Master's Thesis

submitted by

Bernhard Haas

Registration Number 01525110

Compressing MobileNet With Shunt Connections for NVIDIA Hardware

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria,

Study code:	066 646
Field of study:	Computational Science and Engineering
Supervisor:	Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Co-Supervisor:	Projektass. Dipl.-Ing. Dr.techn. Alexander Wendt

If you find this work useful, please cite it using the following BibTeX entry:

```
@Thesis{AuthorLastName2021,
 type        = {Master's Thesis},
 author      = {Bernhard Haas},
 title       = {Compressing MobileNet With Shunt Connections for NVIDIA Hardware},
 school      = {Vienna University of Technology (TU Wien)},
 year        = {2021},
 address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
 month       = {May},
}
```

Contact me:

bernhardhaas55@gmail.com

Abstract

Although modern convolutional neural network architectures achieve astounding results on large-scale tasks like semantic segmentation, employing them on embedded devices still resembles a big challenge, hindering their applications for the real world. This thesis aims at closing the computational gap between embedded devices like the NVIDIA Jetson series and modern deep-learning architectures through improving the method of compressing networks through shunt connections. This thesis shows the applicability of shunt connections on MobileNetV3 and for the semantic segmentation task on the Cityscapes dataset. For this, knowledge distillation is explored as a method to efficiently fine-tune shunt inserted models, while the achieved speed-up of shunt connection compression is measured on the NVIDIA Jetson Xavier device to validate results. It is shown that MobileNetV3 trained on CIFAR100 can be compressed by 31% without losing any accuracy. MobileNetV3 trained on Cityscapes is compressed by 28%, resulting in a drop of four points of mIoU. Results are compared against the compression through using smaller depth multipliers, proving the advantages of shunt connections.

Kurzfassung

Obwohl enorme Fortschritte im Bereich ’semantic segmentation’ mithilfe von neuartigen neuronalen Netzwerkarchitekturen erzielt wurden, besteht das Problem immer noch, diese Netzwerke auch auf ressourcenschwacher Hardware in akzeptabler Zeit auszuführen, wodurch diese Netzwerke in der Praxis nur eingeschränkt eingesetzt werden können. Ziel dieser Diplomarbeit ist, diese Kluft zwischen modernen ’deep-learning’-Netzwerken und ’embedded’-Hardware, wie beispielsweise Geräte der NVIDIA Jetson Serie, zu verkleinern oder bestenfalls zu schließen. Dafür soll die Kompressionsmethode der sogenannten ’shunt connections’ genutzt werden. Diese Diplomarbeit zeigt erste Resultate von ’shunt connections’ für die MobileNetV3-Architektur und für den ’semantic-segmentation’-Datensatz ’Cityscapes’. Resultate werden erzielt, indem die Modelle mittels ’knowledge distillation’ trainiert werden. Erste Beschleunigungswerte auf dem NVIDIA Jetson Xavier werden ebenfalls berichtet. Es wird gezeigt, dass MobileNetV3, trainiert auf dem CIFAR100 Datensatz, um 31% komprimiert werden kann, ohne dabei an Genauigkeit des Modells einzubüßen. Trainiert man MobileNetV3 auf ’Cityscapes’, lässt sich dieses Modell um 28% komprimieren, wobei 4 Punkte an mIoU verloren gehen. Resultate werden mit der Methode verglichen, MobileNetV3-Modelle mittel ’depth-multiplier’ zu komprimieren, was die Vorteile von ’shunt connections’ weiter hervorhebt.

Contents

Chapter 1

Introduction

Machine learning and especially deep learning ended the AI winter of the early 21st century. According to mainstream media, machine learning, more often referred to as artificial intelligence (AI), can advance humanity to the next level, promising a bright future full of intelligent technology in every aspect of our lives[1].

Applications covered by media often utilize cloud computing for model inference since these models require huge computational power to be run in a feasible time frame. This setup is not an option for fault-critical devices, such as an autonomous car, because the connection to the cloud could be lost easily. As a result, inference has to be done locally on embedded devices with much less computational resources available. This hardware is probably not able to run the most modern network architecture at the desired speed. Closing this gap is not a trivial task, since models often have to be run in real-time while keeping power and memory consumption constraints in mind.

These days, visual recognition tasks are solved by deep learning models using millions of learnable parameters. For each forward pass through the network, thousands of tensor multiplications have to be performed. It is very likely, that running your state-of-the-art convolutional neural network on a standard CPU will have an inference latency of over one second. Hence, these models are mostly employed on GPUs or similar hardware like NPUs or TPUs.

The situation is even more discouraging when trying to run the best-performing models on embedded devices, which hold very limited computational power. Hence, deep network architectures, which hold much fewer parameters and are easier to compute, were designed exactly for this purpose. The most prominent architecture family is arguably MobileNet [How+17; San+19; How+19], which was proposed with the idea to employ convolutional neural networks on mobile devices.

Combining these models with embedded computers, for example, devices from the NVIDIA Jetson series, which is aimed at enabling modern machine learning models, it is possible to run certain tasks in real-time, although higher workloads are still out of reach. Therefore, it is necessary to optimize the already relatively small models further concerning inference time. This task is called neural network compression, which is achieved by reducing redundancy inside the model and compressing its knowledge. One relatively new proposed method is called shunt connections [STA19], where a whole part of a model is replaced with a smaller model. This novel approach holds great potential since it is possible to

[1] https://www.theguardian.com/commentisfree/2019/jan/13/dont-believe-the-hype-media-are-selling-us-an-ai-fantasy

produce really small models, which still provide high accuracies in their respective tasks. So far, it has been shown that shunt connections can compress efficiently the MobileNetV2 architecture trained on small classification tasks.

This thesis investigates the applicability of shunt connections in a more general setting and quantify results regarding latency speed-up on embedded hardware. The following questions should be answered: Can the results of shunt connections on MobileNetV2 be replicated for the newer MobileNetV3 architecture? How big is the accuracy drop, when applying shunt connections to the MobileNetV3 architecture trained on a large-scale dataset in the form of the Cityscapes dataset? How big is the latency speed-up of shunt-inserted models on the NVIDIA Jetson Xavier?

To answer those questions, this thesis consists of the following steps:

- Extract the compression method from state-of-the-art-sources
- Implement the method in the same way as state-of-the-art on the same datasets and MobileNets
- Evaluate the MobileNets on NVIDIA hardware regarding latency and accuracy
- Apply and evaluate the MobileNetV3 segmentation network with the same methods on NVIDIA hardware

The first two steps should reproduce the state-of-the-art regarding shunt connections and lay down the foundations for the next steps. This work does not aim at providing the best possible accuracy result for a given model, hence hyperparameter optimization is only applied in a few cases.

Although the original shunt connection paper relies on PyTorch [Pas+19] as its deep learning backend, this thesis uses TensorFlow [Mar+15] and Keras [Cho+15] to implement models and train them. Both PyTorch and TensorFlow are using Python [VD09] as their main language and are highly valued by the online community, making the choice just a matter of personal preference.

Chapter 2 should provide the necessary theory and state-of-the-art needed for building modern convolutional neural network architectures, training them, and neural network compression including shunt connections. Chapter 3 describes the methodology used for employing shunt connections and chapter 4 shows how to implement the necessary tools in Keras and TensorFlow. Afterwards, the results of the original shunt connection paper are tried to be reproduced in chapter 5 and extended for the MobileNetV3 case for classification and semantic segmentation.

Chapter 2

Theory and State of the Art

This chapter should act as a small introduction into modern convolution neural networks used for visual recognition tasks and how to optimize them in regard to latency. The state-of-the-art regarding shunt connections is also presented in detail.

2.1 Image Analysis

Analysis of image data is necessary for a variety of fields with many different tasks. Large progress was achieved in recent years in visual recognition tasks like image classification and semantic segmentation. Since this thesis conducts all experiments on these tasks, this section should define them and show how to evaluate models on them. For the sake of completeness, the task of object detection, which ties nicely in classification and semantic segmentation, should be briefly explained as well.

2.1.1 Image Classification

The goal of this task is to assign an image a predefined label or class. Since the output is a single scalar value corresponding to a class, this task is arguable the most simple of the three presented in this thesis. A dataset used by a lot of researchers for proof of concepts of novel methods or models, are the CIFAR [Kri12] datasets: CIFAR10 and CIFAR100. The images in colour have a resolution of 32x32, which are labelled using 10 classes, in the case of CIFAR10, or 100 classes, in the case of CIFAR100. Some sample images are shown in fig.2.1.

Models can be evaluated by computed the relative number of correctly classified samples n_{corr}, also known as accuracy acc:

$$\text{acc} = \frac{n_{\text{corr}}}{n_{\text{samples}}} \cdot 100\%. \tag{2.1}$$

When a sample is considered as classified correctly, is most of the time defined by the top-1 or top-5 predictions. When considering top-5 predictions, a sample is correctly classified, if the target class is contained in the top 5 predictions of your model. In the case of top-1, the most likely class predicted by your model has to match the target class.

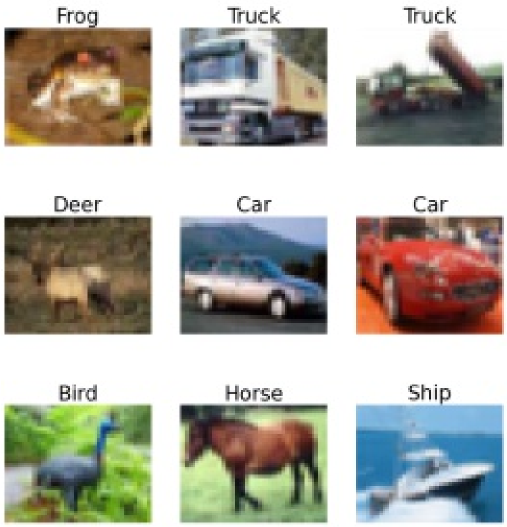
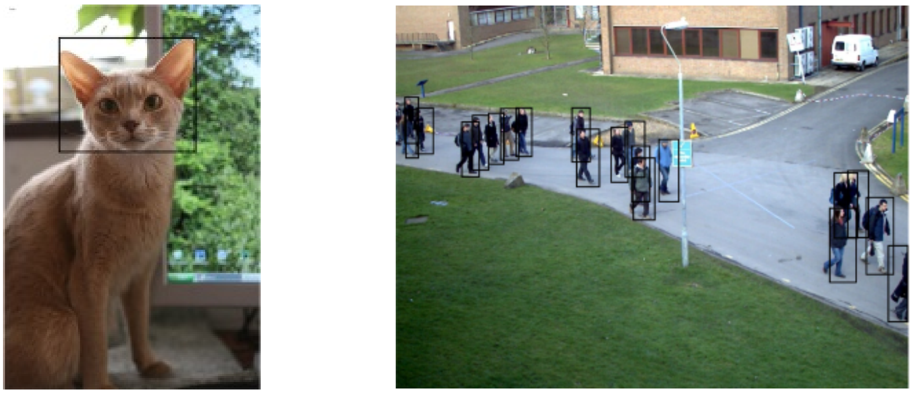


Figure 2.1: Nine random samples from the CIFAR10 dataset.

2.1.2 Object Detection

Going one step further from the simple classification of the whole image is the localization of the predicted object within the image. For example, localizing the predicted animal or object within the image of the CIFAR10 dataset. If multiple objects, potentially with different labels, are visible within one image, one does not talk about image classification, but object detection (see fig.2.2). Usually, objects are marked by simple bounding boxes indicating the location and size of the object.



(a) Image classification of an image showing a cat.

(b) Object detection of pedestrians.

Figure 2.2: One sample of image classification and object detection.

2.1.3 Semantic Segmentation

Localizing objects by bounding boxes may be not accurate enough for certain tasks. Detecting objects on a pixel-level basis is called semantic segmentation. In this case, each pixel of the image gets categorized by a class. One of the most used semantic segmentation dataset, which is openly available, is Cityscapes [Cor+16], which describes urban scenes. Some samples of it are shown in fig.2.3.

Model evaluation is a bit more complicated in the case of semantic segmentation. One could use the top-1 accuracy for

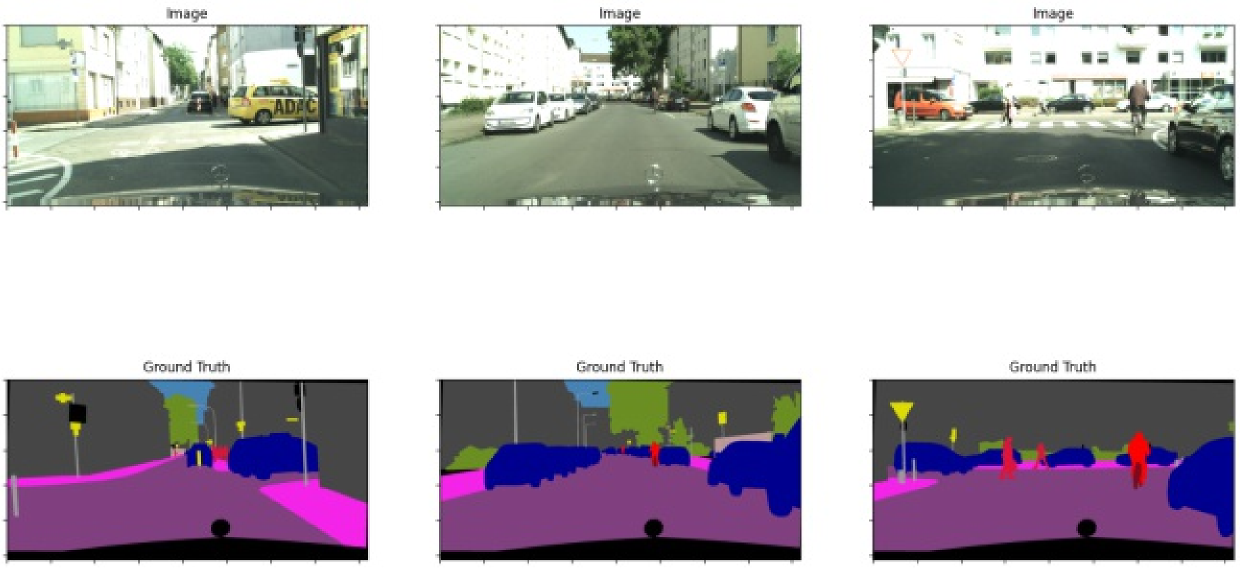


Figure 2.3: Three random samples from the Cityscapes dataset.

each pixel and average the scores over the whole image. The problem with this approach is, that the point of semantic segmentation is often to detect a small object against a big background. Hence, the model's accuracy would be quite high if the whole image would get classified as 'background', although it misses the goal of the task completely. Most of the time (f.e. [LSD15]), a more suitable apporach is to compute the mean intersection over union (mIoU) for each class and average over all classes:

$$\mathrm{mIoU} = \frac{1}{n_{\text{classes}}} \sum_{i=1}^{n_{\text{classes}}} \frac{A_i^{\text{inter}}}{A_i^{\text{union}}}, \tag{2.2}$$

where A_i^{inter} is the area of intersection and A_i^{union} the area of union between the predicted labels and ground truth labels of class i.

2.2 Convolutional Neural Networks

The exploration of convolutional neural networks (CNNs) is the main reason for recent advancements in tasks like image classification, object detection, and semantic segmentation. These networks build the backbone of many computer vision algorithms since traditional approaches do not perform well on these tasks. This section explains the basic structure behind CNNs and how to train them. For introducing different CNN architectures, a brief history of the ImageNet challenge is given.

2.2.1 Stacking Perceptrons: Neural Networks

Before going into CNNs, general neural networks shall be discussed. Their basic building block is the perceptron, which got already introduced back in 1958 [Ros58]. It is motivated by the structure of the human brain and was initially used as a linear classifier for binary classification tasks.

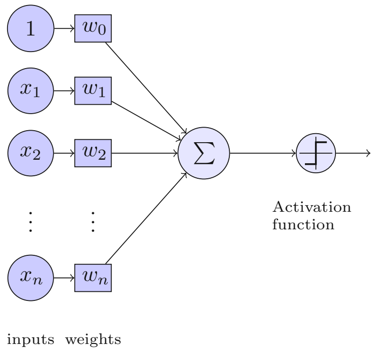


Figure 2.4: General structure of a perceptron.

Each input variable and an intercept term are multiplied by a learnable parameter called weight and summed up to produce a single output. Afterwards, this value gets put into a non-linear 'activation' function, for example, the Heaviside function, to produce a binary output. This structure is visualized in fig.2.4.

Combining multiple perceptrons leads to multilayer perceptrons also known as neural networks. A perceptron in layer l multiples each output O^i_{l-1} from the previous layer by a learnable weight, sums up all n_{l-1} outputs, and adds its second learnable parameter called bias b. This value is again put into a non-linear activation function f to produce the output O^i_l:

$$O^i_l = f\left(\sum_{i=0}^{n_{l-1}} O^i_{l-1} w_i + b\right). \tag{2.3}$$

There can be an arbitrary number of perceptrons (neurons) in each layer and also an arbitrary number of layers in the neural network. The simplest architecture is shown in fig.2.5, where only one hidden layer of perceptrons is used between the input and output layers. Networks that use more than one hidden layer are also referred to as deep neural networks.

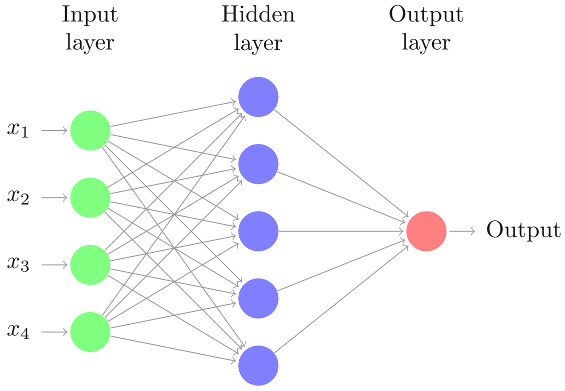


Figure 2.5: General structure of a neural network [Fau].

So far, it has not been mentioned how to train these models. In the case of neural networks, all weights and biases of each perceptron have to be learned. This is usually done by the combination of gradient descent and backpropagation [Wer90], which can be used to minimize a loss function. This loss function must correspond to the task one wants to solve. The gradients can be calculated by backpropagation, where the gradient of the top layers gets calculated first and the gradients of previous layers iteratively afterwards.

The choice of the right activation function is not obvious. For a binary classification problem, the Heaviside function seems the straightforward choice but has the disadvantage that it is not differentiable, therefore not suitable for training strategies using gradient descent methods. The sigmoid function was the most used activation function, being very similar to the Heaviside function, along with tanh. In modern deep neural networks, rectified linear units (ReLU) and variations of this function are extensively used. These activation functions are visualized and compared in fig.2.6.

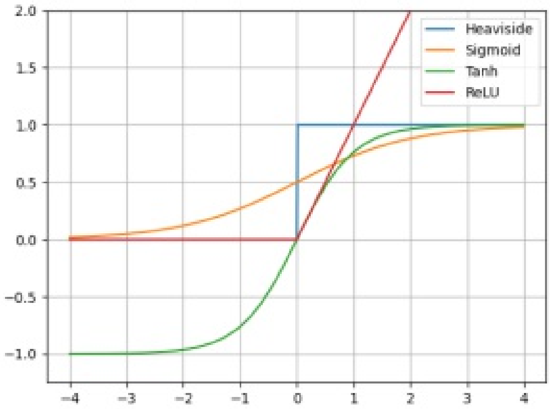


Figure 2.6: Comparison of most popular activation functions for perceptrons and neural networks.

2.2.2 Convolutional Layers

The neural networks presented until now were fully connected networks, meaning a neuron is connected to all neurons from the previous layer. This leads to a loss of the locality of input data, which may be crucial for certain tasks.

For tasks regarding object detection or classification of images, data locality is of high importance. Therefore, the fully connected neural network architecture is not suitable for such tasks. The breakthrough was the LeNet [Lec+98] paper, which introduced convolutional neural networks (CNNs) for handwritten digit classification tasks. CNNs are not using fully connected layers but use convolutional layers, which are strongly inspired by convolutional kernels used by many computer vision applications. Convolutional kernels are mathematical operations, where each pixel's value of an image is changed according to its own value as well as its neighbours' values in a matrix multiplication fashion. One convolutional layer consists of many kernels. Each of the kernels produces a new image called a feature map while acting on all feature maps produced by the previous layer (see fig.2.7).

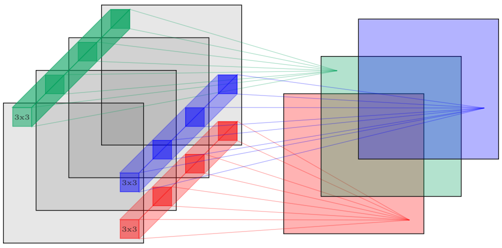


Figure 2.7: Working principle of a convolutional kernel.

To achieve good results on more complex tasks, it is necessary to generate models with as many learnable parameters as possible, while still being able to properly train the network. Number of parameters in CNNs are dependent on kernel size k, number of input feature maps n_{IFM} and number of output feature maps n_{OFM}. The number of parameters n_{P} is given by:

$$n_{\mathrm{P}}^{\mathrm{conv}} = n_{\mathrm{IFM}} \cdot k \cdot k \cdot n_{\mathrm{OFM}}. \tag{2.4}$$

One main raison, why CNNs can be implemented and used nowadays, is the fact that they can be efficiently run on GPUs. Neural networks including CNNs are inherently computational expensive since they require a very high number of floating point operations (FLOPs) in the form of kernel convolutions. Those convolutions are implemented through many matrix-matrix-multiplications, which are highly parallelizable through GPUs. Hence, CNNs consists mostly of multiply-accumulate operations (MACs), where one MAC translates to 2 FLOPs. The MACs of a single convolutional kernel can be computed by this formula:

$$\mathrm{MACs}_{\mathrm{conv}} = n_{\mathrm{IFM}} \cdot k \cdot k \cdot n_{\mathrm{OFM}} \cdot \mathrm{out}_{\mathrm{h}} \cdot \mathrm{out}_{\mathrm{w}}, \tag{2.5}$$

with $\mathrm{out}_{\mathrm{h}}$ and $\mathrm{out}_{\mathrm{w}}$ being the height and width of output feature maps.

2.2.3 Depthwise-seperable Convolutions

Employing deep CNNs with many convolutional layers is not feasible for mobile and embedded devices, since the MACs are simply too high for devices holding a weak GPU or no GPU at all. Depthwise-separable convolutions are responsible for closing this gap. They can be motivated by looking at the Sobel filter [Sob14], which is well known in the domain of computer vision. Its definition and its decomposition are given in eq.2.6. Although this matrix has nine elements, it can be expressed as the dot product of two vectors with 3 elements each. Therefore, the Sobel kernel can be expressed by only 6 elements.

$$S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{2.6}$$

The realization, that convolutional kernels are often symmetrical and can be described by two vectors, leads to the idea of depthwise-separable convolutions [How+17]. As it turns out, convolutional kernels can be decomposed similarly, if assuming some form of symmetry. Instead of performing one convolution on all feature maps, one convolution is computed only on a single feature map (see fig.2.8). This gives an intermediate result, where each feature map got transformed by a convolutional kernel. This process is called depthwise convolution. To form the final output, a pointwise convolution, meaning a convolution with a 1x1 kernel, is applied to all intermediate results.

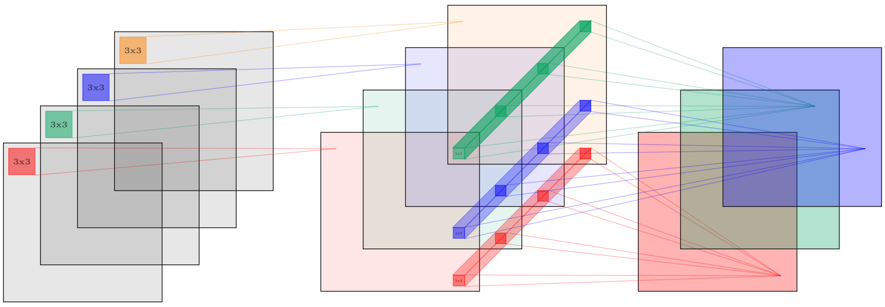


Figure 2.8: Working principle of a depthwise-separable convolution.

The true benefit of this approach becomes visible, when calculating the MACs of a depthwise-separable convolutional layer:

$$\text{MACs}_{\text{dw-sep conv}} = n_{\text{IFM}} \cdot k \cdot k \cdot \text{out}_{\text{h}} \cdot \text{out}_{\text{w}} + n_{\text{IFM}} \cdot \text{out}_{\text{h}} \cdot \text{out}_{\text{w}} \cdot n_{\text{OFM}}, \tag{2.7}$$

which results in the relative reduction of MACs given by:

$$\frac{\text{MACs}_{\text{dw-sep conv}}}{\text{MACs}_{\text{conv}}} = \frac{n_{\text{IFM}} \cdot k \cdot k \cdot \text{out}_{\text{h}} \cdot \text{out}_{\text{w}} + n_{\text{IFM}} \cdot \text{out}_{\text{h}} \cdot \text{out}_{\text{w}} \cdot n_{\text{OFM}}}{n_{\text{IFM}} \cdot k \cdot k \cdot n_{\text{OFM}} \cdot \text{out}_{\text{h}} \cdot \text{out}_{\text{w}}} = \frac{1}{n_{\text{OFM}}} + \frac{1}{k \cdot k}. \tag{2.8}$$

Therefore, it is proven that for a high number of output feature maps, depthwise-separable convolutions outperform the regular convolutional layer by a large margin regarding the number of MACs.

It is also interesting to look at the number and relative reduction of parameters of a depthwise-separable convolution:

$$n_{\text{P}}^{\text{dw-sep conv}} = n_{\text{IFM}} \cdot k \cdot k + n_{\text{IFM}} \cdot n_{\text{OFM}} \tag{2.9}$$

$$\frac{n_{\text{P}}^{\text{dw-sep conv}}}{n_{\text{P}}^{\text{conv}}} = \frac{n_{\text{IFM}} \cdot k \cdot k + n_{\text{IFM}} \cdot n_{\text{OFM}}}{n_{\text{IFM}} \cdot k \cdot k \cdot n_{\text{OFM}}} = \frac{1}{n_{\text{OFM}}} + \frac{1}{k \cdot k}. \tag{2.10}$$

2.2.4 Training of CNNs

Just like normal neural networks, the parameters of CNNs are optimized by minimizing a loss function using a gradient descent method, where the gradient updates are calculated using backpropagation.

The loss function has to be differentiable in order to find the minimum through the gradient descent method. Hence, many accuracy measurements are not suitable to be used as a loss function. Rather, additional loss functions which are related to the task have to be defined. For example, in the case of image classification, the top-1 or top-5 accuracy measurements are not differentiable, hence the cross-entropy function is chosen as the loss function. The cross-entropy function compares not only the predicted class of the network to the ground-truth label but rather the whole produced prediction vector y_m of the model to the one-hot encoded ground-truth label y_{gt}:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{k} y_{gt} log(y_m). \tag{2.11}$$

The loss is minimized when the predicted probability of the ground truth equals 1.

This loss is usually minimized through an iterative gradient descent solver. There are many variations available [Rud17], like gradient descent with momentum, stochastic gradient descent or Adam [KB17]. The most popular of these was stochastic gradient descent (SGD) with momentum for a long time, where gradients are computed on mini-batches of around 4-256 samples instead of the whole dataset.

There exist a lot of different techniques to train a CNN like doing a warm-up phase, complicated learning rate policies, or data augmentation. One method, which should be well understood, is batch normalization, which is explained in the next section.

2.2.5 Batch Normalization

Batch normalization layers are another very important layer type, one can find in almost all modern CNN architectures. It was first introduced in 2015 [IS15] and claims to stabilise and accelerate the training of deep neural networks, including CNNs. The theory behind it is not understood completely, but the initial idea was to reduce the covariate shift, which builds up during the forward pass of a neural network. Covariate shift is described by [Che+17a] in its abstract as: 'In deep neural networks, inputs in each layer are affected by all previous parameters of input layers, so even small changes in input distributions to the network are delivered to internal layers, for leading to differences between source domain and target domain, which is known as covariate shift'.

Batch normalization solves this problem by normalizing each output feature map of the previous convolutional layer independently. Each feature map x of the layer is normalized in the first step:

$$\hat{x} = \frac{x - \mu_B(x)}{\sqrt{\sigma_B(x)}}, \tag{2.12}$$

where $\mu_B(x) = \frac{1}{m}\sum_{i=1}^{m} x_i$ and $\sigma_B = \sqrt{\frac{1}{m}\sum_{i+1}^{m}(x_i - \mu_B)^2}$. μ_B and σ_B are the mean and standard deviation of one mini-batch with size m. During training, these values get calculated for each mini-batch independently, since the global information of the whole training data is not available.

This produces the intermediate result $\hat{x}$ with a mean of 0 and a standard deviation of 1. Since applying only this transformation would limit the descriptive power of the network, another transformation is applied to the normalized result:

$$y = \gamma\hat{x} + \beta, \tag{2.13}$$

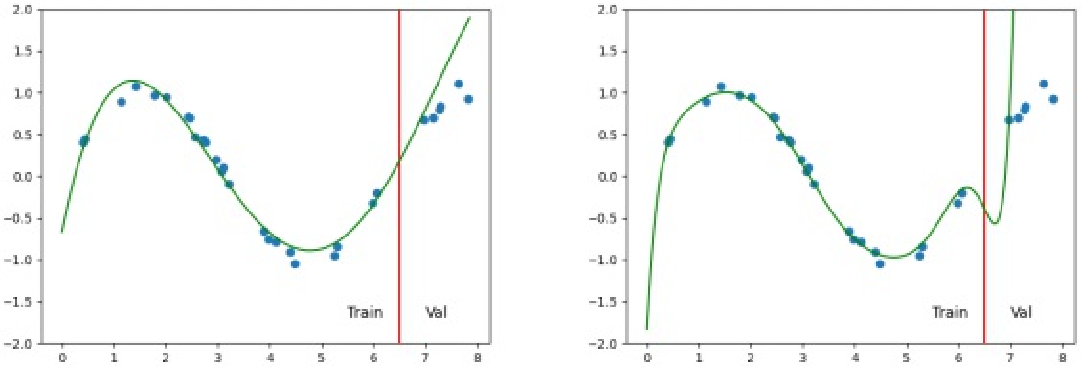
with y being the new output of the feature map, and γ and β are learnable parameters.

So far, it was shown how batch normalization is applied during training, where a mini-batch is available for estimating μ_B and σ_B. During the inference of a single sample, a mini-batch is not available. Therefore, the running mean and standard deviation holding the global information of the whole training data is also estimated during training. This estimated mean and standard deviation is then used for transformation during inference.

Although batch normalization holds many advantages, there is also a big drawback to the method. Since the mean and standard deviation is estimated over a mini-batch during training, batch size has to be as high as possible to get a meaningful estimate. This can represent a big problem if network architectures need a lot of memory to compute and computational resources are limited. Deep CNNs get exclusively trained on GPUs or similar architectures with a fixed size of memory storage. This means that expensive equipment is often necessary for training modern CNN architectures.

2.2.6 Regularization Techniques

When training CNNs, a problem occuring very often is overfitting, which describes the scenario that a model fits the training data too well, resulting in bad predictions. This phenomenon is nicely visualized by comparing two different polynomial fits with different degree. The data consists of a sin-wave with a little random noise. Fig.2.9a shows the result for a polynomial of degree 3, which yields relative good results for the validation data on the right hand side. One would expect, that a more complex model, meaning a higher degree polynomial would give better predictions, since the training data can be fit more precisely. Looking at fig.2.9b, where the polynomial degree is 11, this statement can be disproven. By fitting the training data exactly, the model loses its ability to generalize.



(a) Degree = 3

(b) Degree = 11

Figure 2.9: Polynomial fit as an example overfitting. Two different fits are shown using different polynomial degree.

Neural networks and CNNs are prone to overfitting since they hold so many parameters, that they are able to memorize training data really well without generalizing to the task. Therefore, different techniques were proposed to prevent CNNs from overfitting.

Regularization One fact, which holds true for polynomial fits and CNNs, is, that the absolute values of the coefficients of the model are really big when overfitting. Hence, when limiting the possible values for coefficients, one also limits the amount of overfitting. In the case of CNNs, the absolute value of weights is penalized instead of limiting the range of possible values for weights. This penalization is interpreted as an additional loss, which has to be minimized besides the task loss. The most common functions are the simple L1 and L2 loss:

$$L1(\theta) = \lambda \sum \|\theta\|, \qquad L2(\theta) = \lambda \sum \theta^2, \tag{2.14}$$

with θ being the weights of the model and λ a hyperparameter controlling the regularization strength. Commonly used values for λ are between 1e-3 and 1e-5.

Dropout Besides the regularization techniques, which are also used for other machine learning algorithms, CNNs can also make use of dropout. Dropout means that during training, the input of random neurons is set to 0, therefore ’dropped out’ for one iteration. One hopes, that this leads to models, which consider many different features and do not rely on a single feature existing in an image. They are most commonly used in the last stage of the network, which consists often of fully connected layers. The hyperparameter *rate* controls how likely it is that the input of a neuron is dropped, while the input of other neurons of the layer are scaled by $\frac{1}{1-\text{rate}}$ to ensure that the sum of all inputs stays the same.

2.3 The ILSVRC and its Influence on CNNs

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Rus+15a] started in 2010 intending to standardize results on visual recognition tasks. Here, the focus lies on the classification task where 1000 distinctive classes are used to categorize around one million images. The challenge was held yearly from 2010 to 2017, where the dataset was updated slightly multiple times.

The first competitor using a CNN for the image classification competition was AlexNet [KSH12], beating the previous winner by over 10% and achieving a top-5 accuracy of 84.7% on the test set. AlexNet consists of 5 convolutional layers using a mixture of 3x3 and 5x5 kernels and ReLUs followed by fully connected layers, which sum up to 61 million parameters while using 724 million MACs. The next popular architecture introduced through the ILSVRC, is VGGNet [SZ14], which showed that deeper neural networks are needed to achieve better results. The largest variant of this architecture: VGG-19 holds 16 convolutional layers with 3x3 kernels, resulting in 138 million weights and 15.5 billion MACs.

Training deeper neural networks represents a large challenge, due to the vanishing gradient problem [Hoc98]. One possible solution to this problem is to add residual connections to your network's building blocks. ResNets [He+15] heavily rely on residual connections to build very deep neural networks, which are trainable on the ImageNet dataset. The variant with 50 layers, called ResNet-50, won the ILSVRC in 2015 with 94.7% top-5 accuracy. This network uses only 25.5 million weights and 3.9 billion MACs, hence it is more efficient than VGG-19.

The challenge was closed in 2017 after most entries had achieved over 95% top-5 accuracy [Rus+15b] and the challenge was considered as satisfyingly solved.

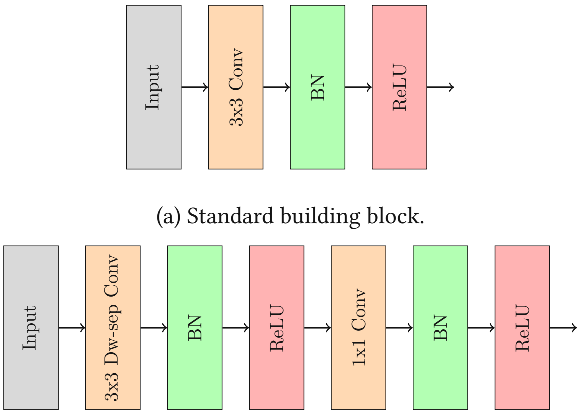
2.4 MobileNets

In the previous section it has been shown that modern CNNs may need a lot of MACs to achieve satisfying results. Therefore, researchers started to design CNN architectures explicitly to be run on mobile and embedded devices. MobileNets are arguably the most popular network family in this regard. They promise a good trade-off between computational costs and the accuracy of the network compared to other CNN architectures like ResNets or VGGNets. In this section their underlying ideas and their results shall be introduced and discussed.

So far, three versions of MobileNets have been introduced, each building upon the previous version and adding new ideas. One main idea borrowed from VGG or ResNet, is that the network consists of basic building blocks, which are placed sequentially inside the network.

2.4.1 Version 1

MobileNetV1 [How+17] was proposed in 2017 by a team of Google. It consists of VGG-style building blocks while reducing the parameters and MACs of the model tremendously by replacing usual convolutional layers with depthwise-separable convolutions and pointwise convolutions. The comparison with a block using a regular convolution is visualized in fig.2.10. It is worth mentioning, that each convolutional layer is followed by a batch normalization layer and a ReLU layer.



(a) Standard building block.

(b) Building block of MobileNetV1.

Figure 2.10: Comparison between the regular CNN building block and the one used by MobileNetV1.

The whole architecture is shown in tab.2.1. It was initially designed for the classification task on ImageNet. Therefore the architecture assumes an input size of 224x224x3, which is the standard image size for ImageNet, and final output size of 1000x1, the number of classes for ImageNet.

It was shown in the previous section, that the number of parameters gets reduced heavily by replacing regular convolutions with depthwise-separable convolutions. Therefore, it seems obvious, that the model will also lose descriptiveness. As a surprise, MobileNetV1 showed that replacing all regular convolutions by depthwise-separable convolutions only results in a 1% loss in top-1 accuracy (see tab.2.2). The assumption made for introducing depthwise-separable convolutions, that convolutional layers learn symmetric kernels, is therefore reinforced.

Two hyperparameters are defining the concrete structure of a MobileNetV1: width and resolution multiplier. The width multiplier α controls the number of feature maps in each layer. Number of feature maps for each layer are taken from tab.2.1 and multiplied by $\alpha \in (0, 1]$. Often, this hyperparameter is implemented in a way, that the number of feature maps is always divisible by 8, such that the layer fits nicely into the GPUs memory structure. The resolution multiplier ρ, as the name already suggests, controls the spatial resolution of the MobileNet layers. Again $\rho = 1$ equals the default architecture in tab.2.1 and $\rho \in (0, 1]$ for reducing the computational cost of the model. Usually, ρ is chosen in a way that the input resolution of the network is either 224, 192, 160, or 128.

2.4.2 Version 2

Version 2 of MobileNet was introduced in 2018 [San+19] and builds upon version 1 while being inspired by the ResNet architecture. Its building block uses two new concepts: bottlenecks and inverted residual connections.

Linear bottlenecks can be motivated by looking more closely into the meaning of the number of channels of a layer. Channels have two different responsibilities: to store information and to transform information. The idea is, that transforming information needs more dimensions than storing information. Hence, one can motivate architectures, where channel numbers of layers do not grow monotonously going further into the network, but rather grow and shrink alternately. MobileNetV2 is such an architecture.

They introduced the bottleneck layer, which in itself consists of several layers including three convolutional layers. The bottleneck layer is visualized in fig.2.11. In the first convolution, a 1x1 pointwise convolution, input feature maps get

Input	Operator	n_{OFM}	s
224x224x3	3x3 conv	32	2
112x112x32	3x3 dw conv	32	1
112x112x32	1x1 conv	64	1
112x112x64	3x3 dw conv	64	2
56x56x64	1x1 conv	128	1
56x56x128	3x3 dw conv	128	1
56x56x128	1x1 conv	128	1
56x56x128	3x3 dw conv	128	2
28x28x128	1x1 conv	256	1
28x28x256	3x3 dw conv	256	1
28x28x256	1x1 conv	256	1
28x28x256	3x3 dw conv	256	2
14x14x256	1x1 conv	512	1
5 × 14x14x512	3x3 dw conv	512	1
14x14x512	1x1 conv	512	1
14x14x512	3x3 dw conv	512	2
7x7x512	1x1 conv	1024	1
7x7x1024	3x3 dw conv	1024	1
7x7x1024	1x1 conv	1024	1
7x7x1024	7x7 pool	1024	1
1x1x1024	1x1 FC	k	1

Table 2.1: Architecture of MobileNetV1. Residual connections for blocks are applied whenever possible.

n_{OFM} ... channel number after projection
s ... stride of the depthwise-separable convolution
dw conv ... depthwise-separable convolution
FC ... fully-connected layer
k ... number of classes

	Parameters [M]	MACs [M]	ImageNet Accuracy [%]
MobileNetV1 (conv)	29.3	4866	71.7
MobiletNetV1 (dw conv)	4.2	569	70.6

Table 2.2: Comparison of MobileNetV1 using regular convolutions, compared to partly using depthwise separable convolutions [How+17].

expanded to higher dimensionality. Afterwards, these channels get transformed by a 3x3 depthwise-separable convolution, and the channel number again reduced by another 1x1 pointwise convolution, forming the bottleneck. The last projecting convolution is not followed by a ReLU, therefore this bottleneck is referred to as a linear bottleneck. Besides the usual strides parameter, which is implemented in the depthwise-separable convolution, one has also to specify the expansion factor for the first expanding convolution. For MobiletNetV2, this parameter is fixed at 6, meaning that if the input holds k feature maps, the first convolution expands them to $6 \cdot k$ feature maps, which get transformed by the depthwise-separably convolution.

The second concept added to MobileNetV1 is residual connections, which have already appeared before in sec.2.3. A residual connection connects the input of a layer directly with the output, skipping the layers of a block. In the case of MobileNetV2, the input of a residual block gets added to the output of the block. Its exact term is 'inverted residual connection' because the residual connections connect bottleneck layers and not expanded layers like it is usually done. The building block used by MobileNetV2 is shown in fig.2.12. Since input and output are simply summed up, their

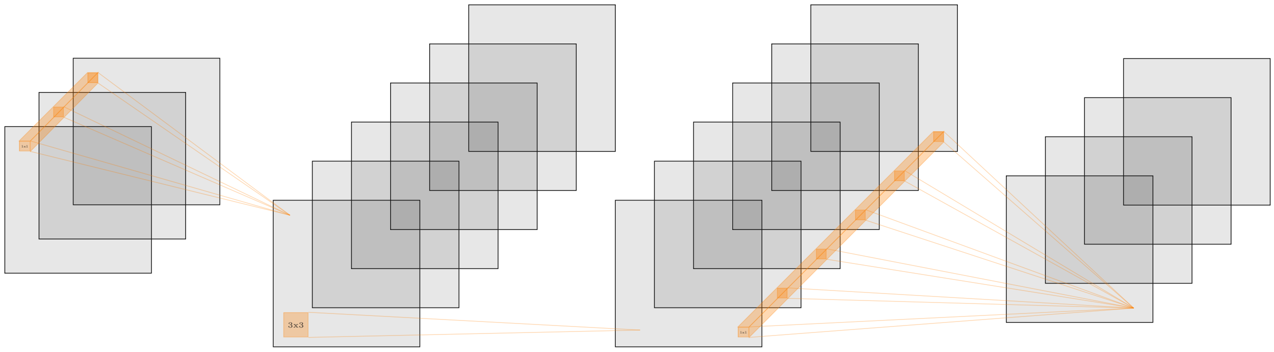


Figure 2.11: Working principle of a bottleneck layer.

dimensions must match. Therefore, residual connections can only be added when stride $= 1$ and $n_{\text{OFM}} = n_{\text{IFM}}$.

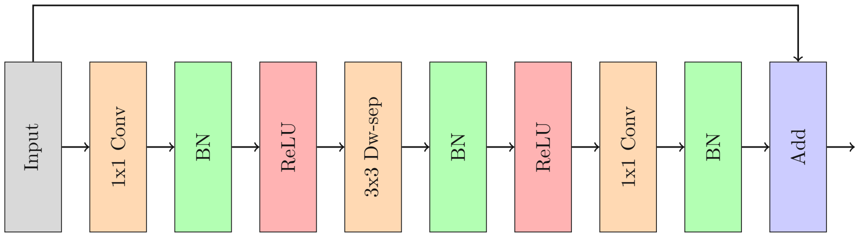


Figure 2.12: Building block for MobileNetV2. For stride $= 2$ blocks, the residual connection is not included.

Using these concepts, the default MobileNetV2 architecture can be defined (see tab.2.3). This model holds 3.4 million parameters and uses 300 million MACs. Again, the width and resolution parameters can be used to produce cheaper, smaller models.

Compared to MobileNetV1, the version 2 model uses around 20% fewer parameters and 47% fewer MACs. On paper MobileNetV2 is more efficient concerning MAC per parameter. Initially MobileNetV2 was again designed for ImageNet, but model variants for object detection and segmentation are also proposed in [San+19]. With a 72% top-1 accuracy on ImageNet, MobileNetV2 beats its predecessor by all metrics.

2.4.3 Version 3

The third version of the MobileNet was published in 2019 [How+19]. The authors added again some concepts, including squeeze-and-excitation models and more modern activation functions. This time the model's architecture is obtained by a network architecture search (NAS) on ImageNet.

Firstly, the new concepts shall be investigated. Squeeze-and-excitation (SE) modules [Hu+19] are generic additions to any CNN architecture. The idea behind them is to give the model the ability to weigh certain channels or regions of a feature map more for a certain input. A global pooling operation at the first layer of the module makes the module cheap to compute since all the following layers act on a spatial resolution of 1x1. The exact structure used for MobileNetV3 is shown in fig.2.14b. Note that n_{OFM} of the first 1x1 convolution is fixed to $n_{\text{IFM}}/4$ in the case of MobileNetV3.

MobileNetV3 also investiagtes the effect of using different activation functions besides the usual ReLU. One activation

Input	Operator	t	n_{OFM}	s
224x224x3	3x3 Conv	-	32	2
112x112x32	Block	1	16	1
112x112x16	Block	6	24	2
56x56x24	Block	6	24	1
56x56x24	Block	6	32	2
28x28x32	Block	6	32	1
28x28x32	Block	6	32	1
28x28x32	Block	6	64	2
14x14x64	Block	6	64	1
14x14x64	Block	6	64	1
14x14x64	Block	6	64	1
14x14x64	Block	6	96	1
14x14x96	Block	6	96	1
14x14x96	Block	6	96	1
14x14x96	Block	6	96	1
14x14x96	Block	6	160	2
7x7x160	Block	6	160	1
7x7x160	Block	6	160	1
7x7x160	Block	6	320	1
7x7x320	Block	6	1280	1
7x7x1280	Pool	-	1280	-
1x1x1280	1x1 Conv	-	k	-

Table 2.3: Architecture of MobileNetV2. Residual connections for blocks are applied whenever possible.

t ... expansion factor used for the bottleneck layer
n_{OFM} ... output channel number
s ... stride
k ... number of classes

function, which was proven to outperform ReLU, is swish [RZL17]. It is defined as:

$$\text{swish}(x) = x \cdot \sigma(x), \tag{2.15}$$

with $\sigma(x)$ being the sigmoid function. One reason, why ReLUs are preferred over sigmoid functions in MobileNets, is the rather high computational cost of the sigmoid function in comparison to the simpler ReLU. Luckily, there is an approximation for the sigmoid function available called hard sigmoid or h-sigmoid [CBD16a]:

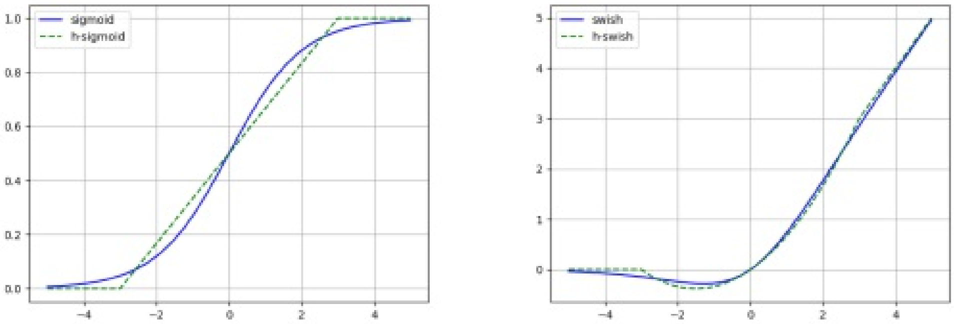
$$\text{h-sigmoid}(x) = \frac{\text{ReLU6}(x+3)}{6}. \tag{2.16}$$

ReLU6 is the usual ReLU limited at the maximum of 6. Analogously, a piecewise version of the swish can be defined [AV19] to reduce the computational cost of the swish function:

$$\text{h-swish}(x) = x \cdot \frac{\text{ReLU6}(x+3)}{6}. \tag{2.17}$$

A comparison between these activation functions and their hard counterpart is shown in fig.2.13. The h-swish activation function is still more expensive to compute than ReLU. Hence, MobileNetV3 decides to only use the more advanced activation function for layers deep into the network, where spatial resolution is lower and activation functions do not contribute as much to the model's computational costs compared to earlier layers.

Combining SE and new activation functions, a building block for MobileNetV3 can be defined. It is visualized in fig.2.14,

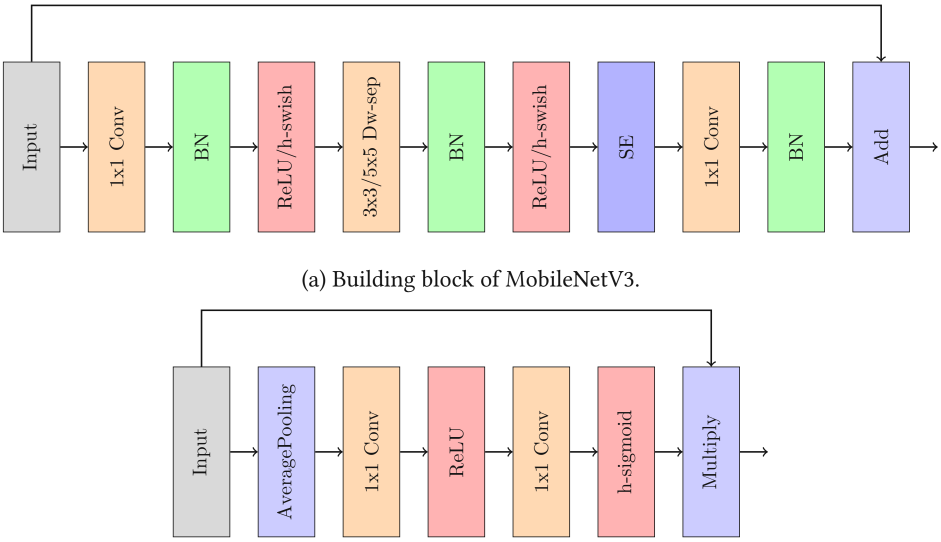


(a) Sigmoid - h-sigmoid.

(b) Swift - h-swift.

Figure 2.13: Comparison between activation functions and hard activation functions.

which also shows that this building block is not fixed, rather small variations are possible, like choosing between different activation functions or change the kernel size of the depthwise-separable convolution. The addition of the SE module is also optional.



(a) Building block of MobileNetV3.

(b) Building block of the squeeze-and-excitation module used for MobileNetV3.

Figure 2.14: Building block for MobileNetV3 and squeeze-and-excitation modules.

Network architecture search (NAS) [Tan+19; Yan+18] is called the process of optimizing neural network architectures using a heuristic procedure. MobileNetV3 uses two different approaches, which are explained in detail in [How+19]. They optimize kernel sizes, usage of SE modules, and the channel number of the bottleneck layers while keeping the trade-off between inference latency and model accuracy in mind. The authors also propose a more efficient classification head compared to the one used for MobileNetV2. In the end, they propose two different variants of MobileNetV3, both found using NAS: MobileNetV3-Small and MobileNetV3-Large (see tab.2.4). As seen in the previous version, these models can be modified through the width parameter, which is now called the depth multiplier. The resolution parameter got removed in this version.

The default MobileNetV3-Large uses 5.4 million parameters and 219 million MACs to achieve 75.2% top-1 accuracy on ImageNet. Its smaller counterpart, MobileNetV3-Small, uses only 2.5 million parameters and 56 million MACs to achieve 67.4% top-1 accuracy. Tab.2.5 summarizes parameters, MACs and ImageNet accuracy of all three MobileNet versions.

Input	Operator	n_{EFM}	n_{OFM}	SE	AF	s
224x224x3	3x3 conv	-	16	×	h-swish	2
112x112x16	3x3 block	16	16	×	ReLU	1
112x112x16	3x3 block	64	24	×	ReLU	2
56x56x24	3x3 block	72	24	×	ReLU	1
56x56x24	5x5 block	72	40	✓	ReLU	2
28x28x40	5x5 block	120	40	✓	ReLU	1
28x28x40	5x5 block	120	40	✓	ReLU	1
28x28x40	3x3 block	240	80	×	h-swish	2
14x14x80	3x3 block	200	80	×	h-swish	1
14x14x80	3x3 block	184	80	×	h-swish	1
14x14x80	3x3 block	184	80	×	h-swish	1
14x14x80	3x3 block	480	112	✓	h-swish	1
14x14x112	5x5 block	672	160	✓	h-swish	2
7x7x160	5x5 block	960	160	✓	h-swish	1
7x7x160	5x5 block	960	160	✓	h-swish	1
7x7x160	1x1 conv	-	960	×	h-swish	1
7x7x960	7x7 pool	-	960	×	-	1
1x1x960	1x1 conv*	-	1280	×	h-swish	1
1x1x1280	1x1 conv*	-	k	×	-	1

Input	Operator	n_{EFM}	n_{OFM}	SE	AF	s
224x224x3	3x3 conv	-	16	×	h-swish	2
112x112x16	3x3 block	16	16	✓	ReLU	2
56x56x16	3x3 block	72	24	×	ReLU	2
28x28x24	3x3 block	88	24	×	ReLU	1
28x28x24	5x5 block	96	40	✓	h-swish	2
14x14x40	5x5 block	240	40	✓	h-swish	1
14x14x40	5x5 block	240	40	✓	h-swish	1
14x14x40	5x5 block	120	48	✓	h-swish	1
14x14x48	5x5 block	144	48	✓	h-swish	1
14x14x48	5x5 block	288	96	✓	h-swish	2
7x7x96	5x5 block	576	96	✓	h-swish	1
7x7x96	5x5 block	576	96	✓	h-swish	1
7x7x96	1x1 conv	-	576	✓	h-swish	1
7x7x576	7x7 pool	-	576	×	-	1
1x1x576	1x1 conv*	-	1024	×	h-swish	1
1x1x1024	1x1 conv*	-	k	×	-	1

Table 2.4: Architecture of MobileNetV3-Large (left) and MobileNetV3-Small (right). Residual connections for blocks are applied whenever possible.

n_{EFM} ... channel number after expansion
n_{OFM} ... channel number after projection
SE ... squeeze-and-excitation
AF ... activation function
s ... stride of the depthwise-separable convolution
conv* ... convolutional layer without batch normalization
k ... number of classes

While MobileNetV1 had still used a lot of MACs to produce its results, MobileNetV2 almost halved them while improving accuracy using bottleneck layers. MobileNetV3 can utilize more parameters through easily to compute SE models, while further reduce MACs of the MobileNet. The MobileNetV3-Small variant takes 1/10 of MACs of MobileNetV1, while only losing around 3% accuracy on ImageNet. Since the final goal of this work is to bring powerful CNNs to embedded hardware, MobileNetV3-Small looks like the perfect candidate for that.

	Parameters [M]	MACs [M]	ImageNet Accuracy [%]
MobiletNetV1	4.2	569	70.6
MobiletNetV2	3.4	300	72.0
MobiletNetV3-Large	5.4	219	75.2
MobiletNetV3-Small	2.5	56	67.4

Table 2.5: Summarized results of the three MobileNet versions.

The MobileNetV3 paper [How+19] also proposes interesting architectures for object detection and semantic segmentation models. The low latency model for semantic segmentation will be discussed in detail in sec.2.7.

2.5 CNNs as Feature Extractors: Transfer Learning

CNNs are often referred to as feature extractors [Her+17], meaning that they learn to identify interesting features inside an image, like edges, corners, or certain shapes. The idea that CNNs learn to extract generic features independent of the dataset is proven by the well-established method of deep transfer learning [HBF18; Tan+18; KSL19]. Transfer learning is about transferring learned knowledge from one task or dataset to another one. This method is very common when working with deep CNNs where training can be hard and expensive.

The thought process is as follows: Since a model would learn to extract generic features anyway, independent of the dataset, one can use a pre-trained model as a starting point for a new task. Note, that a really small and simple dataset will not produce a strong feature extractor. Models, which are pre-trained on ImageNet, are often used for transfer learning [HBF18], since ImageNet is a very rich and complex dataset. It was also shown, that models, which perform well on ImageNet, are good candidates for transfer learning [KSL19].

When transferring knowledge from ImageNet to another classification task, which holds a different number of classes than ImageNet, one has to change the top layer of the model, since this layer represents the number of classes. It is also possible to reuse features learned on a classification task for object detection or semantic segmentation. In this case, one may have to delete additional layers on top of the model, dependent on the model architecture. Usually, there exists a point within the network architecture, which is considered to be the end of the feature extractor.

Pre-trained models can be used as a starting point for new tasks. The top layers of the model, which were necessary to be replaced, have to be trained from scratch. The danger of destroying previously learned knowledge, which would be valuable for the new task, also has to be kept in mind. There are different approaches available to solve this problem. One possibility is freezing layers, which are already satisfying pre-trained. This means, that their weights do not get updated during training, therefore their initial state is kept. Another way is using very low learning rates to not destroy previously gained knowledge.

2.6 Knowledge Distillation

Another form of knowledge transfer is from one model to another one. Most of the time it is desired to transfer the learned knowledge from a big model to a smaller one. This case is known as knowledge distillation [HVD15]. Knowledge distillation comes in many forms and variations, a survey including some theoretical work can be seen in [Gou+21]. In this section the basic idea of knowledge distillation should be presented and explained through two example algorithms.

The setup of knowledge distillation is the following: one big model, which got trained on a certain task, shall be replaced by a smaller model. Of course, it is desired, that the smaller model performs as well as possible, ideally as good as the big model. Training the small model from scratch like usual often does not achieve this, resulting in a drop in accuracy. Knowledge distillation now uses the big model to help training the small model by extracting some sort of knowledge from the big model. During the training of the small model, an additional loss formed by the extracted knowledge is added to the default task loss. Due to this setup, the big model is called 'teacher' and the small model 'student'.

2.6.1 Dark Knowledge

'Dark knowledge' was the first proposed method for knowledge distillation, which is designed for classification tasks. The idea is to take learned hidden knowledge from the teacher model using a different variation of the softmax activation function and transfer it to the student model.

An image classification network has always a softmax activation function at the top to produce the final predictions.

This activation function takes a vector with a size equal to the number of classes and transforms it to have length one (see eq.2.18). Due to this normalization, the vector can be interpreted as a vector of probabilities and the element with the highest value corresponds to the predicted class. In this way, one also gets a measurement of how confident the model is in its prediction.

$$\sigma_{softmax}(\vec{x}) = \frac{1}{\sum_i^k e^{x_i}} \begin{pmatrix} e^{x_0} \\ e^{x_1} \\ \vdots \\ e^{x_{k-1}} \\ e^{x_k} \end{pmatrix} \tag{2.18}$$

The usual softmax activation function tries to make the prediction as confident as possible, meaning that one element is much greater than the other ones. This property can be changed, by adding the parameter T, called temperature, to the original equation:

$$\sigma_{softened}(\vec{x}) = \frac{1}{\sum_i^k e^{x_i/T}} \begin{pmatrix} e^{x_0/T} \\ e^{x_1/T} \\ \vdots \\ e^{x_{k-1}/T} \\ e^{x_k/T} \end{pmatrix}. \tag{2.19}$$

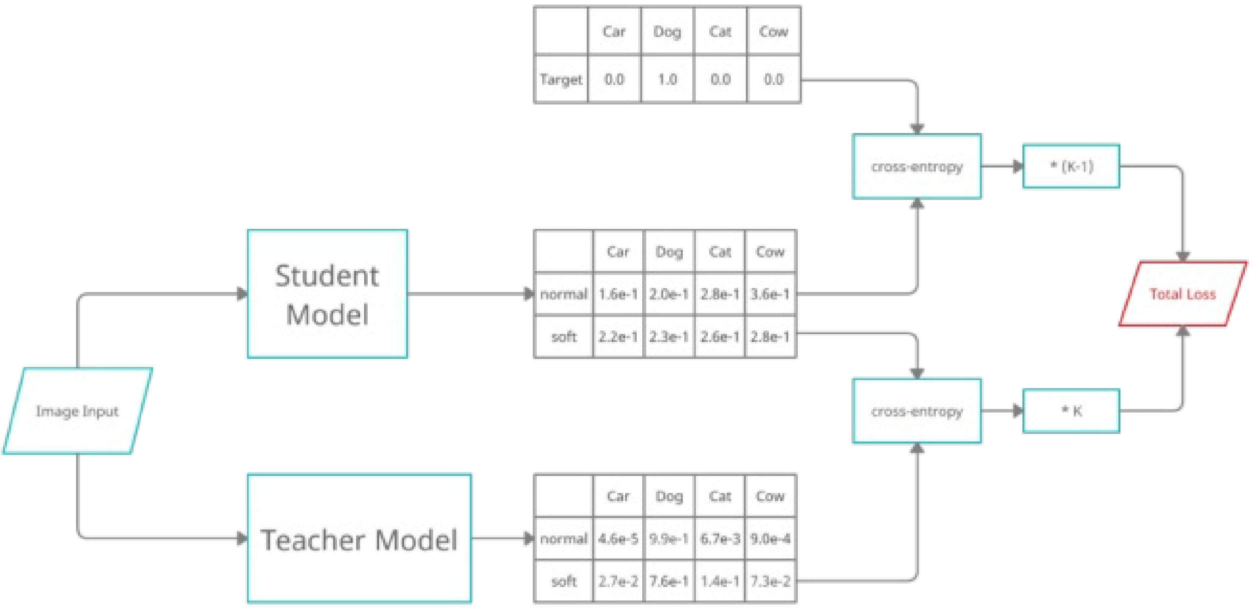
This produces softened-up predictions, where similarities between classes are more visible. For example, when classifying animals, a wolf and a dog are more similar than a dog and an elephant. This kind of information is not stored in the ground truth labels, but a good model can produce this kind of information using the softened softmax activation function.

The hidden information of similarities between classes is the dark knowledge, which should be transferred from the teacher to the student model. The whole workflow is visualized in fig.2.15. For a training sample, the produced softened predictions of the teacher and student model are compared and the cross-entropy loss between them are computed. This loss is weighted by a hyperparameter κ and added to the default task loss to form the final loss, which should be minimized.

2.6.2 Adaptive Cross-Entropy

Adaptive cross-entropy [PH20] is conceptually similar to dark knowledge, but is targeted to semantic segmentation tasks. Instead of calculating the cross-entropy loss of two different probability vectors, the target probability map itself gets build up from the ground truth labels and the teachers predictions. The workflow is visualized in fig.2.16. The final probability map $P(x, y)$ is defined as:

$$P(x,y) = \begin{cases} \kappa \cdot p_t(x,y) + (1-\kappa) \cdot p_{gt}(x,y), & \text{if } (x,y) \text{ is predicted correctly} \\ p_{gt}(x,y), & \text{otherwise} \end{cases} \tag{2.20}$$



	Car	Dog	Cat	Cow
Target	0.0	1.0	0.0	0.0

	Car	Dog	Cat	Cow
normal	1.6e-1	2.0e-1	2.8e-1	3.6e-1
soft	2.2e-1	2.3e-1	2.6e-1	2.8e-1

	Car	Dog	Cat	Cow
normal	4.6e-5	9.9e-1	6.7e-3	9.0e-4
soft	2.7e-2	7.6e-1	1.4e-1	7.3e-2

Figure 2.15: Conceptional structure of the dark knowledge method.

where p_t are the teacher's predictions and p_{gt} are the ground truth labels. κ is a hyperparamter controlling the strength of distillation. As the equation shows, the teacher's prediction of a certain pixel only gets added to the ground truth label if the teacher predicted this pixel correctly. This makes sense, since it is not desired to teach the student model wrong information.

2.7 CNNs for Semancic Segmentation: the DeeplabV3+ Architecture

There exist many different CNN architecture for semantic segmentation [Min+20]. One family of models called Deeplab [Che+16; Che+17b; Che+17c; Che+18] utilize dilated convolutions and special segmentation heads to produce the target segmentation mask. Deeplab architectures use other CNN models as feature extractors, as described in the sec.2.5, and build the segmentation mask from these features afterwards.

DeeplabV3 [Che+17c] shows that downsampling the image too much in the feature extractor, harms accuracy. Therefore, they propose to not downsample the image using stride 2 layers, but rather replace those layers with atrous convolutions while doubling their dilation rates. Fig.2.17 shows that a depthwise-separable atrous convolution has its kernel stretched out according to its dilation rate. A convolution using a stride of 2 produces the same receptive field as using dilation rates of 2 for all following convolutions while preserving the spatial resolution of the image. Therefore it is conceptually right to replace stride 2 layers with atrous convolutions. Looking at MoblieNets (see sec.2.4), they have a default output stride (OS) of 32, hence extracted feature maps are downsampled by a factor of 32. DeeplabV3 shows, that it can be beneficial, to set the OS to 16 or 8, changing the depthwise-separable convolutions dilation rates to match the receptive field of the original model.

After extracting features from the input image, the segmentation mask has to be built. DeeplabV3 uses the atrous spatial pyramid pooling (ASPP) module for that to collect semantic information on multiple scales. Different operations act on the extracted features and build one single output through concatenation. A 1x1 pointwise convolution builds the final logistic layer, which is upsampled to the original resolution to represent the final segmentation mask. The ASPP

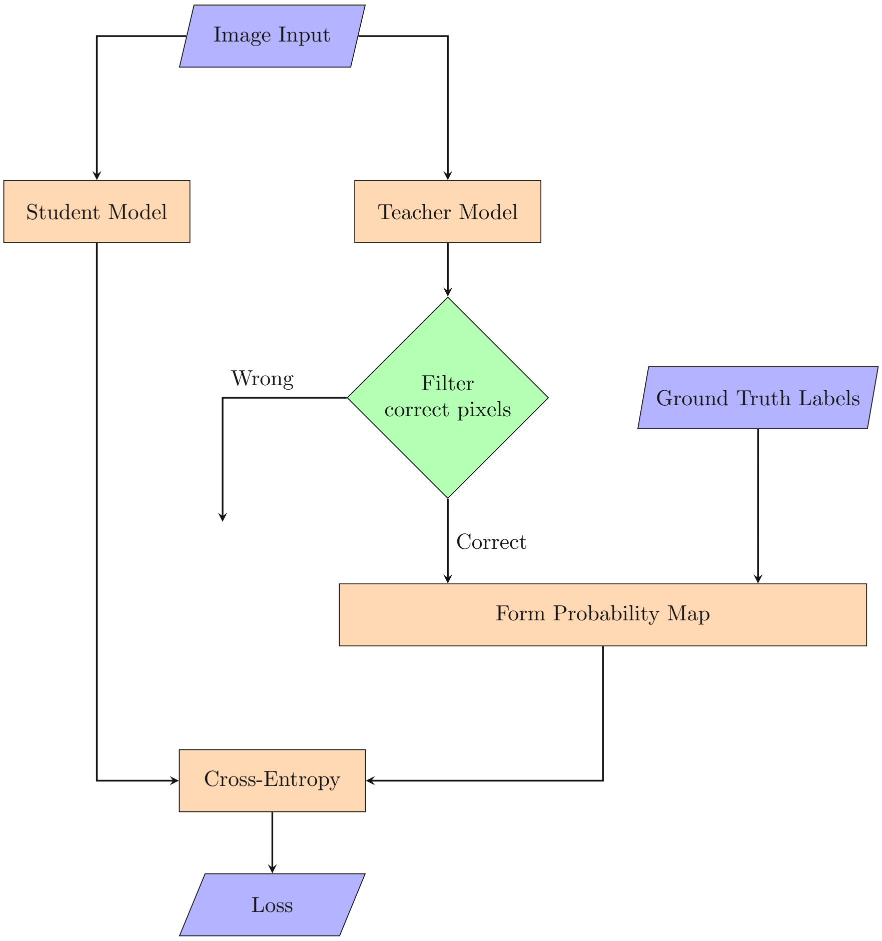


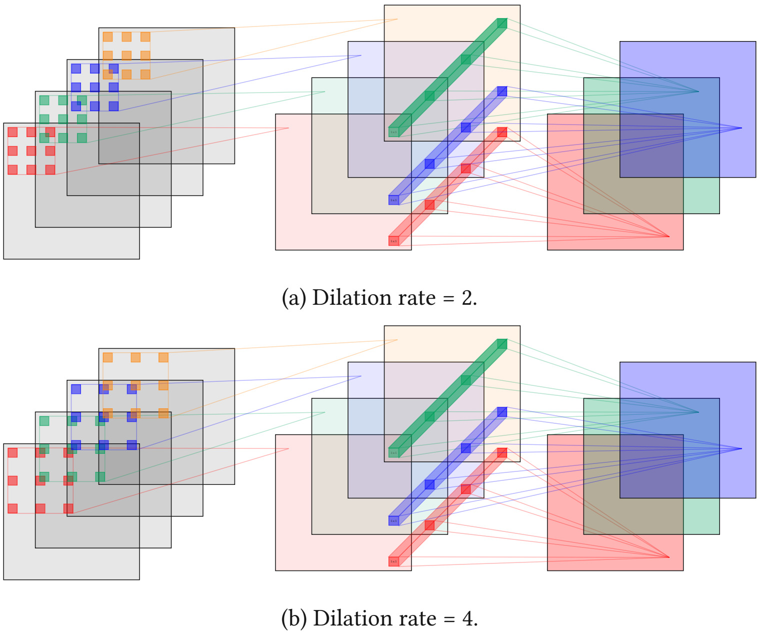
Figure 2.16: Conceptional structure of the ACE loss.

module consists of a global average pooling operation, a 1x1 pointwise convolution, and several atrous convolutions using different dilation rates. The module used by [Che+17c] is visualized in fig.2.18.

The most recent architecture, DeeplabV3+, combines the ideas of encoder-decoder architectures with the ASPP module. Fig.2.19 shows the full architecture. Low-level features from the feature extractor get fed into a pointwise convolution. Afterwards, these features get concatenated with the upsampled features produced from the ASPP module. After a convolution and another upsampling layer, the original image resolution is restored and the final segmentation mask is formed.

2.7.1 MobileNetV3's version of DeeplabV3+

Computing all branches of the ASPP module can be quite expensive, what represents a problem, when one wants to use it in a low-resource setting. Therefore, there was already a reduced version, R-ASPP, proposed for MobileNetV2 [San+19], which knocks out the more expensive atrous convolutions from the module. MobileNetV3 [How+19] improved this module again, by concatenating the two branches in a squeeze-and-excitation manner (see fig.2.20). This module is called lite R-ASPP or LR-ASPP. The image pooling branch uses an average pooling layer with large kernel and stride values, which output gets upsampled again after transformation through a pointwise convolution to its original size. This output gets multiplied to the other branch, acting like a squeeze-and-excitation module. As in the DeeplabV3+ model, low-level features from the feature extractor get transferred directly to the segmentation head.



(a) Dilation rate = 2.

(b) Dilation rate = 4.

Figure 2.17: Working principle of depthwise-separable convolutions with different dilation rates.

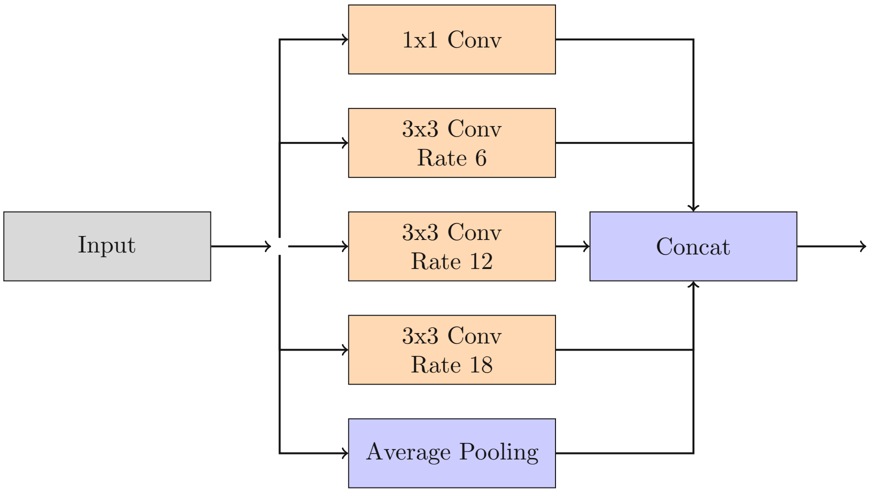


Figure 2.18: ASPP module introduced in [Che+17c] for a feature extractor using OS= 16.

2.8 Neural Network Compression

Neural network compression is an important step when deploying CNNs on resource-limited devices like CPUs, mobile phones ore embedded systems. The goal is to compress a given network to reduce its memory requirements and/or inference latency. According to [Che+20], compression techniques belong to one of four categories: parameter pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation. Tab.2.6 gives a quick overview of the four categories, while this section should introduce basic ideas of the first category because pruning and quantization are well established for applications using embedded devices.

Network pruning describes the method of deleting non-informative neurons to compress the model [SB15]. Some measure of the sensitivity of each neuron has to be defined, which can be supervised or non-supervised. Model size can be reduced by setting the weights of pruned neurons to 0. This does not guarantee a latency speed-up as well. For this, the computation graph has to be modified to exclude pruned neutrons explicitly. Dependent on the target hardware, this can

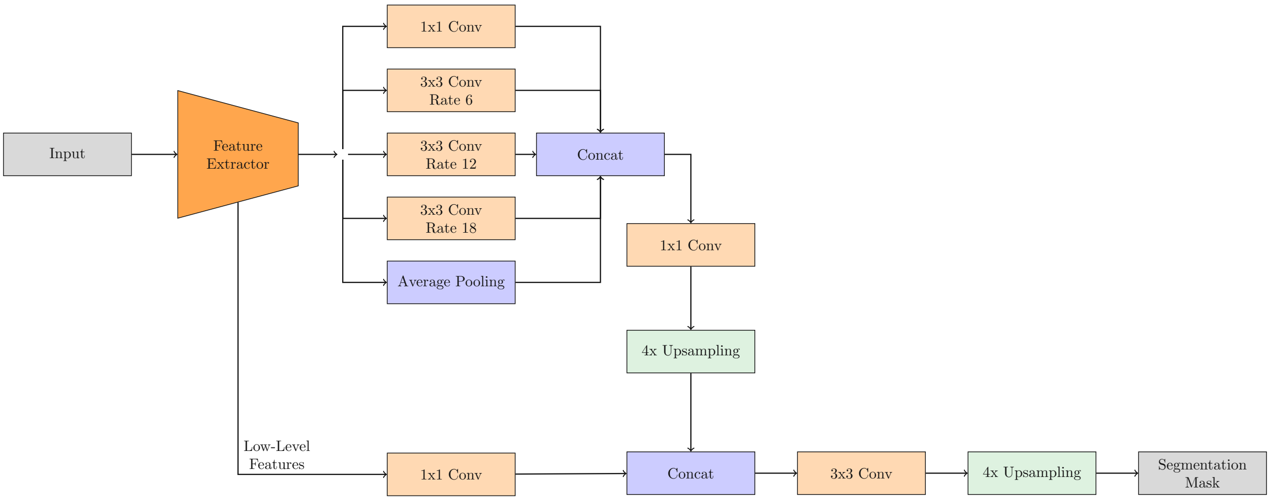


Figure 2.19: Full architecture of the DeeplabV3+ model.

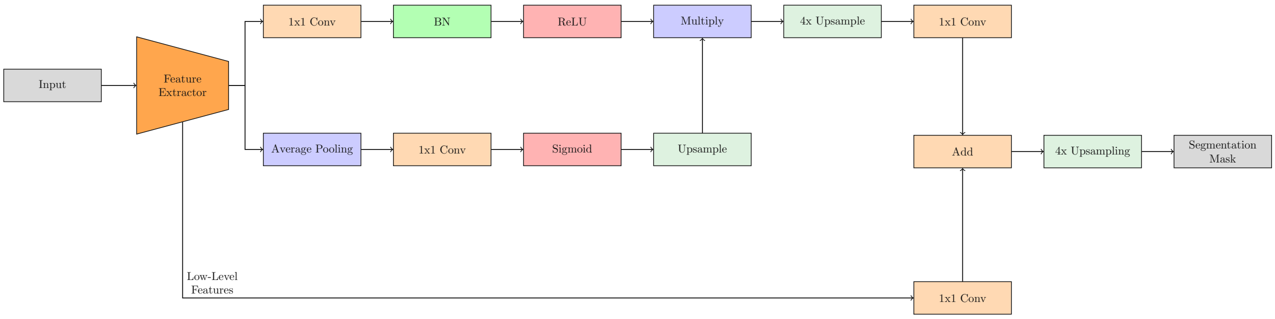


Figure 2.20: Segmentation head used for the MobileNetV3.

be a non-trivial task, although vendor libraries may support some sort of pruning for its specific hardware[1]. Pruning algorithms are capable of compressing MobileNets trained on CIFAR10 by 30% without loosing accuracy [Zhu+19].

Quantization [Gon+14; VSM11] does not rely on deleting weights, but rather stores them with another data type with less resolution to save memory and speed up inference. Usually, weights are stored in single-precision (FP32), meaning 32 bits. The model size can be almost halved by quantizing them to half-precision (FP16). The latency speed-up depends strongly on the hardware since not all architectures can work efficiently with FP16 numbers. Quantization often requires fine-tuning of the quantized model, since the weights and therefore the output of the model slightly changes. The extreme case of quantization is a binary network, where weights are represented with a single bit [CBD16b]. The difference between floating point inference and integer inference is reported to be around 25-30% in the case of MobileNets on the Pixel phones [Jac+18].

To compare these different approaches, quantitative criteria are necessary. One can quantify compression by comparing the number of parameters of the original (M) and the compressed model (M*). This may be suitable for pruning, but quantizing a network does not remove any parameters at all. When dealing with a memory bounded system, the relative rate of memory compression may be the most fitting. Most embedded hardware work in a computational bounded regime, making the relative change of compressed MAccs a reasonable choice:

$$\delta(M, M*) = \frac{\text{MAccs}_M - \text{MAccs}_{M*}}{\text{MAccs}_M} \cdot 100\%. \tag{2.21}$$

[1] f.e. IntelLabs' Distiller: https://github.com/IntelLabs/distiller

Category	Description	Details
Parameter pruning and quantization	Reducing redundant parameters which are not sensitive to the performance	Robust to various settings, can achieve good performance, can support both train from scratch and pre-trained model
Low-rank factorization	Using matrix/tensor decomposition to estimate the informative parameters	Standardized pipeline, easily to be implemented, can support both train from scratch and pre-trained model
Transferred/compact convolutional filters	Designing special structural convolutional filters to save parameters	Algorithms are dependent on applications, usually achieve good performance, only support train from scratch
Knowledge distillation	Training a compact neural network with distilled knowledge of a large model	Model performances are sensitive to applications and network structure only support train from scratch

Table 2.6: Overview over neural network compression categories proposed by [Che+20].

The relative change δ will be further equated as compression rate.

As the final goal of neural network compression is to speed up the inference latency of a mode, the relative speed-up of inference times will also be investigated in this work:

$$\delta_t(M, M*) = \frac{t_M - t_M*}{t_M} \cdot 100\%. \tag{2.22}$$

2.9 Shunt Connections

Shunt connections are another way to compress CNNs, which has not been explored much. They got introduced in 2019 [STA19] with no other work building upon the original paper. This method can replace a big part of a model with a smaller CNN. Therefore it holds great potential for compressing parameters and MACs of your model tremendously. This section should motivate the idea behind shunt connections, show the basic workflow to compress a residual CNN by describing the work done in the original paper, and formulate open questions regarding the state of the art.

2.9.1 Motivating Shunt Connections for Residual CNNs

To motivate shunt connections, one has to look at how residual CNNs work in more detail. Due to the skip connections around blocks, information can travel on different paths in the network. It has been shown, that short paths are much more important for the network's performance than long paths [VWB16]. This has been proven by deleting a single residual block of the network and observing, that its accuracy does not change much for most of the blocks. Removing a single block removes the longest path from the network, the one which goes through all blocks, but only a few short ones.

This means, that some blocks of a residual CNN are only responsible for transforming a small amount of data in a performance-critical way. This can be interpreted as redundancy inside the network, which could be compressed since the parameters of such blocks are not used in an optimized way. A block, which is not crucial for the performance of the network, is denoted as vulnerable as in [STA19].

Using this implementation, one can try to find a series of vulnerable blocks and try to compress them. It was shown, that short paths hold more knowledge than long paths. Therefore, it should be possible to exchange multiple blocks with fewer blocks. This is exactly the idea of shunt connections: replacing multiple computational expensive blocks with a single block called shunt connection.

2.9.2 Quantifying Compression Potential with Knowledge Quotients

A block not hold much knowledge, is denoted as vulnerable, meaning a high potential for compression. This qualitative statement can be quantified by calculating the knowledge quotients for each block [STA19]. The knowledge quotient KQ of block B is defined as:

$$\mathrm{KQ}_B = \frac{\mathrm{acc}_{\text{original}} - \mathrm{acc}_{\text{w}\backslash\text{o B}}}{\mathrm{acc}_{\text{original}}}, \tag{2.23}$$

while $\mathrm{acc}_{\text{original}}$ is the accuracy of your residual CNN on your task and $\mathrm{acc}_{\text{w}\backslash\text{o B}}$ is the accuracy when removing block B from the model. When $\mathrm{KQ}_B = 1$, it means that $\mathrm{acc}_{\text{w}\backslash\text{o B}}$ is close to zero, corresponding to a small potential of compression. On the other hand, $\mathrm{KQ}_B \approx 0$ means that the accuracy has not changed when deleting the block, corresponding to much redundancy of the block and high compression potential. Therefore, the goal is to find a series of blocks with low knowledge quotients, which can be replaced with a shunt connection.

2.9.3 Shunt Architectures

Shunt connections are also CNNs, meaning that a concrete architecture has to be defined. The design space for CNNs is really big, as it has been already seen in sec.2.3. Hence, it is necessary to make reasonable assumptions about the problem and look for architectures solving similar problems.

Shunt connections have to solve the task called feature matching. Their input and output consist of feature maps, which may have a different number of channels and/or a different spatial resolution. This task is similar to semantic segmentation, where multiple feature maps have to be produced as well. Encoder-decoder architectures were proposed for semantic segmentation [Yas18] and therefore could also fit well as a basic architecture for shunt connections.

The final goal of the whole procedure is to speed up the computationally heavy model. Hence, shunt connections should be as lightweight as possible to achieve the highest speed-up. MobileNet's (see sec.2.4) have proven to be highly efficient when looking at the number of MACs necessary for each parameter. Therefore, it makes sense to use the building blocks of these networks to build up shunt connections.

The original paper about shunt connections [STA19] proposes five different architectures following these two ideas. Building blocks, which are very similar to the ones of MobileNetV2 (see sec.2.4.2), are used. Hence, the encoder-decoder structure of MobileNetV2's bottleneck layers is inherited. The five architectures differ in the number of blocks and number of channels, resulting in models with different sizes which should be suitable for various problem sizes. The proposed architectures for inputs with 64 channels and outputs with 96 channels are shown in tab.2.7. The spatial resolution of input and output feature maps are the same.

2.9.4 Training of Shunt Connections

Another important aspect when using CNNs is how to train them. In contrast to semantic segmentation, there are no labels or classes defined, when doing feature matching. A regression loss is therefore most fitting. The standard

Operator	Arch 1		Arch 2		Arch 3		Arch 4		Arch 5	
	n_{IFM}	n_{OFM}	n_{IFM}	n_{OFM}	n_{IFM}	n_{OFM}	n_{IFM}	n_{OFM}	n_{IFM}	n_{OFM}
1x1 Conv	64	192	64	128	64	64	64	128	64	192
BN	192	192	128	128	64	64	128	128	192	192
ReLU6	192	192	128	128	64	64	128	128	192	192
3x3 Dw Conv	192	192	128	128	64	64	128	128	192	192
BN	192	192	128	128	64	64	128	128	192	192
ReLU6	192	192	128	128	64	64	128	128	192	192
1x1 Conv	192	64	128	64	64	64	128	96	192	64
BN	64	64	64	64	64	64	96	96	64	64
1x1 Conv	64	192	64	128	64	128			64	192
BN	192	192	128	128	128	128			192	192
ReLU6	192	192	128	128	128	128			192	192
3x3 Dw Conv	192	192	128	128	128	128			192	192
BN	192	192	128	128	128	128			192	192
ReLU6	192	192	128	128	128	128			192	192
1x1 Conv	192	96	128	96	128	96			192	64
BN	96	96	96	96	96	96			64	64
1x1 Conv									64	192
BN									192	192
ReLU6									192	192
3x3 Dw Conv									192	192
BN									192	192
ReLU6									192	192
1x1 Conv									192	96
BN									96	96

Table 2.7: Proposed architectures for shunt connections.
n_{IFM} ... input channel number
n_{OFM} ... output channel number

mean-squared-error (MSE) can be used to measure the accuracy for each input feature map:

$$\text{MSE} = \sum_{i=0}^{n} \sum_{x,y} \left(F_{\text{target}}^{i}(x,y) - F_{\text{out}}^{i}(x,y) \right)^2 , \tag{2.24}$$

where n is the number of channels, F_{target} the target feature maps, which the shunt connector should replicate and F_{out} the output feature maps of the shunt. The original paper proposes to use the Adam optimizer [KB17] to solve this minimization problem.

The data for this training procedure can be extracted by forward passing through the original network and saving the feature maps of the corresponding layers. How to do this efficiently and simply in Keras is described in sec.4.3.1.

Assuming, the training has converged successfully, the shunt connection will reproduce the behaviour of the blocks of the original model. Now, the shunt connection can be inserted into the original model, replacing the original blocks and forming the shunt-inserted model. When evaluating the shunt-inserted model on the original task, one will see that this model has pretty high accuracy, indicating that the training of the shunt connection was successful. Although, original accuracy will not be achieved most likely, since some error remains while training the shunt.

Therefore, an additional step is used to overcome this problem: fine-tuning the shunt-inserted model. In this step, the shunt-inserted model is trained on the original dataset. It is not further explained in the original work, but the name indicates using low learning rates or freezing layers like it was described in sec.2.5.

One can now try to categorize shunt connections according to the list in sec.2.8. They do fit quite nicely into the category of knowledge distillation since a large model is used to train a smaller one.

2.9.5 Results and Limits of Original Paper

The original paper [STA19] tested their shunt connection workflow on MobileNetV2, which was trained on four different datasets: both CIFAR [Kri12] datasets and Caltech [Li+13; GHP07] datasets. All five architectures described in the previous section were used to compress the original model.

First, the datasets shall be briefly described. The CIFAR datasets were already mentioned in sec.2.1.1. There are two variants: CIFAR10 and CIFAR100, which hold 10 or 100 different balanced classes accordingly. Both datasets consist of 60000 RGB images with a spatial resolution of 32x32. Caltech101 and Caltech256, forming the family of Caltech datasets, consist of bigger RGB images with a resolution of 300x200 pixels. Caltech101 classifies 9144 images into 102 unbalanced classes, while Caltech256 has 30608 images with 257 unbalanced classes.

The default MobileNetV2 model is trained on these datasets and knowledge quotients calculated to chose the optimal shunt location. Unfortunately, the knowledge quotients are not stated in the paper, but by looking at the number of input and output channels, one can guess the exact location of the shunt connection. We know through Table1 of the paper, that the shunt connections have 64 input channels and 96 output channels. Comparing these numbers with the layers in tab.2.3, it can be concluded that blocks 7-14 are in the range of possibility to be replaced. The exact location cannot be derived, even when looking at the compressed FLOPs for a given architecture, because the exact model structure of the uncompressed model is not known.

Nevertheless, the results obtained by the authors shall now be discussed, which are summarized in tab.2.8. Independent of the dataset, they achieve a relative compression of FLOPS of 32-34% dependent on the used shunt architecture. Since the naming of shunt architecture is rather confusing the ascending ordering in relation to MACs is highlighted again:

$$\text{Arch 4} < \text{Arch 3} < \text{Arch 2} < \text{Arch 1} < \text{Arch 5}.$$

Keeping this in mind, the results presented in tab.2.8 can be understood in more depth. Looking at the results for CIFAR10, one can see that shunt-inserted models are not capable to reach the accuracy of 91.28% of the original model. Therefore, the fine-tuning step got introduced to boost the accuracy of the resulting models. All architectures achieve almost original accuracy, while Arch 1 is even outperforming the original model. Another important insight is, that a bigger shunt with more parameters does not yield better results necessarily. Arch 5's accuracy, which is smaller than Arch 1's, supports this claim.

For CIFAR100, it looks like a bigger shunt shall be favoured here to achieve the highest fine-tuned accuracy. Again, the original model is outperformed.

The Caltech datasets get outperformed by all shunt architectures, after the fine-tuning step. For these datasets, the choice of architecture does not have a great effect on accuracy. Hence, the smallest shunt, Arch 4, should be preferred. This result is quite astonishing: it is possible to compress the network by almost a third, while boosting its accuracy.

			Arch 1	Arch 2	Arch 3	Arch 4	Arch 5
Compression of FLOPs			32.4%	32.8%	33.2%	33.5%	30.3%
Dataset	Original Model Acc [%]	Acc with shunt connection [%]					
CIFAR10	91.28	Shunt-inserted	87.61	85.83	83.7	72.4	90.0
		Fine-tuned	91.42	90.8	90.9	91.0	91.1
CIFAR100	68.3	Shunt-inserted	65.7	62.3	60.0	55.8	66.9
		Fine-tuned	68.2	68.3	68.0	67.6	68.9
Caltech101	65.4	Shunt-inserted	63.48	63.48	63.51	59.82	65.4
		Fine-tuned	68.15	68.31	68.1	68.3	68.1
Caltech256	40.0	Shunt-inserted	37.0	35.2	34.5	29.8	38.2
		Fine-tuned	42.4	41.9	41.2	42.1	42.3

Table 2.8: Results of the original shunt connection paper.

The fact, that a smaller model with fewer parameters performs better on a certain task, indicates that the task leads to overfitting. The original paper also provides its learning curves of the training of the initial, original models. The curves for all four datasets show at least some overfitting, since the MobileNetV2 model reaches around 100% accuracy on the train set, but performs much worse on the test set. This is expected when training MobileNetV2, which is designed for the much more complex ImageNet dataset in comparison to CIFAR and Caltech.

It can be concluded, that shunt connections work very well for simple tasks, which tend to overfit for a given model. It looks like, that the redundancy occurring by overfitting can be well compressed by shunt connections, reaching results above the accuracy of the original model. However, it is not known, how shunt connections behave for tasks and models, which do not tend to overfit. The real challenge of compression is to compress the knowledge of a model and not only get rid of its obvious redundancies.

A second conclusion is, that fine-tuning the shunt-inserted model on the original dataset is indeed required to meet the accuracy of the original model. Unfortunately, this step is not further explained in the paper, so readers can only guess, how the authors did it exactly. The name indicates training with small learning rates and freezing some layers potentially. This step may also depend on the dataset or shunt architecture, making it even more complicated.

The paper also shows results for inserted two shunt connections, which are not covered in this summary. A real-world scenario, where a drone is taking pictures of animals and classifies them, is also provided. For this example, the latency speed-up of the network was assumed to scale linearly with reduced MACs. This may be true for serial hardware, like a CPU, but it is not that simple for parallel architectures like GPUs. Embedded devices, f.e. NVIDIA Jetson devices (see sec.3.5.1), have an onboard GPU and are well suitable for drone applications. In conclusion, the assumption of linear speed-up does not hold on to those devices. Hence, the real speed-up of compression through shunt connections on modern embedded hardware is still an open question, which this thesis tries to answer.

Chapter 3

Methodology

The goal of this work is to expand shunt connections for the MobileNetV3 model and semantic segmentation tasks. For this, the state-of-the-art regarding shunt connections has to be established on MobileNetV2 using either the CIFAR or Caltech datasets. Afterwards, the same methods are applied to the MobileNetV3 classification and semantic segmentation model. For the semantic segmentation tasks, the Cityscapes dataset is used. The main deep-learning framework used for this work is Keras, hence the CIFAR datasets are chosen for classification tasks, since they are obtainable through Keras itself. While the next chapter lays out the aspects of the Keras implementation, this chapter should describe the basic methodology necessary for carrying out the shunt connection experiments. Remember the general workflow of inserting shunt connections:

1. Train original model
2. Calculate knowledge quotients
3. Choose shunt location
4. Extract feature maps of original model at shunt location
5. Chose shunt architecture
6. Train shunt architecture on feature maps
7. Fine-tune shunt-inserted model

In multiple steps, the training and evaluation of CNN models is required. The datasets used in this thesis are described regarding their properties and pre-processing steps, while basic procedure of using training, validation and test splits is outlined as well. The exact methods for training CNNs is also described in this chapter including the used learning policies to guarantee good training results. The chapter ends with a brief introduction how inference of a Keras model can be measured on the NVIDIA Jetson Xavier using TensorRT.

3.1 Dataset descriptions

Instead of throwing a model blindly at a dataset, one should always investigate how the data looks like and how training could be improved through pre-processing and data augmentation.

3.1.1 CIFAR

Both CIFAR10 and CIFAR100 [Kri12] datasets consist of RGB images with a spatial resolution of 32x32. The images of CIFAR10 are categorized as 10 distinct classes and CIFAR100 as 100 distinct classes. 60 000 labeled images are available, which are pre-split into 50 000 training and 10 000 test images with an equal distribution of classes. 10 000 images from the training images are used as validation. Example images with classes are shown in fig.2.1.

Preprocessing For both CIFAR datasets, images should be either normalized or standardized. Doing a quick online search, no method seems to be superior. Hence, channel-wise standardization is chosen. This means, that each of the RGB channels has a mean of 0 and a standard deviation of 1 after pre-processing. For this, the statistics of the dataset, used for transforming the training and validation sets, are calculated only on the training set to avoid bias towards the validation or test set.

Data augmentation The CIFAR datasets, especially CIFAR100, do not contain a lot of images per class when compared to, for example, ImageNet. Therefore, it makes sense to apply data augmentation to the training dataset to enrich the variability for each class. The standard techniques concerning data augmentation for image datasets can be applied to CIFAR. This includes rotation, horizontal and vertical shifts, and horizontal flips. These functions should be applied randomly to every image at each epoch.

In Keras, this functions are already built-in through the `ImageDataGenerator` class, which is used for creating custom `tf.data.Dataset` objects holding the training and validation data.

Both CIFAR10 and CIFAR100 ared transformed through a horizontal flip, which is applied with a 50% chance and a random 20% shift in both horizontal and vertical directions. Results are already satisfying with those augmentation functions, hence rotation is not applied to images.

3.1.2 Cityscapes

The Cityscapes [Cor+16] dataset holds RGB images with a size of 1024x2048 showing urban scenes. Ground truth labels for semantic segmentation consist of 19 classes including cyclists, pedestrians, cars, and vegetation. This dataset is often used for semantic segmentation datasets for comparing different methods and network architectures and is therefore also chosen for testing shunt connections for semantic segmentation tasks.

Coarse and fine annotations are available for a different number of images. While 20 000 images are available with coarse annotations, only 3475 images are annotated finely. Nevertheless, in this work, only the fine annotations are used since they are also used for evaluating MobileNetV3 on Cityscapes [How+19]. The images are split into 2975 training images and 500 validation images. The test set for Cityscapes is not available for the public yet, hence no test set will be used and the validation accuracy is reported. This choice has the disadvantage that a bias towards the validation set is introduced, but it holds the advantage of a bigger training set yielding better results. Examples of images including the fine annotations are shown in fig.2.3.

Pre-processing Images from Cityscapes are pre-processed by normalizing images to the range [-1,1] like it is done for the Cityscapes experiments on MobileNetV3 [How+19].

Data augmentation Regarding data augmentation, images and labels are cropped to a given size by randomly cropping while preserving the aspect ratio of the original image. Furthermore, before cropping there is a 50% change for an image to be flipped horizontally and scaled with a factor between 0.5 and 2. This ensures that a wide variety of object sizes are found in the dataset and the model does not hold a strong bias towards small or big objects.

These data augmentation steps are also used by the original authors of the DeeplabV3 [Che+17c] and the MobileNetV3 [How+19] model. The official TensorFlow repository implements these transformations in native TensorFlow, resulting in simple and fast code. Hence the code of the official repository[1] is cloned and used for this thesis, although small changes are made to how the labels are represented at the end. The original method returns the labels already hot-encoded, while for this work it is preferred to hot-encode the labels directly when calculating the cross-entropy loss. It is also worth highlighting, that the labels and images both get scaled and cropped when applying data augmentation.

3.2 Training, Validation and Test Splits

In the research area of machine learning and deep learning one key method is splitting the available data into training, validation, and test splits. This procedure assures that results are not biased towards the training data. It is shown in sec.2.2.6 that machine learning algorithms often lead to overfitting, where results on the training set are satisfying, while predictions on the validation set do not hold any value.

These three splits are necessary when trying to find the best hyperparameter configuration for a given algorithm and dataset, which is called hyperparameter optimization. Theoretically, one could use only two splits: training and validation, where hyperparameter configurations are optimized by looking at the score on the validation set. This approach would result in a bias regarding the validation training set. For example, when the validation dataset is not balanced regarding its classes, a bias is introduced to weigh images labeled as the most prominent class inside the validation set more heavily. If this bias is not correctly representing the real use case, the model will yield worse results on unseen data. That is the reason why the third dataset is introduced: the test set. This dataset is not used during the optimization process of hyperparameters. It truly represents unseen data and the evaluation of it corresponds to the real performance of the model.

In conclusion, this process shall be used whenever possible. Sometimes evaluation scores of the validation datasets are compared in literature when the validation set is clearly defined and the test set is not openly available, as is the case for ImageNet and Cityscapes.

3.3 Learning Rate Policies

When training a deep CNN model, it is desired to get the best possible result while training times are still feasible. The learning rate, which is a hyperparameter when applying a gradient descent solver, has to be chosen carefully since too high or too low learning rates will possibly not find the optimal solution. At the beginning of the training, higher

[1] https://github.com/tensorflow/models/tree/master/research/deeplab

learning rates should be applied to find the global minimum of the loss function approximately, which is further refined by a smaller learning rate. Different policies exist about how and when to reduce the learning rate during training, two of which are used in this thesis and explored in this section.

3.3.1 Plateau policy

Experience shows, that when training models with a high learning rate, the loss of the training will decrease early on but will plateau after some while. Reducing the learning rate at this point ensures that the loss will continue to decrease. The 'plateau policy' can be derived from this observation. Each time the training loss reaches a plateau, the learning rate gets decreased by a fixed factor like it is described in eq.3.1. Two hyperparameters are necessary for this method: *factor* which is used to decrease the current learning rate and *patience* which controls for how many epochs the training loss must not increase to be considered as plateaued. Fig.3.1 shows a typical example of the learning rate curve when using the plateau policy. It is visible, that once the training loss reached a plateau, it can be further reduced by decreasing the learning rate. The disadvantage of this policy is that the learning rate may get reduced too quickly if the loss does not decrease for a few epochs, such that the training gets stuck at a local minimum. Its advantage is, that it is not much sensitive regarding its hyperparameters, therefore eliminating the problem of finding the right learning rate. In practice, this learning rate is suitable for simpler problems, which do not depend strongly on the used learning rate and should be used whenever applicable.

$$\mathrm{lr}_{updated} = \mathrm{lr}_{current} \cdot \mathrm{factor} \tag{3.1}$$

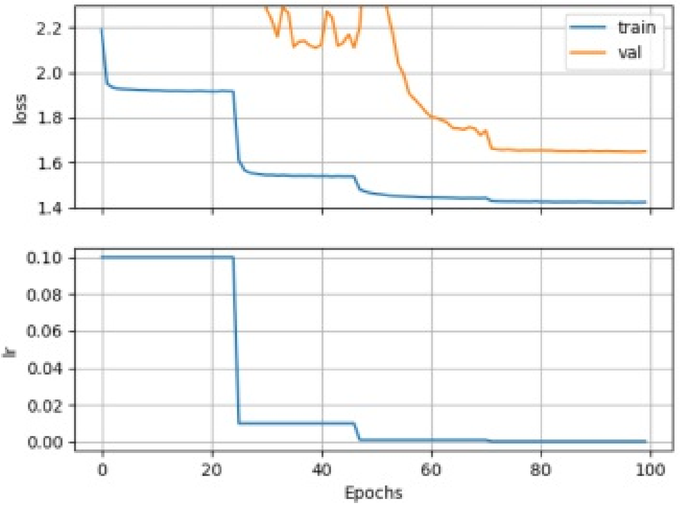


Figure 3.1: Example of a learning rate curve when using the plateau policy.

3.3.2 Poly learning rate

The 'poly' learning rate policy [LRB15] reduces the learning rate continuously over the whole training procedure. For iteration i of the training procedure, the learning rate *lr* gets updated according to the following equation:

$$\text{lr}_i = \text{lr}_{base} \cdot \left(1 - \frac{i}{\text{max\_iter}}\right)^{\text{power}}, \tag{3.2}$$

with *lr*

with lr_{base} is the base learning rate used for the first iteration, *max_iter* the number of maximum iterations of the training and *power* a hyperparameter controlling how the learning rate is reduced for each iteration. Multiple learning rate curves with different *power* values are shown in fig.3.2.

Learning rates in Keras can be easily updated only from one epoch to another one. So the following equation is implemented in the code, which uses epochs instead of iterations:

$$\text{lr}_e = \text{lr}_{base} \cdot \left(1 - \frac{e}{\text{max\_epochs}}\right)^{\text{power}}, \tag{3.3}$$

with e being the current epoch and *max_epochs* the number of maximum epochs.

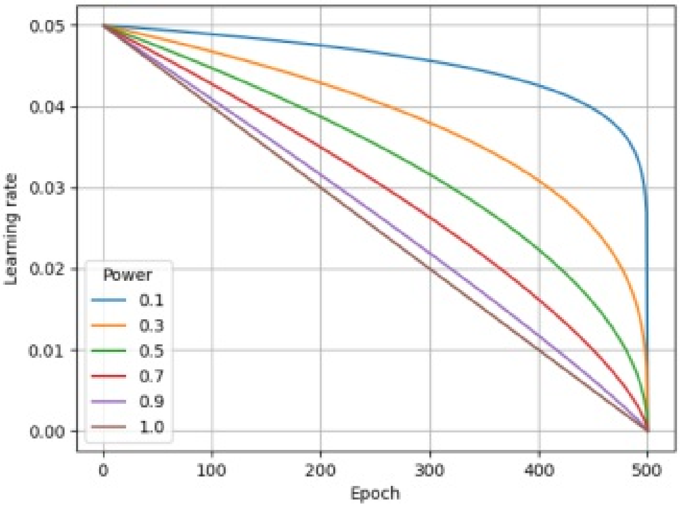


Figure 3.2: Poly learning rate policy for different *power* values.

For the DeeplabV3 architecture, a *power* value of 0.9 yields the best results, which corresponds to an almost linear decrease over the whole training process. Hence, for this work this value is also used whenever using the poly learning rate policy. It should be applied, whenever it is observed, that the plateau policy does not provide the best possible results. This is usually the case, when the problem depends strongly on the right choice of learning rate.

3.4 Hardware Setups used for Training CNNs

In this work two different hardware setups are used for training CNN models. For image classification on CIFAR, a simple GPU is capable of performing training with large batch sizes. Hence, a desktop PC is used for training those models. It consists of a GTX 970, which holds 4 GB of VRAM, an I5-4440 CPU running at 3.10 GHz, and 8 GB of RAM. Data is stored on a M2-SSD for fast access to training data.

For training models, which require much more memory, nodes of the Vienna Scientific Cluster (VSC)[2] are used. The newest iteration of the VSC consists of two supercomputers VSC-3 and VSC-4. VSC-3 is used for training deep learning models because it also holds special-purpose GPU nodes besides its 2800 other computation nodes. VSC-3 is considered highly energy efficient with its innovative oil cooling setup. Each node holds an Intel Xeon Processor E5-2650 v2 with 2.6 GHz holding 8 cores each. The most interesting properties for this thesis are the ones of the GPU nodes. VSC-3 provides single-GPU nodes holding a single GTX 1080 and multi-GPU nodes consisting of up to 8 GTX 1080 GPUs. The multi-GPU nodes seem suitable for training models with a high memory requirement since they provide up to 32 GB of VRAM, which is enough for the needs of this thesis.

3.5 Measuing Inference on NVIDIA Jetson Devices

The abstract goal of this thesis is to enable modern CNN architectures like MobileNets, such that they can be used for real-time applications on embedded devices. To measure the success of the used methods, it is, therefore, necessary to measure inference times on embedded devices.

There are many different embedded devices available with big ranges of power consumption and computational capabilities. A small battery-powered device introduces different constraints for designing CNN architectures compared to a relatively strong device from the NVIDIA Jetson series. This work focuses on the second case, where computational resources are available, which are comparable to the ones of a very small desktop computer. Nevertheless, the investigated methods might also apply to other device categories.

3.5.1 The NVIDIA Jetson series

Firstly introduced in 2014, the Jetson series started with the TK1, which held a quad-core ARM CPU and a GPU using NVIDIA's Kepler architecture. This first iteration of the Jetson series is considered deprecated since the product is no longer vented by NVIDIA. Its successor is the TX1, which uses a Maxwell GPU and more RAM compared to the TK1.

In 2017, the TX2 got released, which holds a GPU designed with the Pascal architecture and again more RAM compared to the older products. The so far strongest member of the Jetson series is the Jetson Xavier, which was released by NVIDIA in early 2019. Using an ARM CPU with 8 cores and a Volta-based GPU with 512 CUDA cores, NVIDIA promises certain applications may run 20 times faster compared to the TX2. It is also explicitly designed for deep learning applications through its built-in tensor processor units (TPUs), which are also referred to as deep learning accelerators (DLAs). The Xavier is chosen as the main device for measuring the inference of generated models since it is the most suited device for running big semantic segmentation models.

The developer version of the Jetson Xavier is also known as the Jetson AGX Xavier. Its main specifications are shown in tab.3.1.

[2] https://vsc.ac.at/home/

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8 MB L2 + 4 MB L3
Memory	32 GB 256-Bit LPDDR4x | 137 GB/s
Storage	32 GB eMMC 5.1
DL Acccelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Power	up to 30 W

Table 3.1: Main specs of the Jetson AGX Xavier [NVI].

3.5.2 Optimizing models through TensorRT

To utilize the full potential of the Jetson devices, and especially of the Xavier, NVIDIA released its own graph format for representing and storing deep learning models called TensorRT[3]. TensorRT uses custom inference engines for ensuring optimized inference times. There are many built-in utilities available like building such engines from other model formats or doing quantized inference.

Building TensorRT inference engines out of other model formats is done by optimizing the graph representation of the network. One possibility is to fuse nodes of the graph. For example, one combination of consecutive layers very often encountered in CNNs is a convolutional layer followed up by a batch normalization layer. A mathematical statement can be made, that shows that these two consecutive layers can be fused into one single layer, removing some unnecessary computations. TensorRT does also something called 'Kernel auto-tuning', where the algorithm for certain kernels is chosen dependent on the exact application and target platform. This means, when creating a TensorRT engine, the same model will be run many times using different combinations of CUDA kernels to find the best configuration for the given platform. This iterative process often needs several minutes to complete, even when using a rather small model.

There is an integration of TensorRT into TensorFlow available, in such a way that low latency inference can be done in Python using a TensorFlow model object. Experience showed that this integration is slower than building a TensorRT engine out of the saved TensorFlow model. Hence, building the engine out of the model produced by TensorFlow is preferred over building the engine directly.

Another main functionality of TensorRT is quantizing inference with minimal accuracy losses. A standard TensorFlow model used FP32 for its inference. TensorRT allows FP16 and INT8 quantization to utilize the GPU and TPU to their full potential.

Building an TensorRT engine from a Keras model Keras models are not natively supported in TensorRT, but one has to make the detour to convert a Keras model first into the ONNX[4] model format, which can be optimized and run by TensorRT. The ONNX model format tries to act as a bridge between different model types including formats from TensorFlow, Keras and PyTorch, and is therefore widely used by the community. An ONNX model uses a certain set of operations called the opset. Newer opsets are released regularly to support the most recent network architectures, which may hold operations, which are not supported by the newest version of TensorRT.

The official tf2onnx repository[5] by the ONNX-group provides a simple way to convert a .h5 model created from Keras

[3] https://developer.nvidia.com/tensorrt
[4] https://onnx.ai/
[5] https://github.com/onnx/tensorflow-onnx

to an ONNX model using a single line of Python code. The desired opset used for building the ONNX model can also be set by the user. The workflow for running a Keras model on the Xavier using TensorRT is therefore to firstly convert it to the ONNX format and benchmark that model using the TensorRT command line tool `trtexec`, which provides the functionalities of TensorRT, which are mentioned above.

Version 7.1.3 of TensorRT is used for all experiments, supporting ONNX opsets up to 11. Hence, Keras models are also converted to ONNX using the opset 11.

Chapter 4

Keras Implementation

This work uses Keras [Cho+15] and TensorFlow [Mar+15] as its deep learning framework to implement CNN models, pre-process datasets, and train models.

TensorFlow is one of the most successful deep learning frameworks developed by Google Brain. It was initially designed for running machine learning algorithms on heterogeneous systems, like HPC clusters or a CPU-GPU pair. Instead of executing computations eagerly, a computational graph is built up, which is optimized before calculations are performed. TensorFlow is written in C while providing a C++ and a python API. CUDA is used for executing computations on GPUs, making NVIDIA GPUs mandatory for high training speeds. In this work, the Python API of TensorFlow 2.x is used instead of 1.x to support useful functionalities like new layer types or the `tf.data` module.

Keras is a high-level deep learning API for Python built on top of TensorFlow. Formerly it supported several backends including Theano [Tea+16] or Microsoft Cognitive Toolkit (CNTK) [Sei17] besides TensorFlow. In version update 2.4, the support for other backends was dropped and at the time of this work, the most recent version of Keras is only available through TensorFlow itself via the `tf.keras` module. Keras is chosen as the main tool for implementing shunt connections since a model is defined as a `Model` object, which still holds information about its layers. This makes it possible to modify or exchange layers of a given model, in contrast to naive TensorFlow, where a model is stored as its graph representation, where single layers are not distinguishable and accessible anymore.

Although Keras promises simplicity, several problems occurred while implementing the necessary methods for inserting shunt connections. This chapter should provide insight, which problems were encountered using Keras and TensorFlow and how they got solved.

4.1 Models

CNN networks are implemented in Keras using its `Sequential` API. It gets its name from the fact, that networks are implemented by passing an input from one layer to the next one at each line sequentially. Under the hood, the computational graph needed for TensorFlow is built, which corresponds to the created model. A statement defines one layer, a block of layers, or again a `Sequential` model, which gets called by a single or multiple input tensors. The most common layer types are already built-in Keras resulting in a simple process of network generation.

Although Keras provides this simple API to create CNN architectures, it is not an easy task to re-implement exactly the network architectures from research papers since many important details like batch normalization parameters or special activation functions must be considered. The target of Keras is to provide a fast and simple generation of the most common architectures, which it does very well. But the process of implementing custom architectures with special needs suffers a bit.

4.1.1 Pre-built MobileNets

Luckily, a variety of models are already pre-built in Keras including MobileNets. The pre-built models can be created through a single line of code and are customized by providing parameters like input size or the number of classes. These models are defined in the `keras.applications` module and pre-trained weights on ImageNet are also available for most architectures. This work uses the MobileNetV2 and MobileNetV3 models from this module. It is worth mentioning, that the resolution, width, and depth multiplier parameters can be provided as a parameter for creating different variants of those models.

The online community does a great job at implementing new CNN architectures in Keras and providing them through GitHub or other tools. Hence, many more modern architectures are as simple to use as they would be implemented as part of `keras.applications`. Unfortunately, the semantic segmentation version of MobileNetV3, which this work uses as the main segmentation architecture, is not openly available in Keras on the internet. Therefore, this model had to be built from scratch.

4.1.2 Segmentation Head for MobileNetV3

The semantic segmentation architecture using MobileNetV3 as its feature extractor as described in sec.2.7.1 was originally implemented using TF-Slim[1], another high-level API for TensorFlow. This implementation is openly available on the TensorFlow GitHub page[2]. Unfortunately, a naive TensorFlow or TF-Slim model can not be converted into a Keras model, since those models are stored abstractly as a computational graph without holding information about the exact layer configuration. Because the whole shunt connection implementation is based on Keras, the segmentation version of MobileNetV3 has to be implemented in Keras.

When implementing a certain network architecture, it is important to validate it in some way. In our case, there are pre-trained weights available for the Cityscapes model as part of the TF-Slim implementation, which can be converted to be usable in Keras and used for validating the custom implementation.

Two sources are available, defining the MobileNetV3-segmentation architecture: the original MobileNetV3 paper [How+19] and the TF-Slim implementation. Extracting information from the TF-Slim repository is a tedious process since the model is spread over several files holding multiple hundred lines of code. To use the pre-trained weights for validation, every detail of the TF-Slim implementation has to be copied over to the Keras code.

The default MobileNetV3 architecture is used as the feature extractor with small modifications like reducing the channel numbers in the last stage and extracting low-level features from the second stage of the network. This part of the network can be built out of the building block defined in `keras_applications` used for building the MobileNetV3 architecture.

[1] https://github.com/TensorFlow/models/tree/master/research/slim
[2] https://github.com/TensorFlow/models/tree/master/research/deeplab

The segmentation head was built by taking inspiration from another repository[3], which implemented the semantic segmentation version of MobileNetV2 in Keras.

The output of the pre-built TF-Slim version of the model could not be reproduced with the custom Keras model. Hence, the custom model was debugged by comparing the outputs of each layer of the TF-Slim model and the custom Keras model one by one. Two errors in the initial custom implementation shall be highlighted.

Numerical constants Checking the output of each layer one by one, the output of the model already diverged greatly after a few convolutional layers. This was due to a different numerical constant used in the h-sigmoid activation function (see eq.2.16). Its divisor is defined as 1.6667 in the TF-Slim model while being truly $1/6$ in the MobileNetV3 implementation of `keras_applications`. Due to the non-linear behaviour of the model, this small error caused the custom model to perform much worse.

Upsampling layers The segmentation head uses two upsample layers to build the final segmentation mask. These upsampling layers are implemented as bilinear resize layers, which are already built-in in naive TensorFlow as `tf.image.resize`. The behaviour of this function got changed from TensorFlow 1 to TensorFlow 2. Since the TF-Slim implementation of the segmentation head uses TensorFlow 1, it uses the old variant of the resize function, which is considered faulty by the online community because it does not match the expected behaviour under some circumstances. Using TensorFlow 2 and the new resize function results in a drop in accuracy using the pre-trained weights from TF-Slim. Therefore, the old resize function, which is fortunately available as `tf.compat.v1.resize.image` in TensorFlow 2, should be preferred when using the pre-trained weights from TF-Slim. This work will follow this approach, although the new one should be considered when training the model from scratch because it is seen as superior in comparison with the old version.

In the end, the same output as the TF-Slim model has been achieved with the custom Keras model up to the third decimal place.

4.2 Loss and Metrics

Besides specifying the optimizer for model training during the `compile()` call of a Keras model, the task-dependent loss and metrics must also be provided.

Keras has the major loss and metrics already built-in. This includes the cross-entropy loss for classification and semantic segmentation, top-1 accuracy for classification tasks, and also the mIoU metric needed for semantic segmentation.

Unfortunately, the Keras implementations of the cross-entropy loss and the mIoU metric do not support void labels, as they occur in the Cityscapes dataset. In this special case, custom functions have to be used. These custom functions are implemented through a weighted approach, where pixels, which are classified as 'void' by the ground truth labels, do not contribute to the loss or the metric calculation. It is important that pixels labeled 'void' are completely ignored and not marked as true or false in mIoU calculations because otherwise false results are obtained, which are not comparable to literature.

[3] https://github.com/bonlime/keras-deeplab-v3-plus

4.3 Training Setups

The training of CNNs can be done through a single line of code in Keras by calling the `fit()` method of a Keras model. It executes training loops including weight updates, validation, custom callbacks, and other functionalities. The most convenient way to provide data for training is using `tf.data` objects, which do pre-processing, data augmentation, and loading in a parallel fashion. They are also recommended to be used when doing parallel training on multiple GPUs.

4.3.1 The shunt-trainings model

Shunt connections have to solve the task of feature matching, where the feature maps of a different model have to be replicated. Creating the data for training and validation is done by feature extraction, where feature maps are extracted during the forward pass of the target network. It is described in sec.5.1.2, that a dynamic approach where feature maps are extracted on the fly during the training process is preferred. This setup can be achieved by creating a single Keras model consisting of the target model and the shunt-inserted model. This is visualized in fig.4.1. It is making use of the fact, that the beginning of the model up to the shunt location is the same for the target and the shunt-inserted model, resulting in fewer calculations. The output of this first part of the network is fed into the shunt connection model, while also traversing further the target model until it reaches the first block, which is not replaced by the shunt connection. The target model from this point on has not to be computed. The outputs of the shunt and the blocks to be replaced are compared using the mean-squared error forming the loss function, which should be minimized. For the input image, the original `tf.data` object can be used, performing data augmentation resulting in a rich variety of training samples.

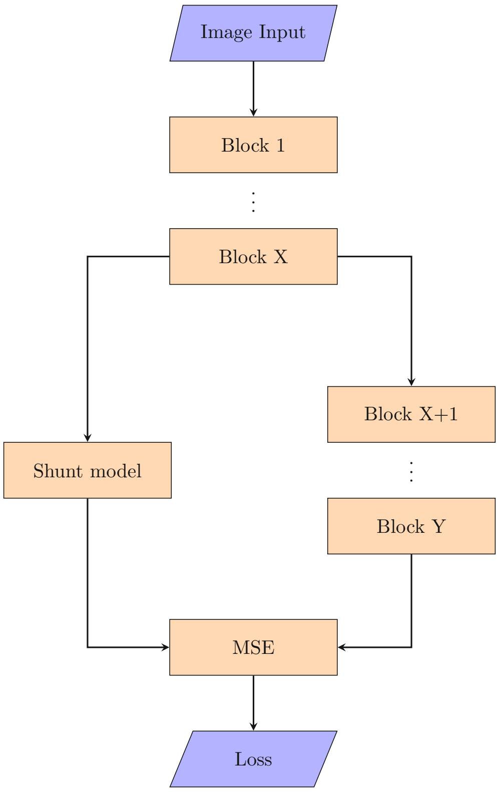


Figure 4.1: Flow diagram for the shunt-trainings model. Block X of the target model marks the shunt location and Blocks X+1 to Y are the blocks replaced by the shunt connection. MSE denotes the mean-squared-error.

4.3.2 Training model for Knowledge Distillation

Similar to the shunt-training model, in the case of knowledge distillation, two models also have to be computed at the same time and their output compared. Fig.2.15 and 2.16 show the general setup for the two knowledge distillation used in this work, but they also visualize the basic setup for a Keras model implementing the algorithms. The teacher and the student model get computed in a parallel fashion using the same image input. This setup is highly memory consuming, since two models have to be stored on the GPU, meaning that batch sizes have to be reduced in some cases, for example when training a model for a semantic segmentation task with high image resolutions.

4.4 Modifying an Existing Keras Model

The underlying backend of a Keras model is still a computational graph even when it is defined through the `Sequential` API. This means, that layers of a Keras model cannot be simple modified or deleted since this would result potentially in an invalid or disconnected graph. Hence, when modifying a single layer in a Keras model, it can be well possible, that the whole model has to be rebuild, potentially using the `Sequential` API again. The advantage of Keras is, that the whole model architecture is also stored as layers, which can be iterated through. This means, in contrast to TF-Slim, that the model can be rebuilt even without having the original code of the initial model. This advantage was the main reason, why Keras was chosen for this work since the methods for inserting shunt connections heavily rely on this property.

This section should motivate and introduce the `modify_model` function, which has been implemented for this work. It is capable to modify an existing Keras model including producing the reduced models needed for knowledge quotients and inserting a shunt into a model while replacing blocks at the same time.

4.4.1 Rebuilding a simple sequential model

A Keras model with a single branch can simply be rebuilt by creating an input tensor and calling all layers of the model sequentially by the output tensor of the previous layer. The exact syntax is shown in prog.4.1.

It is worth mentioning, that the weights of the rebuilt model are initialized using the initializer of the corresponding layer and do not adopt the weights of the original model automatically. When the rebuilt model should hold the same weights as the original one, the weights have to be copied manually.

```
def rebuild_keras_model(model):
    x = model.input
    for layer in model.layers():
        x = layer(x)
    model_rebuilt = keras.models.Model(model.input, x)
    return model_rebuilt
```

Program 4.1: Simple method for rebuilding a Keras model with a single branch.

4.4.2 Rebuilding models with multiple branches

In prog.4.1, each layer gets called by the output tensor of the previous layer. This syntax only works when the model does not split into multiple branches. Modern CNN architectures often split into multiple branches, which are merged shortly after, as is the case for residual blocks or squeeze-and-excitation modules. For such complicated networks with multiple branches, the rebuilt is not trivial, although there are many possible ways to solve this problem. The self-written `modify_model` method, which is capable of modifying an existing Keras model is considered one of the main parts of the code used for inserting shunt connections.

A merging layer has to be called similarly to this:

```
x = merging_layer([x1,x2]),
```

where `x1` and `x2` are the incoming tensors produced by previous layers. It is not very practical to find the input tensors of a merging layer when it is encountered while looping over the layers of a model. If one knows beforehand, which tensors act as input tensors for each merging layer, they can be stored immediately when the corresponding layer is encountered in the loop and used later on.

Calculating the knowledge quotients of a model, as it is defined in sec.2.9.2, requires to built custom models out of the original model, where one residual block is deleted. Building these reduced models requires knowing where the residual blocks are inside the network and which tensors they connect.

Prog.4.2 calculates the indices of merging layers and the indices of merged layers. It is assumed that two layers are merged by each merging layer, where one of the merged layers is the predecessor of the merging layer. Therefore, only one of the two input tensors has to be stored. The information about merging layers and their input tensors is stored in a dictionary, where keys are the indexes of merging layers and values are their corresponding input tensor, produced not by the predecessor. Breaking down the code, its core lies in extracting information about the incoming nodes of each merging layer. The information about the incoming nodes of a layer is stored in the private field `_inbound_nodes[-1].inbound_layers`. The `get_first_layer_by_index` method returns the lowest index of a layer included in a list of layers. This index is stored in the corresponding dictionary as the value of its merging layer.

```
def identify_residual_layer_indexes(model):
    layers = model.layers
    add_incoming_index_dic = {}
    mult_incoming_index_dic = {}
    for i in range(len(layers)):
        layer = layers[i]
        if isinstance(layer, Add):
            input_layers = layer._inbound_nodes[-1].inbound_layers
            incoming_index = get_first_layer_by_index(model, input_layers)
            add_incoming_index_dic[i] = incoming_index
        if isinstance(layer, Multiply):
            input_layers = layer._inbound_nodes[-1].inbound_layers
            incoming_index = get_first_layer_by_index(model, input_layers)
            mult_incoming_index_dic[i] = incoming_index
    return add_incoming_index_dic, mult_incoming_index_dic
```

Program 4.2: Simple method for recreating a Keras model with a single branch

The starting point is the algorithm of prog.4.1. A model holding Add layers can be rebuilt by providing the dictionary and modifying the code accordingly. For this work, it is sufficient that only simpler architectures like MobileNetV2 are investigated, where there are not more than two branches inside the network at a time. Prog.4.3 shows how the add_dictionary is utilized. When a layer is encountered, which acts as an input tensor of an Add layer (line 8), its output tensor is stored inside the dictionary instead of its index. This tensor is accessed when the corresponding Add layer is encountered (lines 4&5).

```
1  def rebuild_keras_model(model, add_dictionary):
2      x = model.input
3      for i, layer in enumerate(model.layers()):
4      if layer.type is tf.keras.layers.Add:
5          x = layer([x, add_dictionary[i]])
6      else:
7          x = layer(x)
8      if i in add_dictionary.values():
9          add_index = get_key(add_dictionary, i)
10         add_dictionary[add_index] = x
11     model_rebuilt = keras.models.Model(model.input, x)
12     return model_rebuilt
```

Program 4.3: Rebuilding a MobileNetV2 mode

In order to calculate the knowledge quotient of a residual block, the residual block has to be deleted from the network and the accuracy of that reduced network has to be measured. One may come up with a similar algorithm to the one in prog.4.4. But this approach already fails, when two residual blocks are consecutive and the first one gets deleted since the information about the input tensor of the add_dictionary is wrong. The input tensor of the second Add layer needs to be changed to the output layer of the previous block.

```
def rebuild_keras_model(model, add_dictionary, delete_block_index):
    # calculate which indices to delete
    add_index = add_dictionary.items()[delete_block_index][0]
    start_block_index = add_dictionary[add_index]

    x = model.input
    for i, layer in enumerate(model.layers()):
        if i in range(start_block_index+1, add_index):
            continue
        if layer.type is tf.keras.layers.Add:
            x = layer([x, add_dictionary[i]])
        else:
            x = layer(x)
        if i in add_dictionary.values():
            add_index = get_key(add_dictionary, i)
            add_dictionary[add_index] = x
    model_rebuilt = keras.models.Model(model.input, x)
    return model_rebuilt
```

Program 4.4: Deleting a block of MobileNetV2

A general solution to the rewiring problem shall be found. We start again with the simpler problem of just rebuilding a model without deleting blocks or modifying layers. Prog.4.5 achieves this by storing the output tensor of every layer in an ordered dictionary. If the algorithm encounters a layer with more than one inbound node, as is the case for all merging layers, a list of input tensors is built, which is used for calling the merging layer. It is necessary to store every output tensor since it is not known beforehand if this tensor has to be available at a later point. Having this code available, the next step will be adding possible modifications when rebuilding a model.

```python
def rebuild_model(model):
    layer_outputs_dict = OrderedDict()

    input_net = Input(model.input_shape[1:], name=model.layers[0].name)
    x = input_net
    layer_outputs_dict[model.layers[0].name] = x
    for i in range(1,len(model.layers)):
        next_layer = model.layers[i]
        input_layers = layer._inbound_nodes[-1].inbound_layers
        if not isinstance(input_layers, list):
            x = next_layer(layer_outputs_dict[input_layers.name])
        else:
            input_list = []
            for layer in input_layers:
                input_list.append(layer_outputs_dict[layer.name])
            x = next_layer(input_list)

        layer_outputs_dict[next_layer.name] = x

    model_rebuilt = keras.models.Model(inputs=input_net, outputs=x)
    return model_rebuilt
```

Program 4.5: General algorithm for rebuilding a model with multiple branches

4.4.3 Supported model modifications

For this work, several model modifications had to be implemented to generate custom models.

Delete layers When calculating the knowledge quotients of a model, it is necessary to delete a single block of the model one at a time. This is realized by adding a parameter that holds a list of ascending layer indices that should be deleted. When iterating over the layers, a layer will be omitted if its index is in this list (see prog.4.6). Now the network needs to be rewired if a merged layer got deleted. It is assumed, that only one merging layer has to be rewired during the whole rebuilding process, which is the case when calculating knowledge quotients. Therefore it makes sense, to rewire to the latest layer of the network, which was not deleted. This is the layer with index `layer_indexes_to_delete[0]-1`. This rewiring process is visualized in fig.4.2 and implemented in prog.4.6. The constraint that only one rewiring is possible is not a strong one, since the method can be simply called multiple times, if multiple residual blocks have to be deleted, where rewiring may occur.

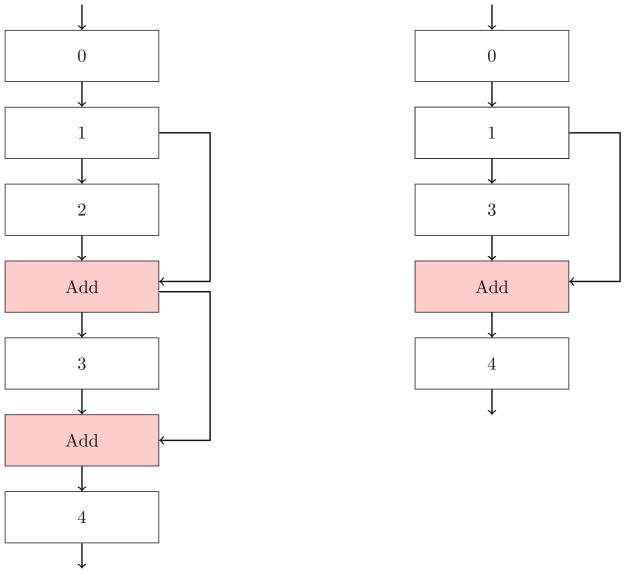


Figure 4.2: Visualization of the rewiring process. Layer 2 and the following Add layer are deleted, which results in rewiring of the second Add layer.

```
def modify_model(model, layer_indexes_to_delete):
    layer_outputs_dict = OrderedDict()

    input_net = Input(model.input_shape[1:], name=model.layers[0].name)
    x = input_net
    layer_outputs_dict[model.layers[0].name] = x
    for i in range(1,len(model.layers)):
        if i in layer_indexes_to_delete:
            continue
        next_layer = model.layers[i]
        input_layers = layer._inbound_nodes[-1].inbound_layers
        if not isinstance(input_layers, list):
            if get_index_of_layer(model, input_layers) in layer_indexes_to_delete:
                x = next_layer(x)
            else:
                x = next_layer(layer_outputs_dict[input_layers.name])
        else:
            input_list = []
            for layer in input_layers:
                if get_index_of_layer(model, layer) in layer_indexes_to_delete:
                    rewired_index = max(layer_indexes_to_delete[0]-1,0)
                    tensor_output_list = list(layer_outputs_dict.items())
                    input_list.append(tensor_output_list[rewired_index][1])
                else:
                    input_list.append(layer_outputs_dict[layer.name])
            x = next_layer(input_list)

        layer_outputs_dict[next_layer.name] = x

    model_rebuilt = keras.models.Model(inputs=input_net, outputs=x)
    return model_rebuilt
```

Program 4.6: General algorithm for deleting layers of a model with multiple branches

It has to be mentioned, that this algorithm does not work on arbitrary setups. For example, in the case of an Add layer, both merged layers must have the same dimensions. Therefore, the rewiring can fail, if the new chosen layer has different dimensions. Hence, the user is responsible to provide a valid list of layers to delete. If an error occurs when building the model, an appropriate error message is print to the console.

Insert shunt connection Another important feature needed for this work is placing a shunt inside a model. This can also be done as part of the `modify_model` method so that blocks can be replaced by a shunt connection in one single step. Two parameters are used for defining the placement of the shunt location: the model itself (`shunt_to_insert`) and where it should be placed (`shunt_location`). Since the indexing of the model changes when inserting the shunt connection, the rewiring has to cover the edge case, when deleting layers and inserting the shunt at the same time. It is solved in a way, that the shunt is chosen as the layer to be merged when a merging layer has to be rewired.

```python
def modify_model(model, layer_indexes_to_delete):
    layer_outputs_dict = OrderedDict()

    input_net = Input(model.input_shape[1:], name=model.layers[0].name)
    x = input_net
    layer_outputs_dict[model.layers[0].name] = x
    for i in range(1,len(model.layers)):
        if i in layer_indexes_to_delete:
            # insert shunt
            if i == shunt_location:
                x = shunt_to_insert(x)
                layer_outputs_dict['shunt'] = x

                if i in layer_indexes_to_output:
                    outputs.append(x)
            continue

        # insert shunt
        if i == shunt_location:
            x = shunt_to_insert(x)
            layer_outputs_dict['shunt'] = x

        next_layer = model.layers[i]
        input_layers = layer._inbound_nodes[-1].inbound_layers
        if not isinstance(input_layers, list):
            if get_index_of_layer(model, input_layers) in layer_indexes_to_delete:
                if shunt_to_insert:
                    x = next_layer(layer_outputs_dict['shunt'])
                else:
                    x = next_layer(x)
            else:
                x = next_layer(layer_outputs_dict[layer_name_prefix + input_layers.name])
        else:
            input_list = []
            for layer in input_layers:
                if get_index_of_layer(model, layer) in layer_indexes_to_delete:
                    if shunt_to_insert:
                        input_list.append(layer_outputs_dict['shunt'])
                    else:
                        rewired_index = max(layer_indexes_to_delete[0]-1,0)
                        tensor_output_list = list(layer_outputs_dict.items())
                        input_list.append(tensor_output_list[rewired_index][1])
                else:
                    input_list.append(layer_outputs_dict[layer.name])
            x = next_layer(input_list)

        layer_outputs_dict[next_layer.name] = x

    model_rebuilt = keras.models.Model(inputs=input_net, outputs=x)
    return model_rebuilt
```

Program 4.7: General algorithm for replacing layers with a shunt connection

Transduce weights When rebuilding a model, in many scenarios it is necessary to adopt the weights from the original model, for example when inserting the shunt connection into the original model. Each layer in Keras implements the `get_weights()` and `set_weights()` method, which can be used for this purpose. The code in prog.4.8 is appended at the end of the previous programs to transduce the weights if the corresponding boolean parameter `transduce_weights` is set to true.

```
    if not transduce_weights:
        return model_reduced

    for j in range(1,len(model_reduced.layers)):

        layer = model_reduced.layers[j]
        if isinstance(layer, ReLU) or isinstance(layer, Lambda) or isinstance(layer, Activation):
            continue

        if shunt_to_insert:
            if layer.name == 'shunt':
                layer.set_weights(shunt_to_insert.get_weights())
                continue

        if len(layer.get_weights()) > 0:
            weights = model.get_layer(name=layer.name[len(layer_name_prefix):]).get_weights()
            model_reduced.layers[j].set_weights(weights)
```

Program 4.8: Transducing weights from the original model to the rebuilt model

Change properties of layers The last modification, which can be applied to a model, is changing properties of certain layers. So far, the layers of the rebuilt model have the same name as the original model, which may be desirable in a usual setting, but causes problems, when building a model for knowledge distillation since each layer in a Keras model must have a unique name. Therefore, another parameter `layer_name_prefix` is added to the program. If it is provided, it is added as a prefix to the name of every layer.

Changing such a property of a layer is not trivial since it is usually set once during the initialization of the layer and not changed afterwards. Hence, a detour has to be made using the `config` object of the layer. This object is a dictionary holding all information of a layer, which is needed to copy it. Under `tf.keras.layers`, the method `deserialize` returns a new layer object out of a given `config` object. Therefore, one can extract the `config` of a layer, modifying its entries, and call the `deserialize` method to create a layer object with changed properties.

Using this method, the name of a layer can be changed by setting the `name` entry of the layer's `config`. This is shown in prog.4.9. This program also shows how to add regularization to a layer, change its stride layers, and remove the activation function from a layer. These functionalities are also controlled through user parameters. When setting the `add_regularization` parameter to true, L2 regularization with a factor of 4e-5 (see sec.2.2.6) is added to every possible layer. By setting the `change_stride_layers` parameter to s, the first s layers of the model with stride 2 are changed to have a stride of 1. Since `ZeroPadding2D` layers are used before stride 2 layers in the Keras implementations of MobileNets to ensure the correct padding before downsampling, they are also removed when changing the stride of the following layer.

```
from tensorflow.keras.layers import deserialize as layer_from_config

layer = model.layers[i]
config = layer.get_config()
config['name'] = layer_name_prefix + config['name']

if add_regularization:
    if 'kernel_regularizer' in config.keys():
        config['kernel_regularizer'] = {'class_name': 'L1L2', 'config': {'l1': 0.0, 'l2': 4e-5}}
    if 'bias_regularizer' in config.keys():
        config['bias_regularizer'] = {'class_name': 'L1L2', 'config': {'l1': 0.0, 'l2': 4e-5}}

if change_stride_layers > 0:
    if isinstance(layer, ZeroPadding2D):
        config['padding'] = (1,1)
    if 'strides' in config.keys():
        if config['strides'] == (2,2):
            config['strides'] = (1,1)
            change_stride_layers -= 1

if dense_softmax_to_seperate_softmax:
    if i == len(model.layers)-1:            # last layer
        config['activation'] = None

next_layer = layer_from_config({'class_name': layer.__class__.__name__, 'config': config})
```

Program 4.9: Changing the properties of a layer using its configuration object

4.5 Multi-GPU training in Keras

Models with a high memory footprint, as is for example the case for semantic segmentation models, may require a multi-GPU setup to be trained efficiently. High batch sizes are crucial when training certain model types, hence requiring a large amount of virtual RAM on the GPU. Therefore, multi-GPU setups are used for training such models.

TensorFlow supports synchronous and asynchronous multi-device training procedures. According to the documentation, the synchronous methods synchronize the gradients of all devices during training after each forward pass. This can not be considered as a fully synchronous method for training a model holding batch normalization layers, since the forward pass itself, which is dependent on the whole batch of inputs, is not synchronized. As a result, a single device yields different results compared to multiple devices using a synchronous distribution strategy from TensorFlow. This fact has to be considered, when comparing the results of training procedures performed on different hardware.

Chapter 5

Experiments and Results

This chapter presents the experiments and results of shunt-connections on MobileNets. First, the results of the original paper are tried to be replicated. For this, the fine-tuning step gets investigated and further refined. Afterwards, the obtained insights will be applied to the MobileNetV3 classification and semantic segmentation case.

5.1 Reproducing the State-of-the-Art of Shunt Connections

Reproducing the results of a machine learning paper is often more challenging as it should be. In the ideal case, a well documented code repository using your favourite machine learning framework is openly available. In the worst case, the authors only state that they 'trained' their model, providing the reader with no information how to reproduce the results.

Unfortunately, the original paper of shunt connections [STA19] falls more into the second case. No hyperparameters are stated for the training procedures, no information is provided regarding data pre-processing and augmentation and the fine-tuning of shunt-inserted models is not described at all. There is also no code provided, so it is necessary to implement the methods on your own.

The goal of this section is to reproduce the results for both CIFAR datasets including the accuracy of the original, uncompressed model and the accuracy results after compression. It should summarize the information given in the original paper and which assumptions are made to reproduce the results.

5.1.1 Information Provided in the Paper

Once again, the general workflow for shunt insertions act as the starting point:

1. Train original model
2. Calculate knowledge quotients
3. Choose shunt location
4. Extract feature maps of original model at shunt location

5. Chose shunt architecture
6. Train shunt architecture on feature maps
7. Fine-tune shunt-inserted model

For steps 1, 3, 4, 6 and 7 detailed information is necessary to exactly reproduce the provided results, while steps 2 and 4 needed additional efforts to be implemented programmatically as described in the previous chapter.

Step 1: Train original model The paper states that MobileNetV2 is used as the original model, which should be compressed. Using the default MobileNetV2 architecture on CIFAR images is a little bit unconventional, because the default architecture has an output stride of 32, meaning, that at the end of the feature extractor, the input image will be downsamples by a factor of 32 in both spatial directions. CIFAR images have a resolution of 32x32. In the end, feeding them to MobileNetV2, the image gets downsampled to a single pixel, which is counterintuitive for a CNN. Running an online search, it is suggested to replace the first couple of 'stride=2' layers with 'stride=1' layers to reduce the total downsampling rate.

The paper does not state any changes to the original architecture, but due to information missing in other aspects, this fact my be not trustworthy.

Training of the original model is not covered in the paper, but the learning curves are provided. It becomes evident, that the model is trained for around 300 epochs in the case of CIFAR10 and 400 epochs in the case of CIFAR100.

Data augmentation, which would be usually applied to such task, is also not mentioned in the paper.

Step 3: Choose shunt location The exact shunt location is also not given in the original paper. From the given shunt architectures, the channel numbers of input and output can be extracted (see tab2.7). Comparing these numbers with the layers in tab.2.3, it can be concluded that blocks 7-14 are in the range of possibility to be replaced. Another source of information is the rate of compression of the original model for a given architecture. Due to the fact the original model architecture is not known either, this information does not help in deriving the used shunt location.

Step 4: Extract feature maps of original model at shunt location To exactly reproduce the results, it is also necessary to know, how the feature maps were extracted, which are used to initially train the shunt connections. The main question is, whether feature maps are saved once from the raw training set, without data augmentation, or if new feature maps are created at each training epoch.

Step 6: Train shunt architecture on feature maps Regarding the initial training of shunt connections on feature maps, it is stated that Adam [KB17] is used as the optimizer. Otherwise, no information is provided about the loss function or hyperparameters of the training.

Step 7: Fine-tune shunt-inserted model After inserting the trained shunt into the original model, a final training step is needed, in order to achieve the accuracy of the original model. In the paper this step is referred to as fine-tuning, indicating that low learning rates are used and that layers are potentially frozen during training. Sadly, the reader is completely left in the dark and has to guess, what the authors did exactly.

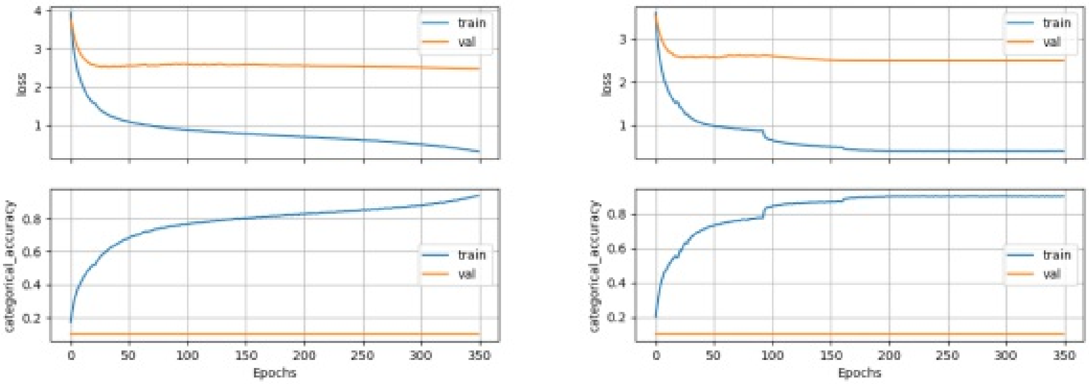
5.1.2 Finding Reasonable Assumptions

Step 1: Train original model Since there are no modifications stated regarding the used MobileNetV2 model it is worth trying the default architecture, although better variations may be available. A default hyperparameter configuration is chosen to create a baseline for each model and dataset variant. Doing hyperparameter optimization for each case needs simply too much time to compute, therefore a reasonable default configuration is chosen. Tab.5.1 shows these hyperparameters including parameters used for creating the model and all parameters used for training. L2-regularization is applied to each layer of the network to reduce the risk of overfitting, whose hyperparameters are chosen according to literature in [San+19; How+19].

Hyperparameter	Value
regularization factor	4e-5
dropout rate	0
learning rate policy	poly
power	0.9
$\mathrm{lr}_{\mathrm{base}}$	0.1
max_epochs	350
optimizer	SGD
momentum	0.9
batch size	64

Table 5.1: Default hyperparameter configuration used for training MobileNets on CIFAR datasets.

Training the default MobileNetV2 architecture on CIFAR10 using the default hyperparameter configuration and preprocessing and data augmentation according to sec.3.1.1, yields the learning curve shown in fig.5.1a. It can be seen that trainings loss decreases nicely, but validation loss plateaus after 30 epochs resulting in a tremendous overfit. It looks like the model is not able to generalize, but rather remembers training samples explicitly. Using a different learning policy does not change this behaviour, as it can be seen in fig.5.1b, which was generating using the plateau policy. It is concluded, that the default MobileNetV2 architecture is not suitable to be used for such low resolution images.



(a) Poly

(b) Plateau

Figure 5.1: Learning curves of the training of the default MobileNetV2 on CIFAR10 using the default hyperparameter configuration from tab.5.1. Once with poly learning rate policy and once with the plateau policy using a patience of 4 and a factor of 0.1.

In the previous section it has already been introduced the idea to remove some 'stride 2' layers at the beginning of the network, such that the image does not get downsampled to a single pixel. Replacing the first two 'stride 2' layers from the architecture, should produce better results according to online forums. Applying the default hyperparameter training configuration to this model, yields 93.32% accuracy on the test data set of CIFAR10, beating the model presented in the original shunt paper, which has 91.28% accuracy. Usually, the obvious choice would be the model with changed stride values to do experiments on. But the original paper most probably did not use this exact model. Nevertheless, some assumptions have to be made at this point, hence this model is chosen as the original model to be compressed.

Training this model on CIFAR100, using the same hyperparameters, yields 73.70% top-1 accuracy, beating the score of 68.3% reported by the original paper.

Step 3: Choose shunt location The location of the shunt location should be chosen manually with the help of the knowledge quotients of residual blocks. Deleting each residual block successively and measuring the accuracy of the created model, quantifies the potential of compression for the deleted block. Doing this for the modified MobileNetV2 models trained in step 1, yields the quotients shown in tab.5.2. It was shown in sec.2.9.1, that a knowledge quotient near 0 corresponds to a high potential of compression. As a consequence for this model, not a single residual block is of utter importance and the shunt location can be chosen freely. Note that for non-residual blocks, the knowledge quotient is not defined, therefore one must be careful with replacing those blocks, since they could hold much knowledge.

Block ID	KQ-CIFAR10	KQ-CIFAR100
1	-	-
2	-	-
3	0.045	0.069
4	-	-
5	0.052	0.051
6	0.031	0.024
7	-	-
8	0.016	0.021
9	0.010	0.022
10	0.008	0.031
11	-	0.033
12	0.020	0.033
13	0.023	0.031
14	-	-
15	0.009	0.064
16	0.010	0.070

Table 5.2: Knowledge quotients for the modified MobileNetV2 modesl trained on CIFAR10 and CIFAR100.

Looking at the original paper, one can enclose the choice to blocks 7-14. The goal is still to reproduce the results of the original paper, hence the shunt should be chosen in a way to achieve similar compression rates, although using maybe a different uncompressed model. Replacing the five blocks 8-12 yields a compression of around 30% when using one of the proposed shunt architectures. Therefore, this location will be used for all other steps.

Step 4: Extract feature maps of original model at shunt location There are two possibilities considered for this step: extracting feature maps once before training from the training and validation dataset and storing them in RAM or on disk, or extract them during the training for each epoch separately. Regarding the second method, practically speaking, one forward pass of the original uncompressed model has to be performed for each training batch, to obtain the feature maps. As a result, extracting the feature maps once, is far superior in speed. The disadvantage is, that data augmentation cannot be applied or at least only in a minimum setting.

In the case of CIFAR datasets, the whole dataset and extracted feature maps fit the local memory, making the option really attractive. This is not the case for other datasets like ImageNet or Cityscapes. For those datasets, the extracted feature maps have to be stored on disk and loaded for each epoch. It was observed, that loading the extracted feature maps from disk is even slower compared to generating them on the fly. In conclusion, the option of generating feature maps at each epoch is chosen due to this reason. The effect of data augmentation on the training of shunt connections is also not known. Generating feature maps dynamically allows to use data augmentation, which is therefore a big plus too.

Step 6: Train shunt architecture on feature maps Besides the used optimizer, no other information is provided for this step. This task is really similar to the task of feature matching, which can be implemented with the mean squared error as a loss function. In the case of shunt connections, it makes sense to compare target feature maps and produced feature maps by the shunt connection using this loss function.

Again, the hyperparameters for the training have to be selected. Luckily, this training is much less sensitive to hyperparameters compared to training the uncompressed model. Therefore a default hyperparameter configuration can be applied to various shunt connection setups, while one can still expect to achieve good results. This configuration is shown in tab.5.3.

Hyperparameter	Value
learning rate policy	plateau
patience	4
lr_{base}	0.1
$\text{lr}_{\text{factor}}$	0.1
max_epochs	100
optimizer	Adam
decay	0
batch size	64

Table 5.3: Default hyperparameter configuration used for training shunt connections on extracted feature maps.

Using the plateau learning policy options holds the advantage, that it controls learning rate automatically without depending on other hyperparameters. The algorithm is allowed to chose its best learning rate itself. Therefore, learning rates do not have to be changed manually for different setups.

Step 7: Fine-tune shunt-inserted model This step is probably the most complicated one and also has the least information provided. The paper writes that the full shunt-inserted model gets trained on the original dataset and calls this process fine-tuning. Before looking into more complicated methods, simpler ideas should be explored. The simplest idea is training the model with very low learning rates, fine-tuning the weights in this process. The configuration displayed in tab.5.1 using the poly learning rate policy with a base learning rate of 1e-4 and a maximum of 100 epochs seems suitable for that.

5.1.3 Final Assumptions and Intermediate Results

In conclusion, a MobileNetV2 model is used as the uncompressed model, where the stride of the first two layers with stride equal 2 is changed to 1, resulting in an output stride of 8. The model is trained with the configuration given in tab.5.1, and pre-processing and data-augmentation is applied to the CIFAR datasets according to sec.3.1.1.

The shunt is replacing blocks 8-12, motivated by the knowledge quotients calculated from the trained, uncompressed model and by the fact, that the compression rate for this setup is matching the rate used in the original experiments. Feature maps are extracted on the fly during the training phase to enable data augmentation for enriching the variety of training data for the shunt connection.

Standard MSE loss is used to train the shunt connection to match the behaviour of original blocks. For this, the hyperparameters in tab.5.3 are used. After inserting the trained shunt into the uncompressed model, replacing the decided blocks, the resulting shunt-inserted model gets fine-tuned using a low learning rate and only a few epochs.

Now, that all the groundwork has been done, shunt connections can be applied to CIFAR10 and CIFAR100. The results are summarized in tab.5.4 similar to the table provided in the original paper.

			Arch 1	Arch 4	Arch 5
Compression of FLOPs			29%	31%	27%
Dataset	Original Model Acc [%]	Acc with shunt connection [%]			
CIFAR10	93.32	Shunt-inserted	90.79	84.93	91.12
		Fine-tuned	91.72	90.11	92.13
CIFAR100	73.70	Shunt-inserted	71.46	66.20	71.85
		Fine-tuned	72.43	70.91	72.50

Table 5.4: First try to reproduce the results of the original paper.

Comparing these results with those of the original paper, it can be seen that our original models provide better accuracies for both CIFAR10 and CIFAR100. The shunt-inserted model after training the shunt on extracted feature maps without fine-tuning, provide comparable accuracies to the original paper. Therefore, this step is considered as being successfully replicated. On the other hand, the fine-tuned models do not reach the accuracy of the uncompressed models as it can be seen in the original paper. This is crucial because this is the final goal when compressing models. Only one hyperparameter configuration has been tested for the fine-tuning step, hence this step shall be further investigated in the next section.

5.1.4 Investigating the 'Fine-Tuning' Step

Training shunt connections on feature maps is a rather straight forward task without much potential of optimization. On the other hand, the fine-tuning step seems it is strongly depending on the hyperparameters used for training. Therefore, a broad study is done to investigate this step, hopefully to find a configuration that restores the accuracy of the uncompressed model.

Tab.5.5 shows a selection of the results of conducted experiments on the CIFAR10 shunt-inserted model. As mentioned previously, the name 'fine-tuning' indicates training with a low learning rate while potentially freezing some layers. The experiments show that this name is not really fitting if not misleading. Good results are only achieved when training with relatively high learning rates without freezing any layers.

The original paper manages to match the uncompressed model's accuracy with the shunt-inserted models after fine-tuning. This is not possible with the standard method or freezing setups used in this thesis. Therefore, another method is introduced to potentially close this gap: knowledge distillation. The big uncompressed model should act as the teacher

Strategy	Learning rate	Epochs	Accuracy [%]
Standard	1e-5	100	89.00
Standard	1e-3	100	91.65
Standard	1e-2	100	92.17
Standard	1e-1	100	91.93
Standard	1e-2	300	92.74
Freeze before shunt	1e-3	100	90.74
Freeze before shunt	1e-1	100	91.53
Freeze except top	1e-4	100	86.68
Freeze except top	1e-2	100	86.89
DK (5T, 4S)	1-e2	300	92.87
DK (8T, 2S)	1-e2	300	92.65

Table 5.5: Results for fine-tuning the shunt-inserted MobileNetV2 model trained on CIFAR10 using different setups. The shunt connection consists of architecture 1. DK stands for using dark knowledge as knowledge distillation with T being the temperature and S the distillation strength.

for the small compressed model like described in sec.2.6. The only disadvantages are, that training times are a bit longer since the second teacher model also has to be computed in each forward pass and that two additional hyperparameters are introduced, which have to be optimized. With this setup, better results are achieved as it can be seen in tab.5.5, although the accuracy of the uncompressed model is still not reached.

Using knowledge distillation for fine-tuning shunt connections can also be motivated by looking at the produced learning curves, when fine-tuning using the standard method. Such typical curves are visualized in fig.5.2. It is clearly visible, that the model starts to overfit heavily right at the beginning. It is shown in [HVD15], that knowledge distillation acts as strong regularization, meaning it can be used to train overfitting models more efficiently.

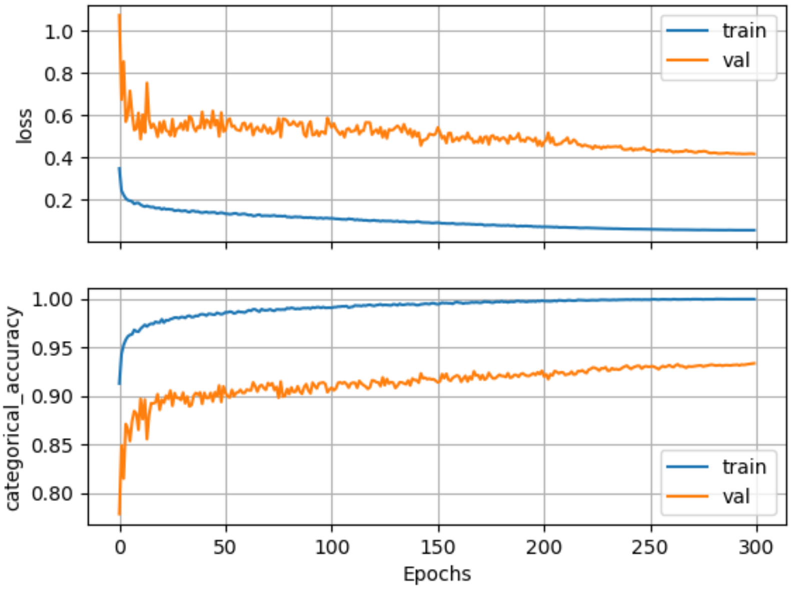


Figure 5.2: Typical learning curves for fine-tuning a shunt-inserted model with the standard method.

So far, only the fine-tuning for the shunt-inserted model trained on CIFAR10 has been investigated. Taking the findings from the previous experiments into account, experiments are conducted also on the CIFAR100 shunt-inserted models. Some notable results are displayed in tab.5.6, where a similar picture as before can be seen. The standard methods including the freezing of layers do not achieve the accuracy of the uncompressed model, although results are overall better compared to ones achieved on CIFAR10. Knowledge distillation once again outperforms other methods, achieving an almost optimal result of 73.68% accuracy.

Strategy	Learning rate	Epochs	Accuracy [%]
Standard	1e-3	300	73.21
Standard	1e-2	300	72.84
Freeze before shunt	1e-2	300	72.45
Freeze except top	1e-2	300	71.98
DK (5T, 4S)	1-e2	300	73.68
DK (8T, 2S)	1-e2	300	73.23

Table 5.6: Results for fine-tuning the shunt-inserted MobileNetV2 model trained on CIFAR100 using different setups. The shunt connections consists of architecture 1. DK stands for using dark knowledge as knowledge distillation with T being the temperature and S the distillation strength.

5.1.5 Conclusion

It would be interesting, how the fine-tuning step was performed in the original shunt connection paper, as the standard fine-tuning methods including the training with low learning rates and the freezing of layers, do not restore the accuracy of the uncompressed model. Maybe this is due to the fact, that a different uncompressed model is used or the feature matching step of the shunt connection is performed differently. Making use of knowledge distillation in the form of dark knowledge, helps to boost results and restores the desired accuracy in the case of CIFAR100. For CIFAR10, this is not achieved, although it can be speculated, that it would be possible, presupposing finding the correct hyperparameter configuration. It is concluded, that knowledge distillation is a strong tool for fine-tuning shunt inserting models.

5.1.6 Inference Measurements

Evaluating the real speed-up of shunt insertions is performed on the NVIDIA Jetson Xavier like described in sec.3.5.2 including the workflow for creating the TensorRT engine from a Keras model. Measurements are done without using any DLA cores since inference times are lower using only the GPU. Inference is measured for the full FP32 precision and all three quantization configurations. The results are summarized in tab.5.7. It has to be kept in mind, that TensorRT does some complicated optimization techniques for each model separately, which can result in non-linear behaviour.

When comparing the results for the full FP32 inference and the quantized FP16 one, some non-linear behaviour can already be observed. Inference times for FP32 precision show an above linear speed-up for all three shunt-inserted models. The speed-up for FP16 quantized models, is much slower, meaning that the uncompressed model benefits more from the quantization. One could guess that inference measurements include some constant term, like transferring data back and forth between the CPU and GPU, which is independent of the measured model. This would result in less relative speed-up for lower inference times. But looking at the speed-ups of INT8 quantized models, they are actually larger than the FP16 counterparts, contradicting this hypothesis.

In conclusion, it can only be assumed that TensorRT can optimize some models better than others when using a certain quantization strategy. The goodness of the results shall be highlighted nevertheless. A linear speed-up is very desired when compressing models, since this guarantees that the compressed models are not only efficient on paper, but also on real hardware. It makes sense, that shunt connections translate compressed MACs nicely to real inference speed-

Model	Compression rate	FP32 [ms]	FP16 [ms]	INT8 [ms]
Uncompressed	-	1.37	0.75	0.66
Shunt-inserted-arch1	29%	0.93 (-32%)	0.59 (-21%)	0.51 (-23%)
Shunt-inserted-arch4	31%	0.88 (-36%)	0.57 (-24%)	0.50 (-24%)
Shunt-inserted-arch5	27%	0.95 (-31%)	0.64 (-15%)	0.53 (-20%)

Table 5.7: Inference measurements of shunt-inserted MobileNetV2 models trained on CIFAR10.

up since the compressed models hold fewer layers, resulting in fewer kernel calls. Comparing that to the example of pruning a model, where calculations get only more sparse and cannot be fully avoided, shunt connections seem like to have the upper hand in regard to this metric.

5.2 MobileNetV3 Experiments

In the previous section, it was shown how to replicate the results from the original paper by compression MobileNetV2. The acquired methods shall now be applied to the newer MoibleNetV3, while trying to push the boundaries of the method at the same time. For this, the CIFAR datasets are once again used to build a simple classification model.

5.2.1 Generating the Baseline Model

To push the boundary of the method, the MobileNet model should be already compressed by using a small depth multiplier before applying shunt connections. This results in a baseline model, which should be much harder to compress and therefore challenges shunt connections.

A depth multiplier of 0.5 is chosen, since it is observed that it achieves almost the same accuracy on the test set compared to the full size model. This work uses the MobileNetV3 model implemented in Keras in the `applications` module. The authors claim that it achieves state-of-the-art results on ImageNet, validating the model. It only has to be modified through adding regularization to all layers, which can be easily done by the `modify_model` method, which is described in sec.4.4.

5.2.2 Training the Original Model

The CIFAR datasets get again pre-processed and augmented like described in sec.3.1.1.

The hyperparameter configuration used for generating the MobileNetV2 baseline acts as a starting point for finding a good training setup for the reduced MobileNetV3 network. The full configuration, which is used to train the baseline is shown in tab.5.8. This configuration yields 91.93% accuracy for CIFAR10 and 67.10% accuracy for CIFAR100, matching almost the accuracies seen with the full size MobileNetV2.

5.2.3 Knowledge Quotients and Shunt Locations

Shunt locations are again chosen by looking at the knowledge quotients of the original model. The quotients of the models trained on CIFAR10 and CIFAR100 are shown in tab.5.9. One can see, that those obtained from the CIFAR10 model are a bit higher compared to the knowledge quotients of the full size MobileNetV2 trained on CIFAR10. This is expected, since this time the original model has already be a bit compressed by using a low depth multiplier, which

Hyperparameter	Value
regularization factor	4e-5
dropout rate	0.2
learning rate policy	poly
power	0.9
lr_{base}	1e-2
max_epochs	350
optimizer	SGD
momentum	0.9
batch size	64

Table 5.8: Default hyperparameter configuration used for training MobileNets on CIFAR datasets

results in less redundancy inside the model. Nevertheless, quotients are really low for all blocks except the earlier ones, meaning that the shunt connection can be chosen almost freely.

The CIFAR100 model on the other hand has one block which is very important for the model's performance. Deleting block 5 results in an accuracy drop of over 50%, hence it should be avoided to replace this block.

To stretch the boundaries of the shunt connection method, blocks 5-11 are chosen to be replaced in the case of CIFAR10 and blocks 6-11 should be replaced in the CIFAR100 model. Using architecture 4 as the shunt, the CIFAR10 model results in a compression rate of 42% of MAC operations, while the CIFAR100 model compresses 31% of MACs.

Block ID	KQ - CIFAR10	KQ - CIFAR100
1	-	-
2	-	-
3	0.12	0.25
4	-	-
5	0.13	0.54
6	0.06	0.16
7	0.02	0.09
8	0.01	010
9	-	-
10	0.03	0.24
11	0.03	0.15

Table 5.9: Knowledge quotients for the MobileNetV3 model trained on the CIFAR datasets

5.2.4 Results of Shunt Connections

A similar training policy to the one for shunts on MobileNetV2 is used, which is described in the previous section. The shunt gets trained on feature maps for 100 epochs using the plateau policy, while the fine-tuning is performed using the poly learning rate policy with a base learning rate of 1e-3 for 300 epochs. It has been observed, that this setup produces the best result for fine-tuning using the standard method, the 'freeze before shunt' method', and when using dark knowledge. The results are summarized in tab.5.10. Shunt-inserted models' accuracies without fine-tuning pretty much match the results of shunts on MobileNetV2, while the standard and freeze method for fine-tuning yield even worse results for the case of MobileNetV3. But the fine-tuning using dark knowledge turns out to be highly effective, boosting the accuracy of the final model by over 3 percentage points compared to the standard method in the case of CIFAR10. For CIFAR100, the fine-tuned model using dark knowledge even beat the uncompressed model, which is probably due to the regularization effect of applying knowledge distillation, which is especially useful in the overfitting case of CIFAR100.

	Acc. CIFAR10 [%]	Acc. CIFAR100 [%]
Original model	91.93	67.10
Shunt-inserted models		
	79.78	53.56
Fine-tuned models with standard shunt		
Standard	88.09	64.63
Freeze before shunt	85.66 (-2.43)	60.04 (-4.59)
DK (T=5, λ=2)	**91.36 (+3.27)**	**67.54 (+2.91)**

Table 5.10: Results for shunt connections for MobileNetV3 trained on the CIFAR datasets

It can be concluded that shunt connections are also effective for compressing MobileNetV3, even when comrpressing an already rather small model.

5.3 Semantic Segmentation Experiments

So far, shunt connections have been only applied to the task of classification. To test shunt connections on a wider scope, they should be applied to the Cityscapes dataset to compress a model doing semantic segmentation. As a first step, a baseline using the MobileNetV3 segmentation model has to be trained. Then, the knowledge quotients are calculated to decide where to place the shunt connection. This shunt connection is again trained on feature maps and the fine-tuning of the shunt-inserted model is investigated like in sec.5.1.4. This compression using shunt connections is compared against the built-in compression of MobileNetV3 in the form of the depth multiplier parameter.

5.3.1 Generating the Baseline Model

As the segmentation model, the MobileNetV3 segmentation variant shall be used, which was proposed along its classification version. This variant of the DeeplabV3+ architecture is described in detail in sec.2.7.1.

Regarding the used dataset for all experiments, the Cityscapes dataset is one of the most used semantic segmentation datasets in research. Its properties are described in sec.3.1.2.

The Keras version of the MobileNetV3 segmentation model, which was implemented in part of this thesis, is able to use the pre-trained weights from the original MobileNetV3 repository (see sec.4.1.2). The model using these weights represents a great baseline for further experiments since it can be safely assumed to be trained nearly optimally. But one has to make sure to be able to replicate this training behaviour exactly because it is necessary to perform fine-tuning on the shunt-inserted model in an efficient manner. If one could not replicate the original training procedure, the accuracy of the compressed model is most likely unsatisfying, leading to wrong conclusions.

The original MobileNetV3 paper uses a 4x4 TPU Pod [Jou+17] in synchronous mode for all its experiments. It is also stated under the 'Classification' section that the `RMSPropOptimizer` from Tensorflow is used as the optimizer. The semantic segmentation model is trained using the same training procedure as in the DeepLabV3 paper [Che+17c]. One reads the following information about training the DeeplabV3 architecture on Cityscapes:

- 90k training iterations are performed

- Batch normalization layers are trained on $\text{OS} = 16$ with a batch size of 16, then freezed for fine-tuning with $\text{OS} = 8$
- Images are cropped to a size of 769x769
- Inference and evaluation are performed on the original image sizes
- Poly learning rate with $\text{power} = 0.9$ and base learning rate of 0.007 is used
- Final logistics of the model are upsampled for comparison to the ground truth labels

The 90k training iterations have to be translated to epochs when training using Keras. The following formula is used for this:

$$\text{epochs} = \text{iterations} \quad / \quad \frac{\text{number of samples}}{\text{batch size}}. \tag{5.1}$$

For a batch size of 16, 90k iterations would correspond to around 450 epochs, which is reasonable.

The second source of the training procedure is the official TensorFlow repository containing the training code for training the MobileNetV3 variant on Cityscapes using an ouput stride of 32, in contrast to the output stride of 16 which is used in the original MobileNetV3 paper. Unfortunately, the code seems to contradict some points stated in the paper. In the repository, it is not stated that batch normalization has been frozen during the training of their model. This step may not be necessary if you can fit the model with a big batch size and an output stride of 8 onto your hardware. The biggest discrepancy is probably, that the final logistics are not upsampled to the size of the unmodified ground truth labels. Labels get downsampled and compared to the prediction of the model at the scale according to the model's output stride. This fact is clearly visible in the pre-processing code for the Cityscapes dataset, which is described in detail in sec.3.1.2.

Since the official repository provides the code for training the MobileNetV3 on Cityscapes and acts as the point of comparison of this work, it is considered as the stronger source compared to the MoibleNetV3 paper, which is a bit vague while describing the training procedure for the semantic segmentation task.

Regarding the crop size during training, a clear source why a quadratic input size is preferred could not be found. It also makes sense to keep the aspect ratio of the original image while resizing input images since the segmentation head uses pooling sizes and strides resembling this aspect ratio. Therefore, it is also tested to train with a crop size of 513x1025.

The hardware used for training has also be considered as a possible factor for different results. Google Brain used a 4x4 TPU Prod, which is not available for this work. A multi-gpu setup can mimic their setup, with the exception, that multi-gpu training in Tensorflow uses a not fully synchronous approach like described in sec.4.5. Tasks like semantic segmentation, which rely heavily of the correct training of batch normalization layers, may be influenced negatively by an asynchronous approach. Luckily, in [Che+17c], this effect had been investigated and it turned out that asynchronous training does not have an effect on their DeeplabV3 model, when training asynchronously on many GPUs. In conclusion, the setup on the VSC is considered as suitable for training the MobileNetV3 segmentation architecture on Cityscapes.

5.3.2 Training MobileNetV3 on Cityscapes

Combining these two sources, a first approach for training the MobileNetV3 segmentation model on Cityscapes can be described. A model using an input size of 769x769 is trained using an output stride of 8. The batch size is chosen to be as high as possible, resulting in a batch size of 24 on a multi-GPU node of the VSC-3. Keras' `RMSprop` optimizer is used with a momentum of 0.9. The learning rate policy is chosen to be *poly* with a base learning rate of 0.007 and a power parameter of 0.9. The model is trained for 500 epochs and evaluated using the same weights with an output stride of 32. The result of this experiment and a selection of the results of following configurations are summarized in tab.5.11.

It has been observed, that the `RMSprop` optimizer is not performing well, producing worse mIoU scores compared to the standard SGD optimizer. A higher learning rate combined with the 513x1025 input size, seems to produce the best results, although the reported result from the original repository of an mIoU score of 68.99 is still out of reach. No tested hyperparameter configuration matched these results. The highest validation score is achieved by training the model with the SGD optimizer with a base learning rate of 5e-2, an input size of 513x1025 and an output stride of 8. Since the final goal is to investigate shunt connection on this baseline model, the trainings method used for creating this model has to be well understood. Therefore it is decided, that the self-trained model is used as the uncompressed model for all following shunt experiments, although it is not performing to its full potential.

Input size	Output stride	Optimizer	Base learning rate	mIoU
769x769	8	RMSprop	7e-3	41.13
769x769	8	SGD	7e-3	47.40
769x769	32	SGD	5e-2	58.94
769x769	8	SGD	5e-2	56.46
513x1025	16	SGD	5e-2	56.46
513x1025	**8**	**SGD**	**5e-2**	**59.50**

Table 5.11: Results for different MobileNetV3 models trained on Cityscapes using different training setups

Some prediction using the 59.50-mIoU-model can be seen in fig.5.3. Comparing them to the ground truth segmentation masks, most objects are correctly detected, although boundaries around objects are often blurry. Very thin objects like lantern poles are often missed, indicating, that the training process might still be improvable in that regard.

5.3.3 Knowledge Quotients and Shunt Locations

The knowledge quotients for the MobileNetV3 segmentation model trained on Cityscapes are written down in tab.5.12. The values are not close to 0, like it was the case for CIFAR10, but they are in the same regime as the quotients for CIFAR100. For CIFAR100 shunt connections were still possible, hence it reasonable to assume that shunt connections can also compress this network efficiently.

Blocks 5 to 11 are in the range of being replaced by shunt connections, since all of them hold roughly the same knowledge quotient. Different setups will be used later on based on this implementation.

It should be highlighted, that shunt connections need to mimic the receptive field of the original model. This means that the shunt must have the same downsampling behaviour as the original model and, in the case of the DeeplabV3 architecture, also the same distillation rates. For example, assuming a MobileNetV3 backbone with an output stride of 8, when replacing blocks 5 to 11, the shunt connections would need to apply atrous convolutions with the correct rates

Block ID	Knowledge Quotients
1	-
2	-
3	-
4	-
5	0.13
6	0.15
7	-
8	0.20
9	-
10	0.20
11	0.19

Table 5.12: Knowledge quotients for the MobileNetV3 segmentation model trained on Cityscapes.

at the corresponding depthwise separable convolutional layers.

5.3.4 Shunt Connection Experiments

Training shunt connections on semantic segmentation models is conceptually not different from training them on classification models. Therefore, the same procedure as in sec.5.1.2 can be safely used. They only additional hyperparameter is the output stride which is used during training. A quick experiment is conducted to see which output stride suits the best for shunt training. The results for the same shunt connection trained using different output strides is shown in tab.5.13. These results are obtained by using again a batch size of 24, while training with the plateau learning rate policy for 100 epochs. It can be seen that the results are quite similar, although different output strides are used. Besides providing the best mIoU for the shunt-inserted model, using an output stride of 32 also has the advantage that training times are a little bit lower, because feature maps are downsampled more often, resulting in fewer computations. Testing all possible output strides for every following shunt experiment is not feasible, and so it is decided to use output stride 32 for all following shunt trainings.

Output stride	mIoU of shunt-inserted model
8	34.47
16	34.23
32	34.90

Table 5.13: Training a shunt connection using different output strides. The validation loss is measured during training on the half input size and the same output stride, which was used during training. The mIoU is measured using an output stride of 32 and the full image size.

After training the shunt connections on extracted feature maps, the fine-tuning step has to be performed. Besides the 'standard' way of fine-tuning the model using only a rather low learning rate, the model should also be fine-tuned by applying knowledge distillation in the form of adoptive cross-entropy, which is explained in sec.2.6.2. The goal of the fine-tuning step is to match the mIoU score of the original model. The same hyperparameters are used for both methods: the *poly* learning policy with a base learning rate of 1e-2, an output stride of 32, a batch size of 12 and a number of epochs of 500. Batch size is reduced for this experiment, since knowledge distillation basically requires two models to be run simultaneously resulting in a much higher memory footprint. The standard fine-tuning method can be run with a batch size of 24 on a multi-GPU node of the VSC, but is also performed with a batch size of 12 to ensure comparability of both fine-tuning methods. It is possible, that results would be better on different hardware, where a higher amount of VRAM is available.

The results for all three shunt connection setups are shown in tab.5.14. The mIoU scores of the shunt-inserted models are interesting, since the shunt connection compressing 15% of the model's MACs yields the worst results. This is probably due to the fact, that for this setup Arch4 is used as the shunt model, which holds only one single block compared to the two blocks Arch1 consists of, which is used for both other setups (see tab.2.7).

The original model's mIoU score of 59.50 can not be achieved by a single setup. Compressing the model by 15% yields already a drop of around 3 points, when doing the standard fine-tuning procedure. Applying knowledge distillation to the same setup, results get even worse, yielding a drop of around 3.5 points. In contrast, for the second setup compressing the model by 28%, knowledge distillation helps to boost the mIoU score of the fine-tuned model by a whole point, resulting again in a drop of around 3.5 points. A visual comparison of the predictions of the uncompressed model and this compressed model is given in fig.5.3. It can be observed, that objects are still visible in the compressed predictions, but boundaries are a bit more blurry compared to the uncompressed predictions. The third setup, which compresses the original model's MACs by 39%, again does not benefit from knowledge distillation in the form of ACE.

	8-11-Arch4	6-11-Arch1	5-11-Arch1
Compression rate	15%	28%	39%
Shunt-inserted models mIoU			
	34.84	37.97	35.13
Fine-tuned models with standard shunt - mIoU			
Standard	**56.52**	54.91	**51.83**
ACE ($\kappa = 0.3$)	56.03 (-0.49)	**55.98 (+1.07)**	51.14 (-0.69)

Table 5.14: Results for compressing MobileNetV3 trained on the Cityscapes dataset.

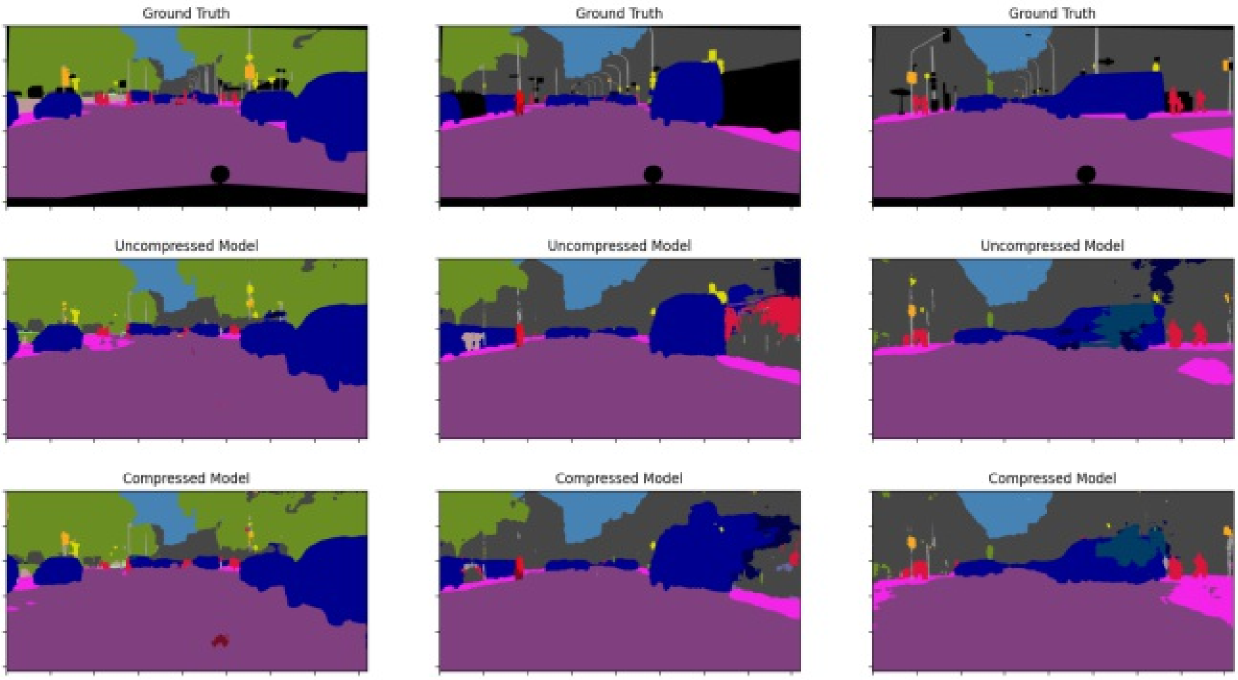


Figure 5.3: Visual comparison of ground truth labels and predictions by the uncompressed and compressed model. The compressed model is compressed using the 6-11-Arch1 setup and fine-tuned using ACE.

In conclusion, shunt connections are applicable to large scale datasets like Cityscapes, although the mIoU score of the uncompressed models could not be matched by the compressed models using shunt connections. Fine-tuning depends heavily on the chosen hyperparameters, which can barely be optimized for a difficult task like semantic segmentation on Cityscapes, when the hardware resources are not available. It has been shown, that the ACE method can improve performance of the shunt-inserted models, although not all setups benefit from it. To catch the whole picture of applying knowledge distillation to shunt connections on segmentation tasks, more knowledge distillation methods still need to be investigated.

5.3.5 Comparison With MobileNetV3's Compression using Depth Multipliers

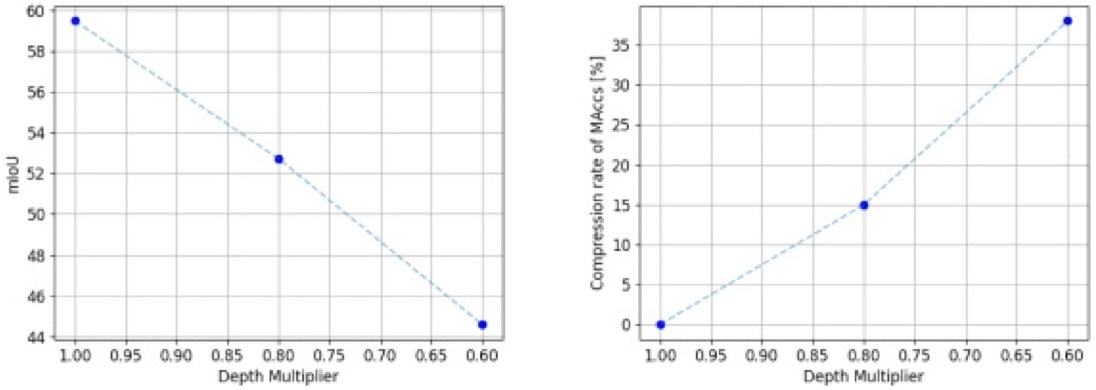
Compressing MobileNetV3 cannot only be done using complicated compressing algorithms like shunt connections, but smaller versions of the model can also be generated using the built-in depth multiplier parameter. This parameter creates slimmed versions of the original architecture and was already used in sec.5.2.1 to create smaller models for CIFAR. The bare minimum goal of any compressing technique is to beat the compression through the depth multiplier.

For comparison with the shunt-inserted models, two additional variants of the MobileNetV3 model are trained on Cityscapes using a depth multiplier of 0.8 and 0.6. The same training procedure as for the full model is used for these models. The accuracy results and rate of compression regarding MACs are shown in tab.5.15 and visualized in fig.5.4.

Depth multipler	mIoU	Compression of MACs
1.0	59.50	0%
0.8	52.73	15%
0.6	44.59	38%

Table 5.15: Comparison of mIoU and compression rate for MobileNetV3 using different depth multipliers.

Both graphs show a almost linear behaviour, which is obviously expected from the compression rate. Regarding the change of the mIoU, it is expected that it drops for a lower depth multiplier, but predicting the exact change is not trivial if possible. It should be highlighted, that no additional hyperparameter search was conducted for the training of these models. It is assumed that the configuration used for the original model also provides competitive results for models with a lower depth multiplier.



(a) Poly

(b) Plateau

Figure 5.4: Results for MobileNetV3 trained on Cityscapes using differnt depth multipliers.

This values can now be well compared to the results of applying shunt connections to the MobileNetV3 segmentation model, since shunt connections are setup to match the compression rates of the depth-multiplier-models. Fig.5.5 plots the relationship between MAC operations and mIoU results for models being compressed using shunt connections or depth multipliers. It is clearly visible, that shunt connections outperfom depth multipliers by a large margin. While the depth-multiplier-models drop off almost linearly, shunt connections do not show linear behaviour, as the compressed model using 140 million MACs holds almost the same mIoU score as the model with 166 million MACs. A bigger jump can be seen when compressing the model to 120 million MACs.

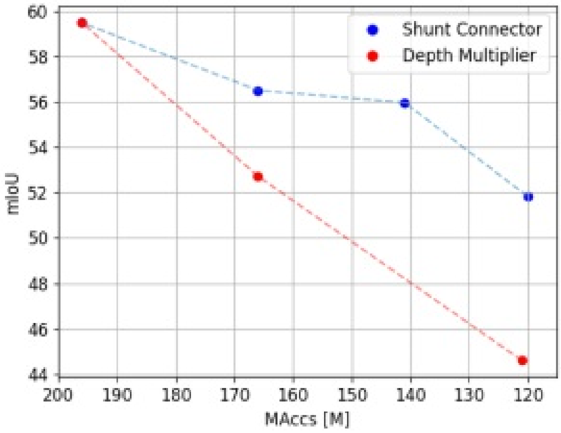


Figure 5.5: Comparison between compression results of depth mutlipliers and shunt connections.

5.3.6 Inference Speed-up of Shunt Connections

It has been seen in sec.5.1.6, that the compression using shunt connections almost linearly translates to real inference speed-up on the NVIDIA Jetson Xavier. Tab.5.16 shows the inference results on the same device for the compressed models using depth multipliers and shunt connections, from the previous section. The workflow for benchmarking the models described in sec.3.5.2 is used again.

First, it can be seen that the quantization of the models does not provide a large benefit, which is unexpected. All compressed models also do not meet ideal linear speed-up. The best trade-off generates the 6-11-Arch1 shunt-inserted model, where half of compressed MACs contribute translates into the inference times. Nevertheless, it can be seen that shunt connections still provide a larger speed-up compared to their counterpart models using depth multipliers. This is considered a big advantage of shunt connections.

Model	Compression rate	FP32 [ms]	FP16 [ms]	INT8 [ms]
Uncompressed	-	80	76	78
0.8-depth-mult.	15%	77 (-4%)	74 (-3%)	76 (-3%)
0.6-depth-mult.	38%	71 (-11%)	71 (-7%)	73 (-6%)
8-11-Arch4	15%	74 (-8%)	71 (-7%)	73 (-6%)
6-11-Arch1	28%	69 (-14%)	67 (-12%)	68 (-13%)
5-11-Arch1	39%	66 (-18%)	64 (-16%)	65 (-17%)

Table 5.16: Inference measurements of shunt-inserted MobileNetV3 segmentation models trained on Cityscapes. Speed-ups compared to the uncompressed model are written in parenthesis.

Since these inference times are generally pretty unexpected, it should be considered to redo those measurements using maybe a different ONNX opset or different TensorRT settings. Doing this requires great insight into TensorRT's backend, which is out of scope for this thesis.

Chapter 6

Conclusion

Shunt connections promise efficient compression of residual CNNs by replacing a big part of the model with a small CNN called 'shunt'. The original paper introducing shunt connections showed that their method can compress MobileNetV2 models trained on small classification datasets without losing any accuracy. This thesis investigates the general applicability of shunt connections by testing their performances on MobileNetV3 on CIFAR and Cityscapes. For this, a framework in Keras has to be implemented to test different shunt connection setups in an efficient manner and validate results by measuring the inference speed-up on the NVIDIA Jetson Xavier.

Reproducing the results of the original paper resembles a big challenge since many import details are missing from the original paper, especially concerning the final fine-tuning step. It is shown, that this name is misleading because higher learning rates lead to better results compared to usual training setups, which would be regarded as fine-tuning. Using a standard training method yields great overfitting, such that the fine-tuned models are not matching the accuracy of the uncompressed model. Knowledge distillation in the form of dark knowledge can help to boost these accuracies, although the desired accuracies are still not reached fully. Testing the real speed-up of compressed models using shunt connections on the NVIDIA Jetson Xavier and TensorRT, it is seen that even more than linear speed-ups are achieved, when not quantizing the models, which is a big win for shunt connections

Applying shunt connections to the newer MobileNetV3-Small variant, produces competitive results, even when looking at the more difficult case of the already compressed model trough a small depth multiplier. The fine-tuning using knowledge distillation is even more important in this case. With it, the accuracies of the uncompressed model is not only matched but even exceeded. Conclusively, shunt connections can compress MobileNetV3 models on classification tasks very efficiently, even when replacing around 40% of MAC operations.

A potentially even more difficult task for shunt connections, is compressing models trained on large-scale problems, like semantic segmentation on Cityscapes. A complicated segmentation variant of the MobileNetV3 model described by its original paper, was re-implemented in Keras, such that shunt connections can be applied to it. Compressing this model trained on Cityscapes, yields mixed results, where compression rates up to 40% are achieved while losing some mIoU points. The fine-tuning step was again tried to be improved by applying knowledge distillation, this time in the form of the ACE loss. In some cases, this method bettered results, while in other cases, results even got worse. To grasp the big picture, several different knowledge distillation methods for semantic segmentation have to be investigated.

Shunt connections on the semantic segmentation task were compared to the built-in compression technique of MobileNetV3 in the form of depth multipliers. It is shown that shunt connections not only provide better mIoU scores for a given compression rate, but also produce models with faster inference times. It can be concluded, that shunt connections can also be used for large scale problems, although performances of models may suffer. Nevertheless, they should definitely be preferred over the native compression technique of depth multipliers as shunt connections outperform it by all metrics.

Shunt connections still need to be evaluated compared to other neural network compression techniques like quantization or pruning both in terms of model accuracy and inference speed on specific hardware. To ensure a correct comparison on the NVIDIA Jetson Xavier, models have to be converted optimally through TensorRT. This conversion was probably not done optimally in this work in the case of the MobileNetV3 segmentation model. Hence, this is still an open problem. Another open task resembles the design of the shunt architecture itself. In this work, the architectures from the original paper are used, without exploring the design space. There may be a much more efficient architecture available, which could potentially be found using an architecture search approach similar to how MobileNetV3 was obtained.

This work provides the first framework for neural network compression for the Christian Doppler Laboraty for Embedded Machine Learning. It will be used as a point of comparison for other compression techniques to find the best model for a certain application and hardware. It will potentially be used in real applications including an autonomous drone project.

Bibliography

[AV19] R. Avenash and P. Viswanath. "Semantic Segmentation of Satellite Images using a Modified CNN with Hard-Swish Activation Function". In: *VISIGRAPP*. 2019.

[CBD16a] M. Courbariaux, Y. Bengio, and J.-P. David. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*. 2016.

[CBD16b] M. Courbariaux, Y. Bengio, and J.-P. David. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*. 2016.

[Che+16] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. *Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs*. 2016.

[Che+17a] L. Chen, H. Fei, Y. Xiao, J. He, and H. Li. "Why batch normalization works? a buckling perspective". In: *2017 IEEE International Conference on Information and Automation (ICIA)*. 2017, pp. 1184–1189.

[Che+17b] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. 2017.

[Che+17c] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. *Rethinking Atrous Convolution for Semantic Image Segmentation*. 2017.

[Che+18] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam. *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. 2018.

[Che+20] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. *A Survey of Model Compression and Acceleration for Deep Neural Networks*. 2020.

[Cho+15] F. Chollet et al. *Keras*. https://keras.io. 2015.

[Cor+16] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. *The Cityscapes Dataset for Semantic Urban Scene Understanding*. 2016.

[Fau] K. M. Fauske. *TeXample.net. Example: Neural Network*. https://texample.net/tikz/examples/neural-network/.

[GHP07] G. Griffin, A. Holub, and P. Perona. "Caltech-256 Object Category Dataset". In: *CalTech Report* (Mar. 2007).

[Gon+14] Y. Gong, L. Liu, M. Yang, and L. Bourdev. *Compressing Deep Convolutional Networks using Vector Quantization*. 2014.

[Gou+21] J. Gou, B. Yu, S. J. Maybank, and D. Tao. *Knowledge Distillation: A Survey*. 2021.

[HBF18] M. Hussain, J. Bird, and D. Faria. "A Study on CNN Transfer Learning for Image Classification". In: June 2018.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015.

[Her+17] L. Hertel, E. Barth, T. Käster, and T. Martinetz. *Deep Convolutional Neural Networks as Generic Feature Extractors.* 2017.

[Hoc98] S. Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (Apr. 1998), pp. 107–116.

[How+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.* 2017.

[How+19] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. *Searching for MobileNetV3.* 2019.

[Hu+19] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu. *Squeeze-and-Excitation Networks.* 2019.

[HVD15] G. Hinton, O. Vinyals, and J. Dean. *Distilling the Knowledge in a Neural Network.* 2015.

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015).

[Jac+18] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2018.

[Jou+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. *In-Datacenter Performance Analysis of a Tensor Processing Unit.* 2017.

[KB17] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization.* 2017.

[Kri12] A. Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: *University of Toronto* (May 2012).

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105.

[KSL19] S. Kornblith, J. Shlens, and Q. V. Le. *Do Better ImageNet Models Transfer Better?* 2019.

[Lec+98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE.* 1998, pp. 2278–2324.

[Li+13] F. Li, R. Fergus, P. Perona, and D. M. S. Zekrifa. "Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories". In: *Computer Vision and Image Understanding* 106 (Apr. 2013), pp. 59–70.

[LRB15] W. Liu, A. Rabinovich, and A. C. Berg. *ParseNet: Looking Wider to See Better.* 2015.

[LSD15] J. Long, E. Shelhamer, and T. Darrell. *Fully Convolutional Networks for Semantic Segmentation.* 2015.

[Mar+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey

Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015.

[Min+20] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos. *Image Segmentation Using Deep Learning: A Survey.* 2020.

[NVI] NVIDIA. *NVIDIA Jetson AGX Xavier - Vendor site.* https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[PH20] S. Park and Y. Heo. “Knowledge Distillation for Semantic Segmentation Using Channel and Spatial Correlations and Adaptive Cross Entropy”. In: *Sensors* 20 (Aug. 2020), p. 4616.

[Ros58] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6 (1958), pp. 386–408.

[Rud17] S. Ruder. *An overview of gradient descent optimization algorithms.* 2017.

[Rus+15a] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge.* 2015.

[Rus+15b] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[RZL17] P. Ramachandran, B. Zoph, and Q. V. Le. *Searching for Activation Functions.* 2017.

[San+19] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. *MobileNetV2: Inverted Residuals and Linear Bottlenecks.* 2019.

[SB15] S. Srinivas and R. V. Babu. *Data-free parameter pruning for Deep Neural Networks.* 2015.

[Sei17] F. Seide. “Keynote: The computer science behind the Microsoft Cognitive Toolkit: An open source large-scale deep learning toolkit for Windows and Linux”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2017, pp. xi–xi.

[Sob14] I. Sobel. “An Isotropic 3x3 Image Gradient Operator”. In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014).

[STA19] B. Singh, D. Toshniwal, and S. Allur. “Shunt connection: An intelligent skipping of contiguous blocks for optimizing MobileNet-V2”. In: *Neural Networks* 118 (June 2019).

[SZ14] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014).

[Tan+18] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu. *A Survey on Deep Transfer Learning.* 2018.

[Tan+19] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. *MnasNet: Platform-Aware Neural Architecture Search for Mobile.* 2019.

[Tea+16] T. T. D. Team et al. *Theano: A Python framework for fast computation of mathematical expressions.* 2016.

[VD09] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[VSM11] V. Vanhoucke, A. Senior, and M. Z. Mao. “Improving the speed of neural networks on CPUs”. In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[VWB16] A. Veit, M. Wilber, and S. Belongie. *Residual Networks Behave Like Ensembles of Relatively Shallow Networks*. 2016.

[Wer90] P. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78 (Nov. 1990), pp. 1550–1560.

[Yan+18] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. *NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications*. 2018.

[Yas18] R. Yasrab. “ECRU: An Encoder-Decoder Based Convolution Neural Network (CNN) for Road-Scene Understanding”. In: *Journal of Imaging* 4 (Oct. 2018), p. 116.

[Zhu+19] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu. *Discrimination-aware Channel Pruning for Deep Neural Networks*. 2019.

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe d er Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria

Bernhard Haas

Bernhard Haas