and linearly constraining these parameters to low-bitwidth integers; **simplify arduous components for integer implementation**
- *Result*: integer-based deep learning accelerators and neuromorphic chips have higher energy efficiency

LCNN: Lookup-Based Convolutional Neural Network [388]

- *Problem*: difficult to deploy neural networks on computationally constrained platforms
- *Action*: **encode convolutions by few dictionary lookups** to make CNN compact
- *Result*: LCNN offers 3.2x speedup on ImageNet, up to 37.6x speedup possible

Training Deep Neural Networks with Low Precision Multiplications [244]

- *Problem*: multiplications are computationally expensive operators
- *Action*: perform comparison of floating point, fixed point and dynamic fixed point to assess impact of multiplication precision
- *Result*: very low precision is sufficient for both inference and training of neural networks

9.1.2.3 Combinations of Compressions

Just like pruning (9.1.1.3), quantization can also be combined with other model compression techniques to have an even greater effect on the compression magnitude:

Model Compression via Distillation and Quantization [391]

- *Problem*: want to use neural networks in resource-constrained environments
- *Action*: jointly leverage weight quantization and distillation: use proposed methods of **quantized distillation and differentiable quantization using "teacher" and "student" networks**
- *Result*: compression of up to an order of magnitude as well as almost linear inference speedup are possible

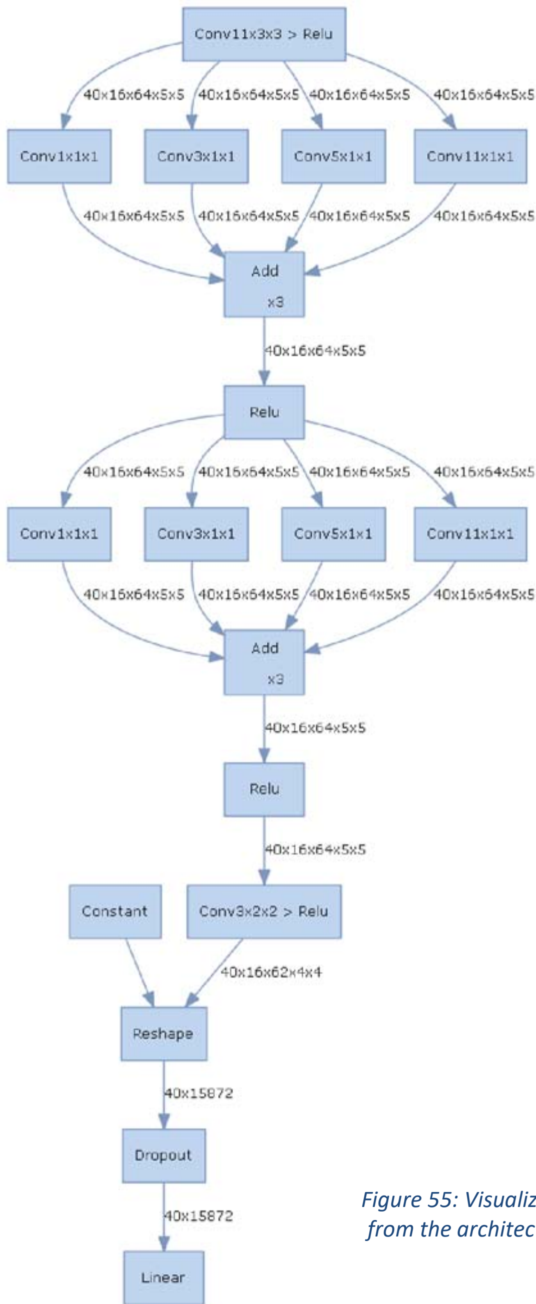
Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy [250]

- *Problem*: neural networks are compute and memory intensive
- *Action*: combine low-precision numerics and knowledge distillation
- *Result*: can compress to ternary precision and 4-bit precision for ResNet variants on the ImageNet dataset

Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications [392]

- *Problem*: running CNNs on mobile devices is challenging
- *Action*: one-shot whole network compression: rank selection with variational Bayesian matrix factorization, Tucker decomposition of kernel tensor and finetuning
- *Result*: significant model size, runtime and energy consumption reductions

9.2 Model Structures



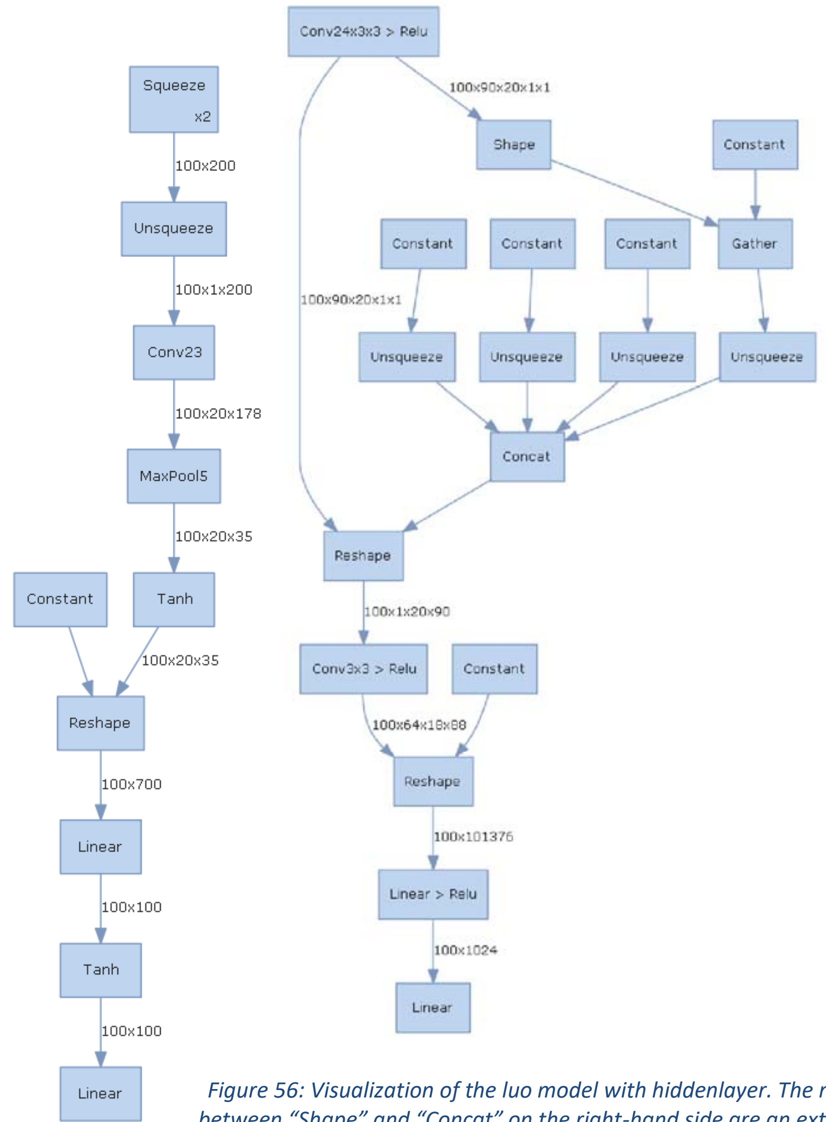
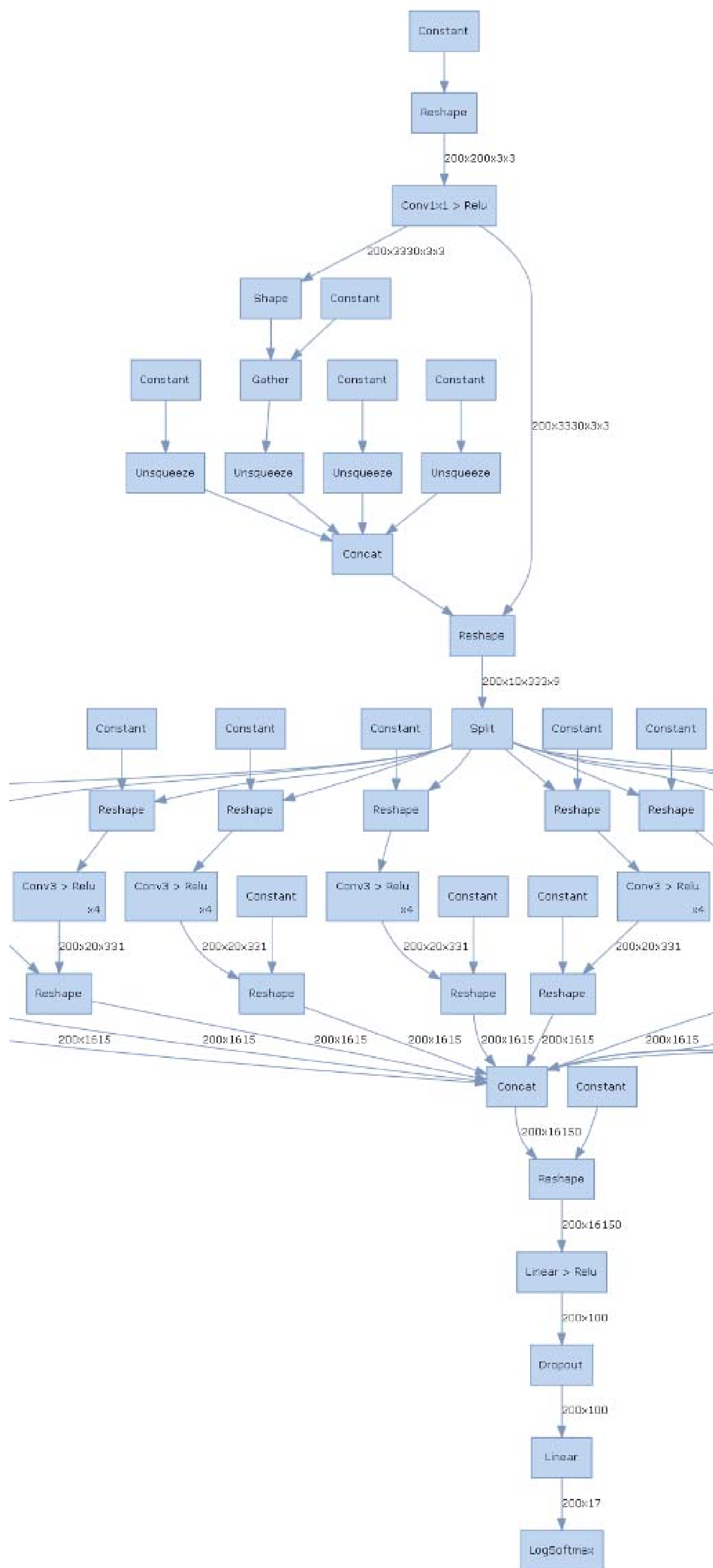


Figure 56: Visualization of the luo model with hiddenlayer. The nodes between “Shape” and “Concat” on the right-hand side are an extensive visualization for a reshape operation, which is key to the model.

Figure 55: Visualization of the hu model with hiddenlayer. Judging from the architecture, this is our simplest and most linear model.

Figure 54: Visualization of the he model with hiddenlayer. The fact that tensors are split into four parts and added twice is characteristic for the model.



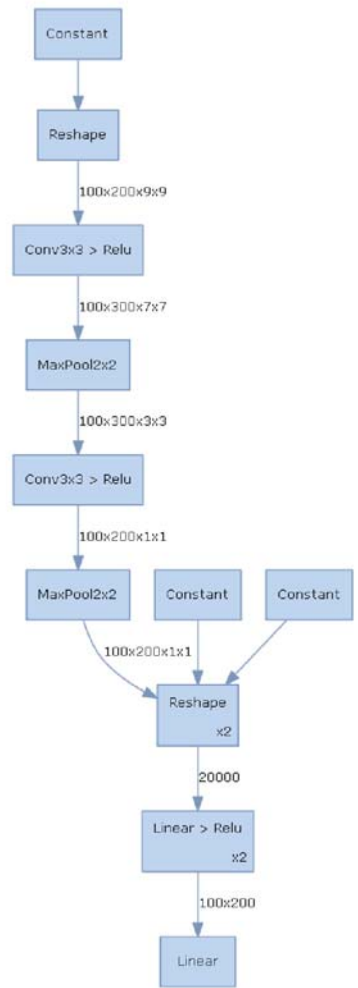


Figure 57: Visualization of the cao model with hiddenlayer. Unlike the one-dimensional hu model, this is an easy example for a two-dimensional model.

Figure 58: Excerpt of the santara model visualization with hiddenlayer. After the Conv2d layer, tensors are split. They pass through different one-dimensional convolutional layers before they are concatenated and go through the linear layers in the end. The tensor splitting gimmick characteristic for this model makes the visualization complex.

9.3 Visualizations

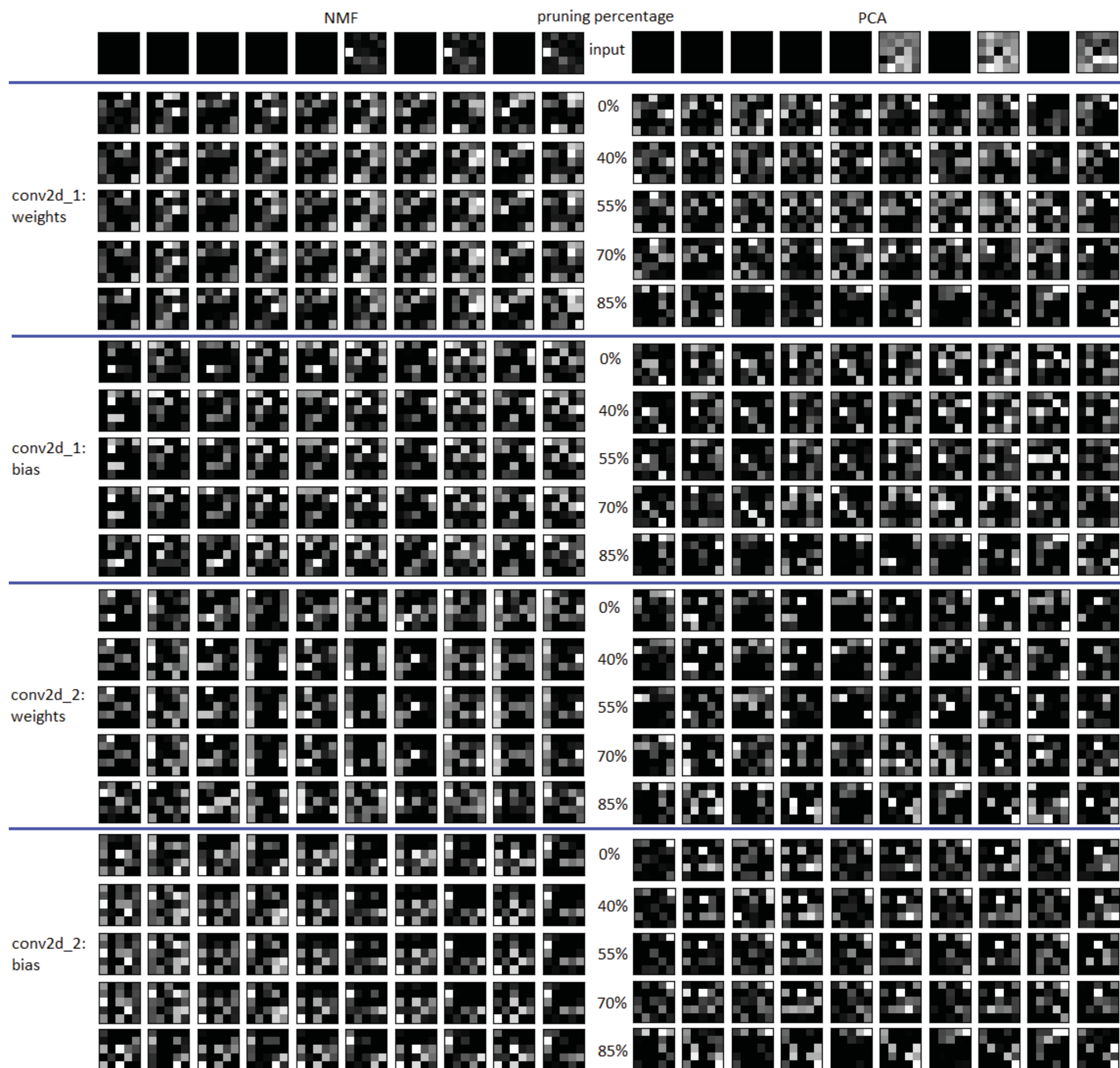


Figure 59: Gradient-based saliency maps after applying NMF (left) / PCA (right) for the cao model's first four trainable weights on IndianPines. They generalize our observations from 4.3.1 by showing they are independent from the trainable weight used. The bands shown, for which the saliency maps were calculated, are bands 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 from left to right for both NMF-100 and PCA-100. Of the many hyperspectral patches (e.g., 4,500 samples) of the dimension `patch_size * patch_size * number_of_bands`, the first such patch was consistently chosen so the results could be compared.

Figure 60: Gradient-based saliency maps after applying NMF (left) / PCA (right) for the cao model's last five trainable weights on IndianPines except for the `dense_1`: biases (they have been shown already in Figure 47 and Figure 48). They generalize our observations from 4.3.1 by showing they are true regardless of the trainable weight used.

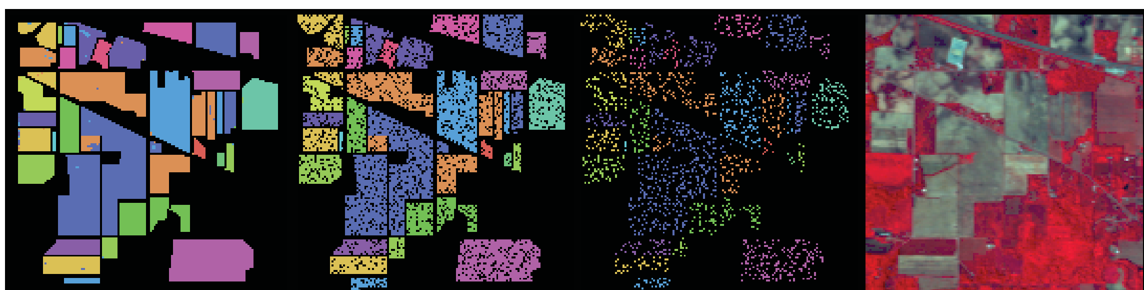


Figure 61: Visualizations of the IndianPines dataset. From left to right: ground truth, a typical training ground truth, a typical testing ground truth, an RGB visualization of the dataset using the bands (43,21,11). Training and testing ground truth add up to the leftmost picture, where the 80%-20% sampling of pixels is performed randomly. The colors of the first three pictures stand for the 16 IndianPines classes (plus the black color to represent unclassified territory).

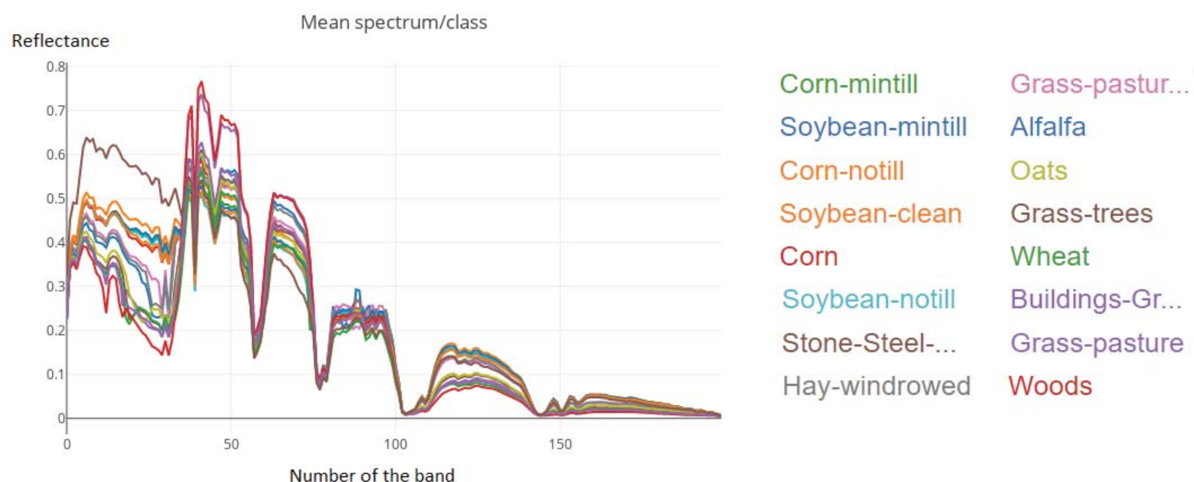


Figure 62: Visdom visualization that shows the reflectances (y axis) of the 16 classes of IndianPines (i.e. without the "Unclassified" class) depending on the number of the band (x axis). The key is visible on the right-hand side. We see several peaks of reflectance for several class curves being close to one another at the band numbers 50, 60 and 120. The most difference in reflectance, i.e. a presumably high discriminative ability, which could be well used to distinguish between the classes in HSI classification tasks, can be found at low band numbers, especially the first 50 bands.

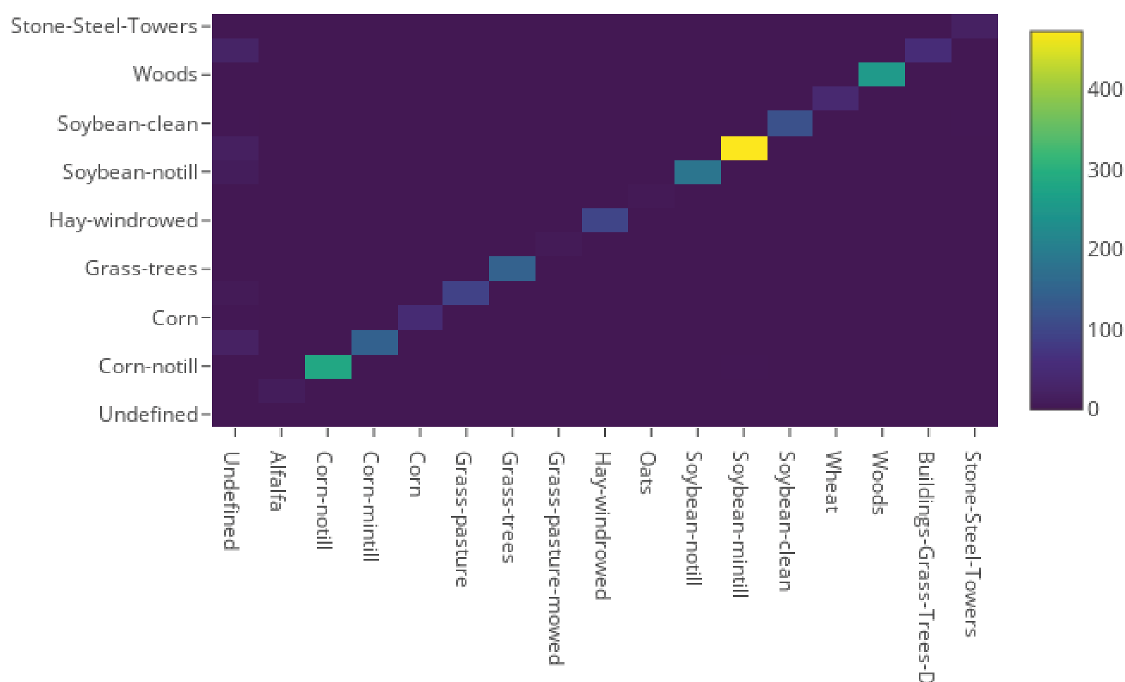


Figure 63: A confusion matrix provided by DeepHyperX's Visdom integration. The descriptions on the left depict the actual class of the pixel, while the labels on the bottom denote the class which a pixel is classified as. In this example, we see that the overwhelming majority of classifications are on the diagonal and only a few unclassified outliers ("Undefined") exist, which could be expressed in the form of good accuracy metrics if desired, although this compact representation would come at the cost of losing the potential of witnessing the distribution of false positives and false negatives, which we could do in the matrix (including the calculation of precision and recall metrics as a result). In any case, many samples on the diagonal of the confusion matrix as shown here denote a high classification accuracy.

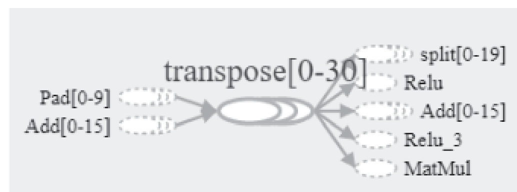


Figure 64: Visualization of the quantized `he.pb` model in TensorBoard. Seeing the nodes, future work could consider the calculations made by the nodes and analyze how the quantization of activations/weights/accumulators to a certain number of bits influences these on a low-level basis.

9.4 Architecture Considerations

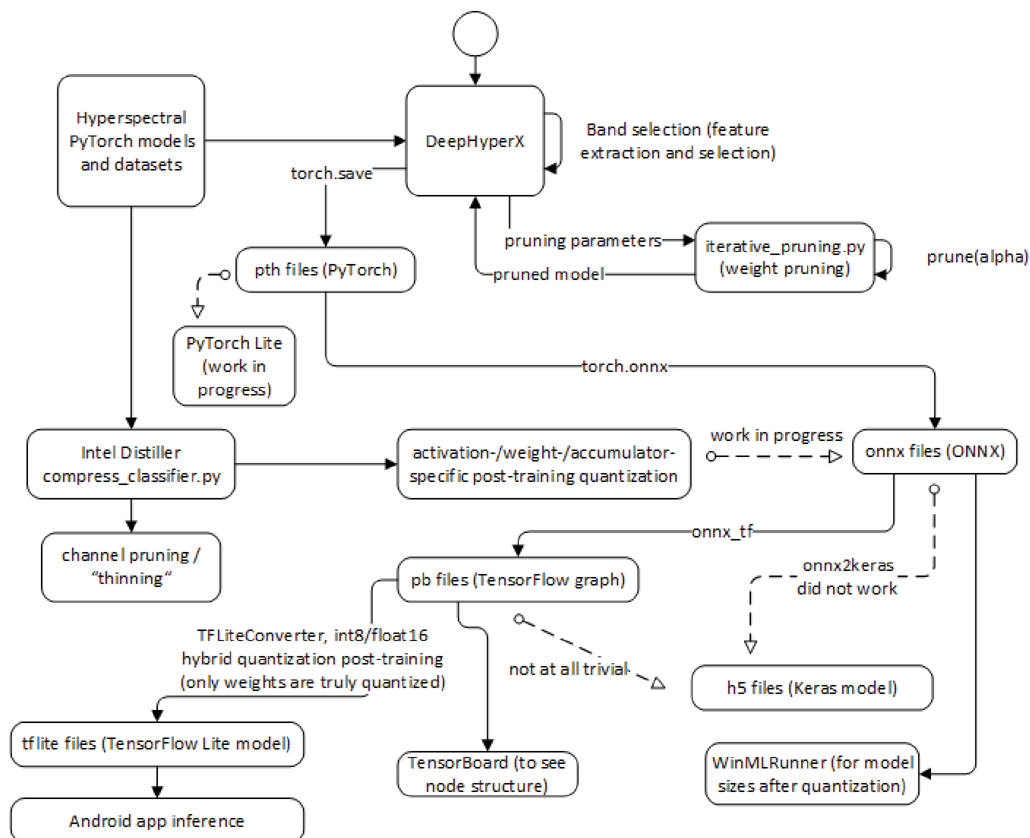


Figure 65: Architecture considerations for compression-related experiments in chapter 4.2. Core components include the hyperspectral models and datasets as well as band selection covered by DeepHyperX, `iterative_pruning.py` implementing fine-grained pruning, Intel Distiller allowing for coarse-grained pruning and component-specific quantization variants, as well as other quantization possibilities for the sake of calculating the model size reductions, seeing the nodes in a visualization and obtaining a model suitable to be used for an Android app, notwithstanding the considerable challenges linked with the computational restrictions of mobile devices.

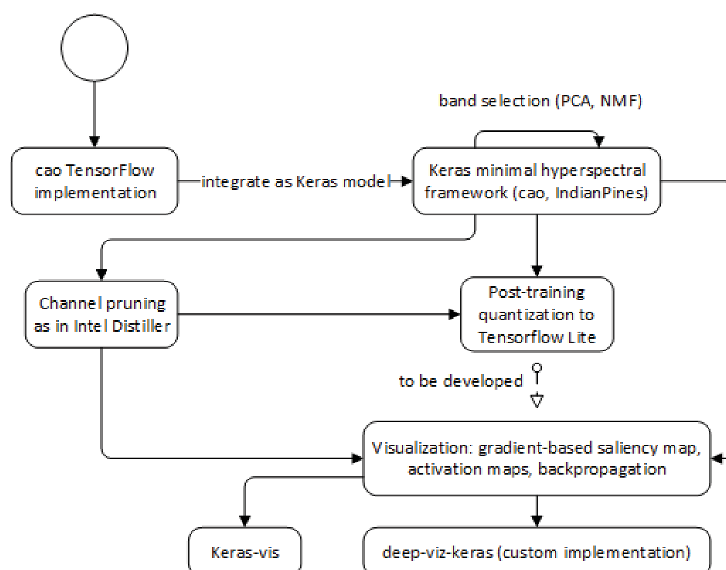


Figure 66: Architecture considerations for visualization-related experiments in chapter 4.3. Once a hyperspectral framework is built in Keras and an appropriate model implemented, channel pruning like we have seen in Intel Distiller can be connected, but this time, with APoZ as the pruning criterion (because such a code fragment has been available and we have worked with it before). The fact that Keras is a high-level TensorFlow-API helps making the quantization process more straightforward than in PyTorch and a matter of few lines of code. Keras-vis offers a variety of inbuilt visualization methods one could use, but for more specific variations like in our case, one could consider own implementations.

9.5 Additional Measurements

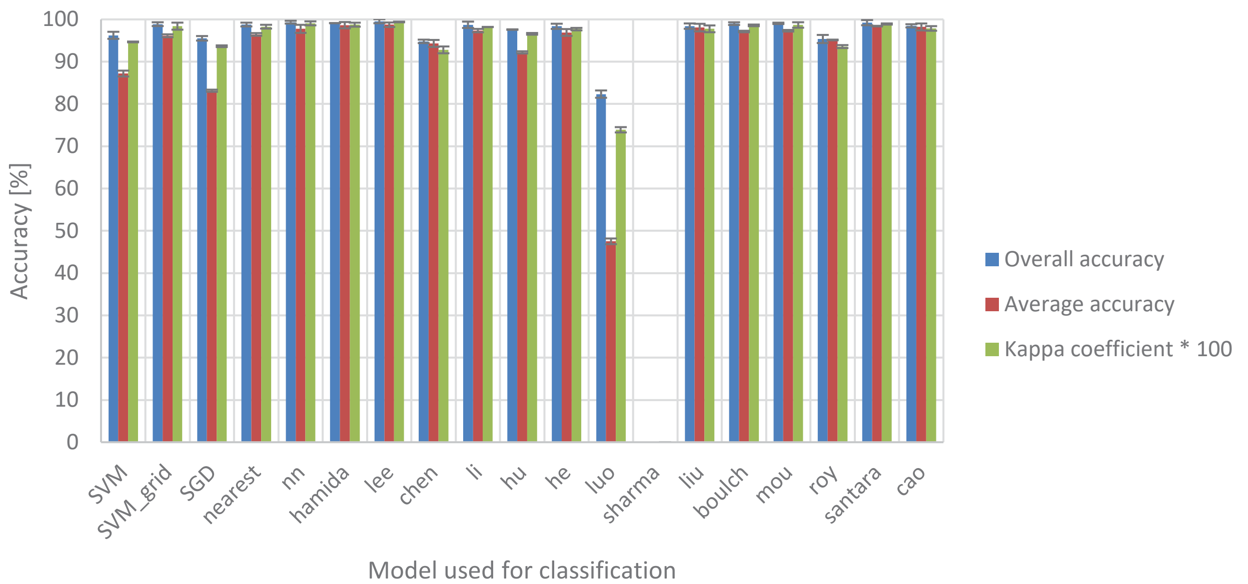


Figure 67: Model accuracies for the PaviaC dataset (n=6, CI=95%). OA, AA and Kappa are so high in general that this dataset would have been a bad foundation for further experiments.

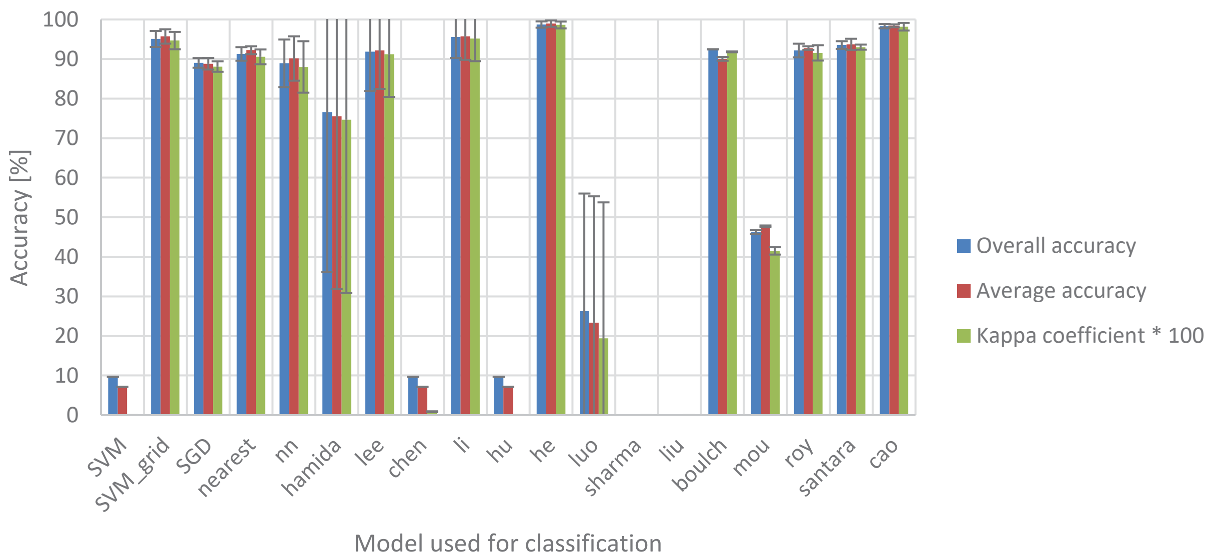


Figure 68: Model accuracies for the Botswana dataset (n=6, CI=95%). The extreme disparities between well and badly performing models discouraged us from choosing this dataset. The exorbitantly large CIs for hamida and luo are conspicuous.

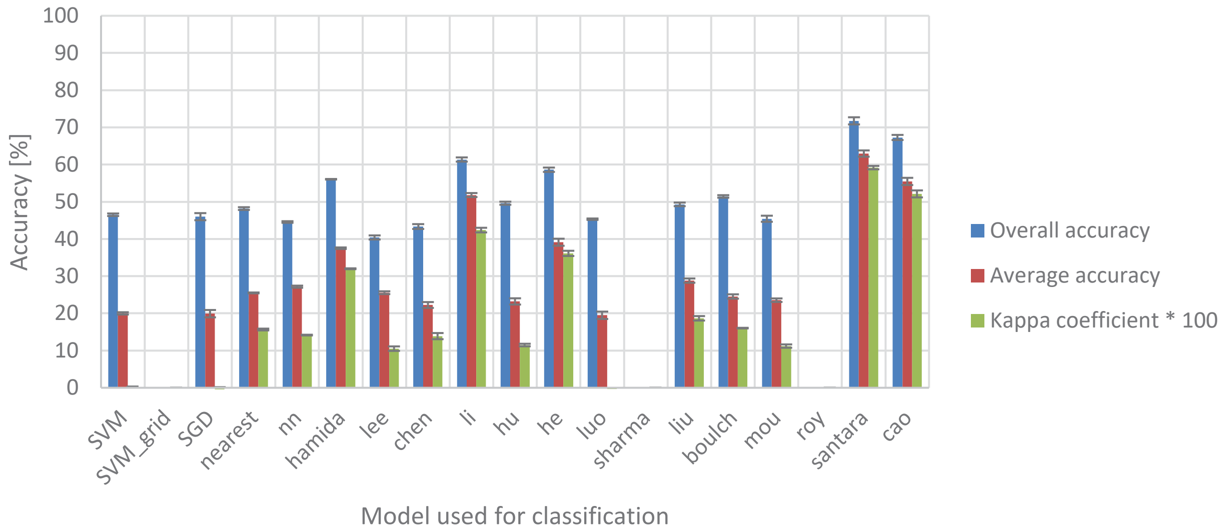


Figure 69: Model accuracies for the Urban-210 dataset (n=6, CI=95%). OA is so much greater than AA, which in turn is greater than Kappa, that this fact that there must be classes difficult to classify discouraged us from choosing this dataset.

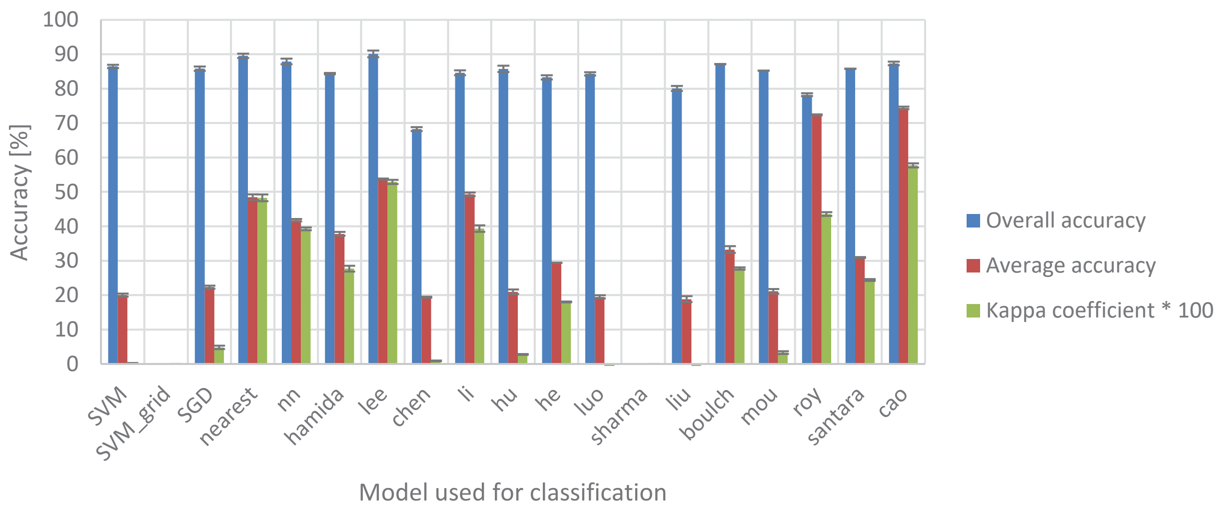


Figure 70: Model accuracies for the USA dataset (n=6, CI=95%). The phenomenon of Figure 69 is even more prevalent here, with OA amounting to more than triple the AA at times. The dataset being rather unknown to the scientific community also contributed to us not choosing this dataset.

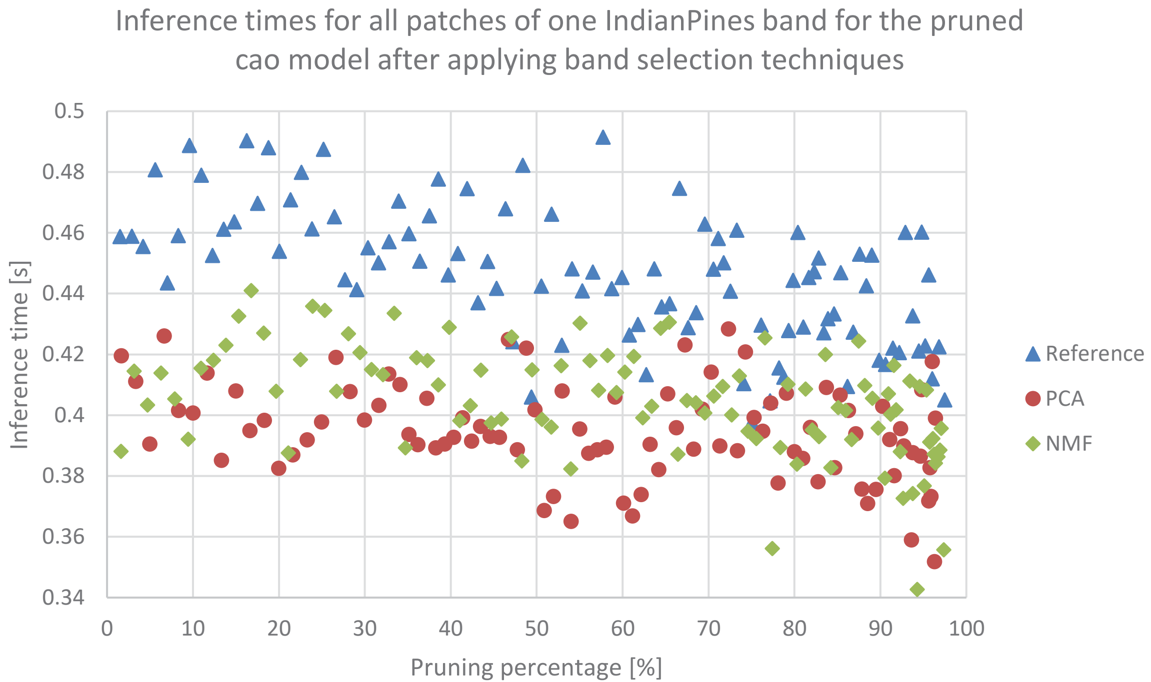


*Figure 71: Inference times for the cao model in the Keras implementation depending on the APoZ channel-based pruning percentage (n=**5** because sufficed for statistical significance, CIs not shown for visibility). The reference times (no prior band selection) surpass the ones after PCA / NMF by roughly 20%. The more is pruned, the closer the PCA and NMF times get. Regardless, we can clearly see that pruning helps to improve inference times, which motivates further research.*

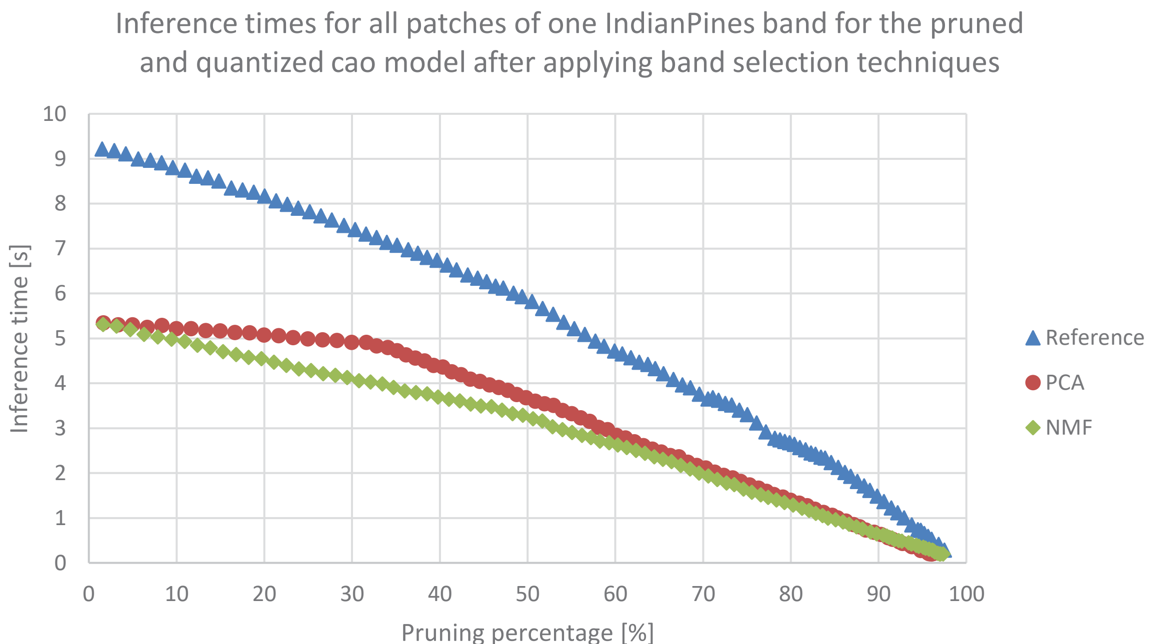


*Figure 72: Inference times for the cao model in the Keras implementation after quantization depending on the APoZ channel-based pruning percentage (n=**5** because sufficed for statistical significance, CIs not shown for visibility). The inference times are about 20 times as high as in Figure 71, which may be attributed to the fact that the way we needed to do inference for the `tflite` model (`interpreter.invoke` in a loop for every patch) is not as efficient as the `model.predict` method for `h5` (which considers all patches and is not available for `tflite`). Regardless, we see linear a decrease for all three cases, where the PCA and NMF curves converge the more we prune.*

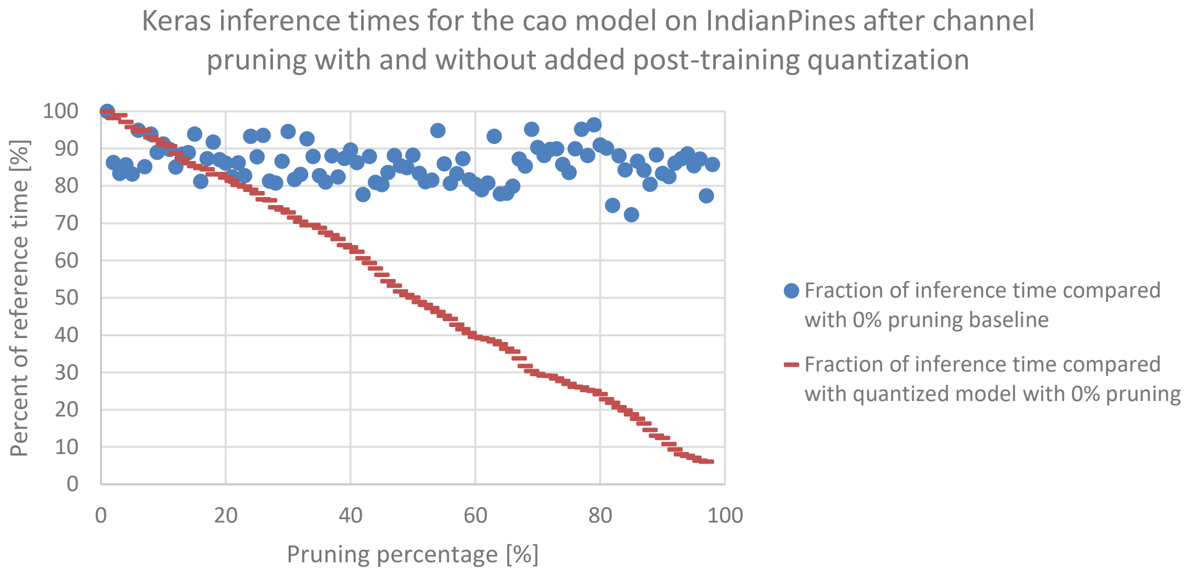


*Figure 73: Comparison of inference time savings for pruning alone vs. for pruning and post-training quantization depending on the APoZ channel-based pruning percentage (no prior band selection applied; n=**5** because sufficed for statistical significance, CIs not shown for visibility). Both curves show that the inference times decrease, but the pruning data is much more scattered, whereas the linear trend is clearly observable for quantization. We should also keep in mind that while these relative inference time savings sound nice, the absolute values shown in Figure 71 and Figure 72 reveal that the quantization inference times are about 20 times as high as the pruning inference times, as described for these figures.*

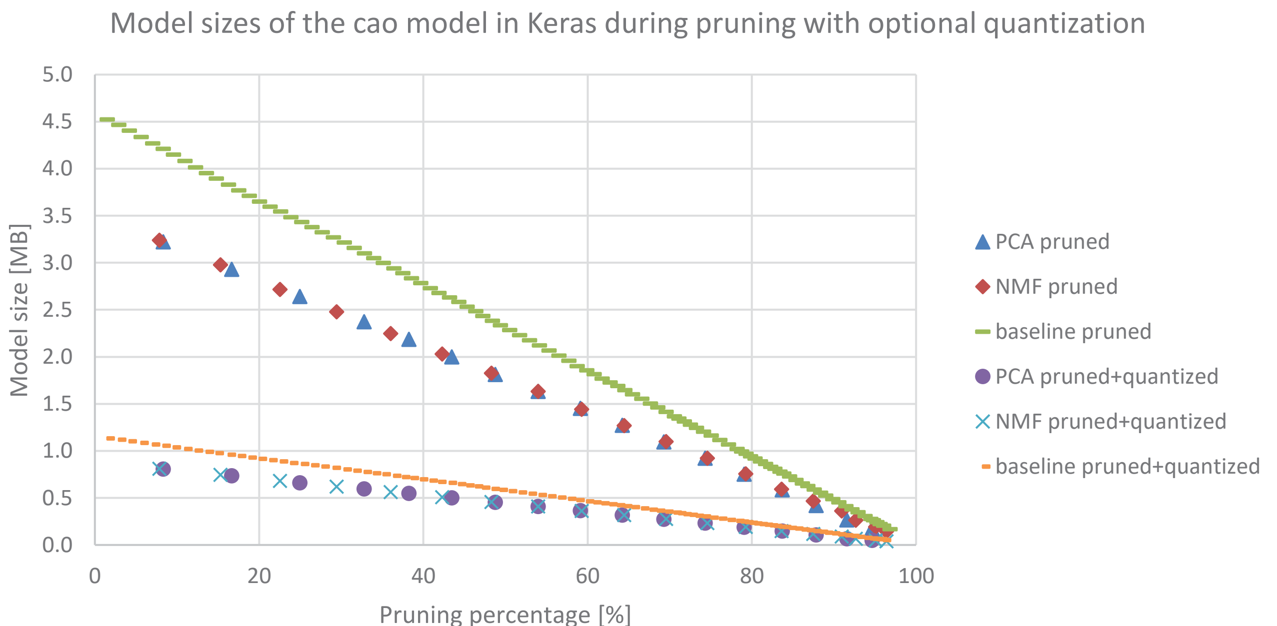


Figure 74: Model sizes of the cao model implemented in Keras after pruning with optional quantization on top after applying (or not applying) the band selection techniques PCA or NMF, depending on the APoZ channel-based pruning percentage. Quantization does account for a huge model size reduction (3MB), whereas the curves do not react that strongly to pruning anymore if we decide to quantize. A band selection technique saves us space. All trends observed scale linearly with the pruning percentage of the model.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 19.11.2019 ………………………………………………………………