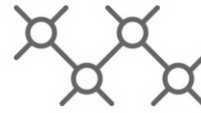




TECHNISCHE
UNIVERSITÄT
WIEN



Institut für
Computertechnik
Institute of
Computer Technology

Master's Thesis

submitted by

Andreas Glinserer

Registration Number 01525864

Autopruning with Intel Distiller and Evaluation on a Jetson Xavier AGX

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria, 2021

Study code:

066 504

Field of study:

Embedded Systems

Supervisor:

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

Co-Supervisor:

Projektass. Dipl.-Ing. Martin Lechner BSc

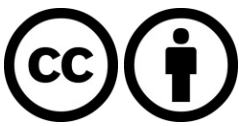
Copyright (C) 2021 Andreas Glinserer

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

```
@Thesis{Glinserer2021,
  type      = {Master's Thesis},
  author    = {Andreas Glinserer},
  title     = {Autopruning with Intel Distiller and Evaluation on a Jetson Xavier AGX},
  school    = {Vienna University of Technology (TU Wien)},
  year      = {2021},
  address   = {Gusshausstrasse 27--29 / 384, 1040 Wien},
  month     = {November},
}
```

Contact me:

author@institution.domain



This thesis is licensed under the following license: Attribution 4.0 International (CC BY 4.0)

You are free to:

1. Share – Copy and redistribute the material in any medium or format
2. Adapt – Remix, transform, and build upon the material for any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

The entire license text is available at: <https://creativecommons.org/licenses/by/4.0/legalcode>

Abstract

Convolutional neural networks build the basis for state-of-the-art computer vision tasks, such as image classification or object detection. Tasks such as collision detection, e.g., cars or drones, or the classification of wildlife or mushrooms to identify harmful or poisonous vegetation require these networks to run on smaller, more power efficient devices. Such edge devices have much less computational power than servers with high-power GPUs and often run only on batteries. The desire is to achieve real-time performance while maintaining a minimum power consumption to enable execution on edge devices without any uplink connection. Pruning is a valuable method to compress neural networks by reducing the required computations and thus improve performance. This work presents an automatic pruning workflow using a measurement-based method to determine which portions of the network contribute how much to the total accuracy. This accuracy contribution can then be used to prioritise parts of the network for pruning, i.e., only portions of the network are pruned, which contribute little to the total accuracy. This approach is interesting, especially for convolutional neural networks. Furthermore, to increase the prunability within networks containing residual blocks, this work evaluates zero-padding as a helpful complement to existing pruning methods. Residual blocks are a basic building block within modern neural networks. With zero-padding added to the pruning, we enable the automatic pruning process to choose layers for pruning, which would otherwise not be possible or only possible with removing additional filters that might significantly impact the total accuracy. This would result in a significant loss regarding the accuracy and, therefore, also the prunability. Zero-padding adds the removed channels back into the original output feature map in a manner such that the shapes of the feature maps remain identical, but computations are still saved. Using this method, a speedup of up to 21% on CPU-based platforms and 5-6% for GPU-based executions of a MobileNetV2 was achieved. For evaluation, a Jetson Xavier AGX is chosen because it offers three different execution modes and thus is a solid baseline for further argumentation regarding the applicability. The pruned network is comparable to the original network with an applied depth multiplier. However, the required additional retraining time is much smaller than training a MobileNetV2 with a depth multiplier from scratch. Furthermore, the effect of pruning with regards to power consumption highly depends on the network architecture. Therefore, the resulting power ranges from actual savings to an even higher consumption depending on the executed network.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Convolutional Neural Networks bilden die Grundlage für bildbezogene Aufgaben, wie z. B. Klassifikation oder Objekterkennung. Da diese eine hohe mathematische Komplexität aufweisen, ist es wünschenswert, diese auch auf weniger leistungsfähiger Hardware nutzbar zu machen. Durch Aufgaben, wie Hinderniserkennung in Hinblick auf autonome Autos oder Drohnen, beziehungsweise der Klassifizierung von Wildtieren oder Pilzen zur Erkennung von schädlicher oder giftiger Vegetation, entsteht ein Bedarf, diese Netzwerke auf kleinere Geräte zu bringen, welche auch nur mit Batterien laufen. Diese Geräte haben meist nur eine geringe Rechenleistung im Vergleich zu Servern mit Hochleistungs-GPUs. Der Wunsch ist es, Echtzeitgeschwindigkeit zu erreichen und dabei ein Minimum an Stromverbrauch zu erzielen, um die Ausführung auf Geräten zu ermöglichen, welche sich nahe am sogenannten edge befinden oder überhaupt keine Möglichkeit für eine Netzwerkanbindung haben. UPruning ist eine wertvolle Methode, um neuronale Netze durch Reduzierung der erforderlichen Berechnungen zu komprimieren und so die Leistung zu verbessern. In dieser Arbeit wird ein automatischer Pruning-Workflow vorgestellt, der auf einem messbasierten Ansatz gründet, um zu bestimmen, welche Teile des Netzes wie viel zur Gesamtgenauigkeit beitragen. Dieser Genauigkeitsbeitrag kann dann verwendet werden, um Teile des Netzes für das Pruning zu priorisieren, d.h. es werden nur Teile des Netzes beschnitten, die wenig zur Gesamtgenauigkeit beitragen. Dieser Ansatz ist vor allem für CNNs interessant. m die Prunebarkeit innerhalb von Netzen mit Residual-Blöcken zu erhöhen, wird in dieser Arbeit Zero-Padding als hilfreiche Ergänzung zu bestehenden Pruning-Methoden evaluiert. Residual-Blöcke bilden einen Basisbaustein für neurale Netze. Durch das Hinzufügen von Zero-Padding zum Pruning wird der automatische Pruning-Prozess in die Lage versetzt, Layer für das Pruning auszuwählen, welche sonst nicht oder nur mit dem Entfernen zusätzlicher Filter möglich wäre. Ohne die zusätzlich entfernten Filter kann eine höhere Ursprungsgenauigkeit beibehalten werden. Dies vermeidet einen erheblichen Verlust hinsichtlich der Genauigkeit und damit auch der Möglichkeiten zum prunen führen. Das Zero-Padding fügt die entfernten Kanäle als Nulltensoren wieder in den Ausgangstensor ein, und zwar so, dass die Dimensionen des Ausgangstensors identisch bleiben, aber dennoch Berechnungen eingespart werden. Mit dieser Methode erreicht man eine Beschleunigung von bis zu 21% auf CPU-basierten Plattformen und 5-6% für GPU-basierte Ausführungen eines MobileNetV2. Für die Evaluierung wurde ein Jetson Xavier AGX gewählt, da dieser drei verschiedene Ausführungsmodi bietet und somit eine solide Basis für die weitere Bewertung hinsichtlich der Anwendbarkeit darstellt. Das geprunte Netz ist vergleichbar mit dem ursprünglichen Netz mit angewandtem Depthmultiplier. Die zusätzlich benötigte Lernzeit ist jedoch deutlich geringer, als das Training eines MobileNetV2 mit Depthmultiplier von Grund auf. Darüber hinaus hängt der Effekt des Prunings hinsichtlich der Leistungsaufnahme stark von der Netzarchitektur ab. Daher reicht die resultierende Leistung von Einsparungen bis hin zu einem erhöhten Verbrauch in Abhängigkeit vom ausgeführten Netzwerk.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Abstract	3
Kurzfassung	5
1 Motivation	15
1.1 Problem Definition	16
1.2 Main Contribution	16
2 State of the art	17
2.1 Neural Networks	17
2.1.1 Fully-Connected Networks	17
2.1.2 Convolutional Neural Networks	19
2.1.3 Neural Network Training	22
2.2 Pruning	23
2.2.1 Related Work	25
2.3 Distiller	27
2.3.1 Distiller Structure	27
2.3.2 Functionalities	29
2.3.3 Pruning	32
3 Autopruner	37
3.1 Pruning Problems	37
3.1.1 Residual Pruning Problems	37
3.1.2 Groupwise Convolution Problems	41
3.1.3 MobileNet Problems	42
3.2 Solutions	44
3.2.1 Residual Solution	44
3.2.2 Groupwise Solution	45
3.2.3 MobileNet Solutions	45
3.3 Pruning algorithm	46
3.3.1 Analyse	47
3.3.2 Prune Best Layers	48

3.4	Implementation	50
3.4.1	Zero-Padding	50
3.4.2	PyTorch-Implementation	51
3.4.3	ONNX-Implementation	52
3.4.4	TRT-Implementation	53
4	Evaluation	57
4.1	Measurement Setup	57
4.2	Test Network Evaluation	58
4.2.1	Small Test Network	58
4.2.2	Big Test Network	62
4.3	Pruning Evaluation	67
4.3.1	Cifar10	67
4.3.2	ImageNet	71
4.4	Pruning together with ONNX Simplifier	72
5	Conclusion and Future Work	75
5.1	Conclusion	75
5.2	Future Work	76

Abbreviations

CAP close-accuracy points

CUDA Compute Unified Device Architecture

CNN Convolutional Neural Network

FM Feature Map

FLOP floating point operation

GEMM general matrix multiply

GPU Graphic Processing Unit

IFM Input Feature Map

MAC multiply accumulate

ONNX Open Neural Network Exchange

OFM Output Feature Map



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1	Used Hardware and Software on the Jetson Xavier AGX	58
4.2	Test Network	59
4.3	Big Test Networks	62
4.4	CPU utilisation	65
4.5	Model Metadata	67
4.6	ResNet50 Pruning Steps	68
4.7	Model Metadata	71
4.8	MobileNetV2 Power Measurements	71



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1	A basic example of a neural network.	18
2.2	A neuron in a simple fully connected neural network. The dashed ellipses are neurons from the layer before this one. The x_i are the outputs of those neurons. Redrawn from [38].	18
2.3	Different activation functions. [38, 34]	19
2.4	An exemplary convolution operation as they are often used in computer vision tasks.	20
2.5	Example of the usage of a filter kernel on an input feature map.	20
2.6	Pooling operations and how they operate. Adopted from [26, 4]. Shown is mean, maximum and minimum pooling.	21
2.7	Example of residual blocks. The skip connection part is visible. Drawn after an idea of [14].	22
2.8	A standard convolution is visible on the left and a depthwise separable convolution on the right. The depthwise separable filter kernel uses a single channel for a single channel of the input feature map.	22
2.9	Example unstructured pruning on a fully connected network.	24
2.10	Example for structured pruning within a fully connected network.	25
2.11	Convolution filter kernel to matrix mapping.	25
2.12	Training schedule with Distiller. The bold parts are new.	28
2.13	Weight Map for a MobileNetV2 trained on Cifar10.	29
2.14	Sensitivity Analysis evaluation on a already pruned MobileNet.	30
2.15	MobileNet V2 sensitivity analysis epoch 0.	31
2.16	MAC count for a MobilenetV2 trained on Cifar10.	32
2.17	Comparison between a masked and a thinned network.	33
3.1	An exemplary block within ResNet. In this example, the residual block consists of an identity part, which is the upper path, and two convolution layers on the lower path.	38
3.2	The redrawn residual block. This time showcasing the shapes of the filters and the feature maps.	38
3.3	Pruning a filter from F1.1. The resulting dependency is drawn in red and the channels which need to be removed in light blue.	39
3.4	Pruning a filter within F1.2 results in an unexpectedly high amount of dependencies to resolve. These dependencies degrade the accuracy of the original net.	40
3.5	Pruning a filter within F1 affects multiple Feature Maps (FM). This results in influencing all other depicted filters in various ways.	40

3.6	Illustration showing the working of groupwise convolutions. The used channels from the input feature map are depicted in the same colour as the filters which are using it. In the case of multiple filters using the same input features only one colour of a used filter is shown.	41
3.7	MobileNetV2 Standard Block consisting of a pointwise convolution, followed by a depthwise convolution followed by a pointwise convolution.	42
3.8	MobileNetV2 Pruning Variant 1	43
3.9	MobileNetV2 Pruning Variant 2	43
3.10	MobileNetV2 Pruning Variant 3	44
3.11	Zero-padding to get the output back into the correct shape.	45
3.12	The refined pruning recipe generation. The newly added blocks are the decision and the handle predecessor blocks.	46
3.13	The created and used pruning workflow.	47
3.14	Showing the deviation of the full set for evaluation and a small subset.	48
3.15	Filter Visualisation for a MobileNetV2 trained on Cifar10.	49
3.16	Netron network graph for the for loop implementation.	51
3.17	Netron network graph for the second implementation.	52
3.18	Netron network graph for the third implementation.	53
3.19	TensorRT compatible zero-padding network graph.	54
4.1	Jetson Xavier AGX from NVIDIA	57
4.2	Testnetwork evaluation executed on the CPU.	59
4.3	Testnetwork evaluation on the GPU. Figure b is zoomed.	60
4.4	Evaluation zero-padding onto the test network. Comparison between CUDA and TRT execution providers.	60
4.5	Plot for the testnetwork power consumption.	62
4.6	Zero-padding inference time evaluation using the CPU execution provider.	63
4.7	Zero-padding inference time evaluation using the CUDA execution provider.	63
4.8	Zero-padding inference time evaluation using the CPU execution provider.	64
4.9	Big testnetwork power consumption across different execution providers.	65
4.10	NVIDIA Nsight Systems Report	66
4.11	Pruningflow executed on a ResNet50	67
4.12	Inference measurements for DenseNet121	68
4.13	Power measurements for DenseNet121	69
4.14	Inference measurements for RegNetX	69
4.15	Power measurements for RegNetX	69
4.16	Inference measurements for SimpleDLA	70
4.17	Power measurements for SimpleDLA	70
4.18	Inference measurements for MobileNetV2	72
4.19	Power measurements for MobileNetV2	72
4.20	Example of onnx-simplifier worsening inference time.	72
4.21	Example of onnx-simplifier improving inference time.	73

Chapter 1

Motivation

Within the last years, Neural Networks have shown to be tremendously useful in a wide variety of tasks. Especially since the winning of AlexNet in the ImageNet challenge [22] the hype around Neural Networks and especially Deep Neural Networks keeps on increasing. There already exist solutions to problems with neural networks not only to image classification but also to object detection, natural language processing [43] and strives to be the main force for self-driving cars [40]. Most of these networks run on servers with high processing power, therefore making them only usable in artificial environments with a powerful uplink connection. But to fully harness the functionality and benefits of these methods, it would be necessary to have them usable in all places. Such devices might be but are not limited to, mobile phones, cars, cameras. These devices are situated on the so-called edge with either a lack of processing power, no sustainable uplink connection, battery dependence or combinations of the aforementioned.

One of the main problems presents within the deployment of neural networks in a wider range of environments is the target platform. Ideally, neural networks would be perfectly accurate and for example, within a camera, only the detections would need to be transmitted. This would lead to tremendous bandwidth savings and therefore also installing savings. Another example would be a deployed network within a car to detect oncoming traffic or obstacles along the way. Since a car is a system running on a battery and without a stable uplink connection the network would need to run solely within the car's ecosystem. The power consumption for GPUs goes up to 225W. Running such a system or even multiple such systems within the car would be done at the cost of the battery and might not even be feasible for the car.

Many of such devices, lack either the processing power or are limited by the amount of available energy. A possible approach is to tailor the neural network to the end devices, such as creating smaller networks. These smaller networks are often less accurate and might be unfit for the task concerning the achieved accuracy. Another viable approach is to use the big trained network and remove parts of it that are not necessary or only contribute marginally to the accuracy of the network. This procedure is also labelled pruning and is by far no new technology. First uses of pruning [7, 13] have already been used in 1990 and 1993. Pruning has been done with heuristics by humans [27] and are also starting to gain traction in automated ways [16]. But pruning also has limits with only being able to remove computations from the most common layers and even therewith restrictions regarding surrounding layers.

1.1 Problem Definition

This work strives to reduce the energy consumption of such networks by either reducing the power consumption of the network or by decreasing the inference time. To achieve this an automated pruning approach is to be implemented. To push the pruneability of networks even farther a pruning implementation called zero-padding is to be implemented and evaluated. This pruning approach is then tested and utilised on a Jetson Xavier AGX from NVIDIA. These embedded computing platforms come with a GPU and are optimised for the execution of neural networks. They offer different power targets and also feature a specialised optimiser for NVIDIA hardware. The implemented method shall also be compared using these. For the implementation Distiller [49] is used as a basis. Building upon this the pruning flow is expanded to automate the approach and zero-padding is introduced to this implementation.

The problem with an automated way is in finding the right positions to prune within both the network and the layer. As pruning too much from a network renders it unusable. These networks then do not regain their accuracy even when retraining for longer periods of time.

1.2 Main Contribution

The main contributions of this work are the following:

- Implementing and evaluating a pruning method to deal with some of the major problems pruning has regarding its applicability.
- Evaluation of the usage of pruning on an embedded platform, which within this thesis is the Jetson Xavier AGX.
- Expanding Distiller [49], a pruning framework for neural network research.

Chapter 2

State of the art

Neural networks are widely used and adopted in a wide variety of tasks. These tasks range from image classification [22], object detection [2, 48] to natural language processing [37, 10]. As the power of neural networks grow further a growing need to use these networks everywhere emerged. But with this need, further problems are also emerging. As it was the the normal procedure to have these networks running on big servers with multiple Graphic Processing Units (GPUs) and an immense calculation power, the desire to use these everywhere meant that these networks also need to be usable on smaller and less powerful devices. This porting to a different platform also should happen ideally no loss regarding accuracy.

2.1 Neural Networks

Neural networks are tailored to fit a specific problem. This is done by training the network for the given task. Furthermore, neural networks outperform most conventional approaches to tasks like image recognition and object detection. These networks are inspired by the human brain and consist of nodes that are interconnected to each other. Values get propagated through these nodes and interpreted at the output. Most networks consist of an input layer, an output layer and multiple hidden layers in between these. Values are propagated through the hidden layers to the output layer and then interpreted. Through a training process, the network gets trained to act on a given input respectively. A basic representation for a simple neural network is depicted in Figure 2.1. The input data is presented at the input layer with the orange background. This data gets propagated through multiple hidden layers, which are three in this case and coloured red. At the output layer, the evaluated input is then presented for further evaluation.

Following now is a short walkthrough for the most common networks and network operations.

2.1.1 Fully-Connected Networks

A basic example for a neural network can be seen in Figure 2.1. This is a fully connected network, where each node within a layer is connected with all of the following nodes in the next layer. Therefore the name fully connected. This can be considered as the most basic structure.

Each of these nodes or neurons has j inputs and one output. These j inputs each get multiplied with a corresponding

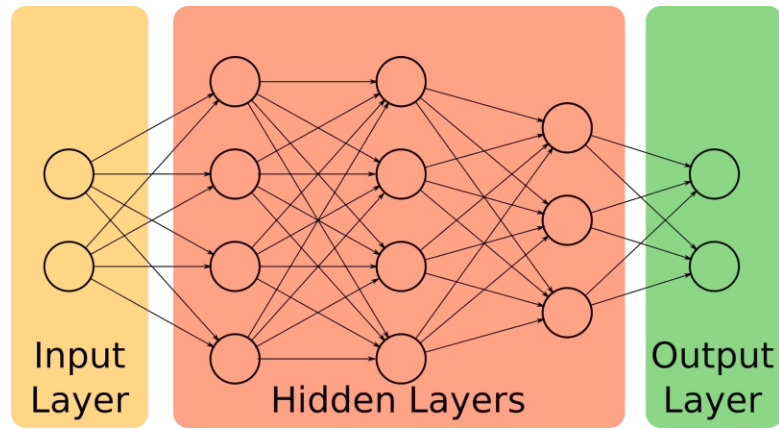


Figure 2.1: A basic example of a neural network.

factor, a so-called weight. These products are then summed together and this sum is put into an activation function. This single value is then the input for other nodes. Each edge connecting the nodes has a weight corresponding to it. Mathematically, this equation for a single neuron i within a layer comes together to

$$y_i = f\left(\sum_j x_j w_j\right) \quad (2.1)$$

with f denoting the used activation function, x_j is the j -th input to the neuron and w_j describing the weight which gets multiplied with the input. The basic makeup of such a unit also called a neuron, can be seen in Figure 2.2. A simple fully connected neural network consists of multiple such neurons, ordered in a column-wise manner. Due to this column positioning, all outputs of a column i can be put into a vector and all weights $w_{i,j}$ can be put into a matrix. The weights now have two indices, due to being different for each neuron connection. With this adaption, the equation of a layer output in a fully connected network becomes $y_j = f(\sum_i W_{ji} \times x_i)$ with y_j describing the output for the neuron j , f being a non-linear activation function, W is the weight matrix and x_i being the inputs to this layer respectively outputs of the neurons a layer before.

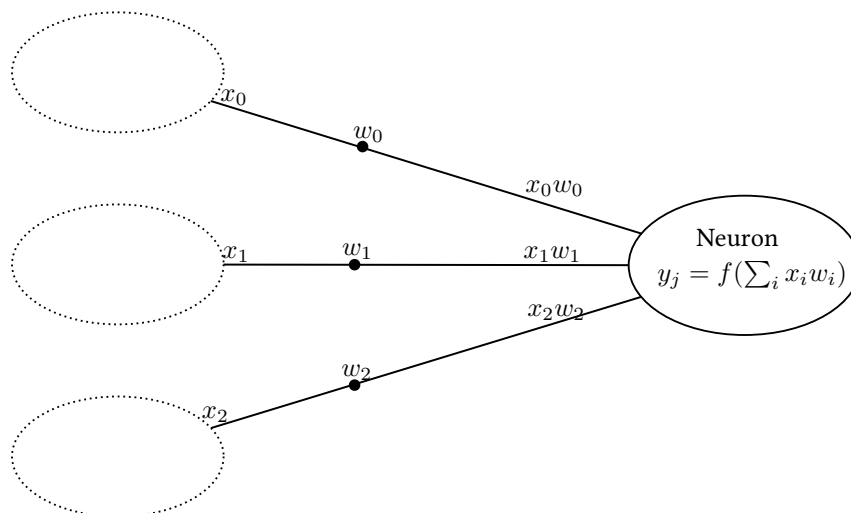


Figure 2.2: A neuron in a simple fully connected neural network. The dashed ellipses are neurons from the layer before this one. The x_i are the outputs of those neurons. Redrawn from [38].

Bias

Bias values are added at each neuron to the output before feeding the sum into an activation function which gets further forwarded to the next layer $y_j = f(\sum_i W_{ji} \times x_i + b)$. This is the same equation for a neuron as in the section above but with an added bias value b . The bias itself allows the shifting of the output along the x-axis. Thus allowing a better fitting to the given data.

Activation Function

The activation function is a non-linear function that takes the summed up inputs (+ the bias) as explained in Section 2.1.1 and shown in Equation 2.1. The reason for non-linearity is to be able to model non-linear behaviour. If a linear function would be used, the whole network would be expandable into a simple equation. Some of the most commonly used activation functions are listed below, together with plots showing their behaviour.

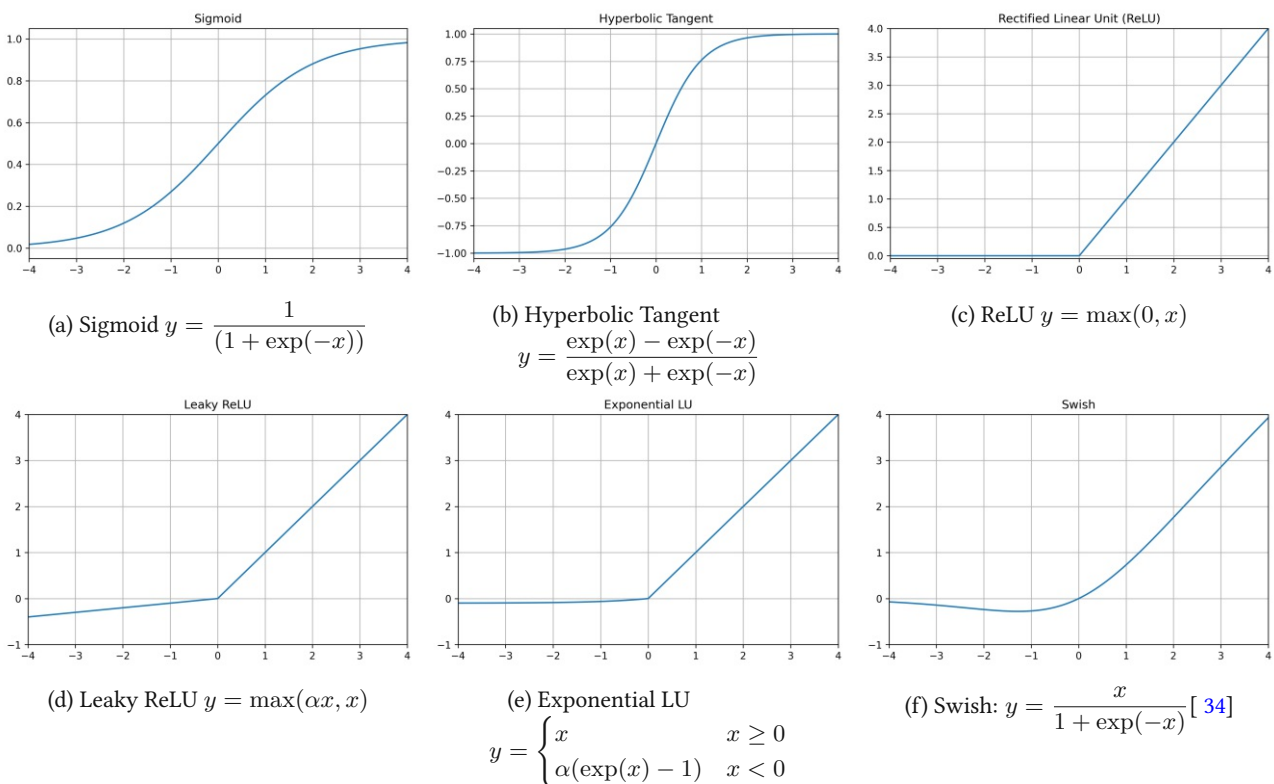


Figure 2.3: Different activation functions. [38, 34]

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks are made up of mostly convolutional layers. These layers are at the heart of tasks handling images or video data. These differ from the explained fully-connected networks in the sense, that they take filter kernels and perform a convolution operation on a given input feature map as shown in Figure 2.4. Feature map in this context refers to the given input to the layer, being an input image in the case of the first layer. The input feature map then would have the following dimensions: 3 channels for the three colour channels red, green and blue and the height and width of the input image. The input convolution layer then holds n filter kernels. Each of these n kernels holds $c \times k_x \times k_y$ matrix with c referring to the channel count of the input feature map, k_x, k_y to the width and height of the convolution kernel. Each of these filter kernels is moved over the whole input feature map. At each position, each value within the kernel is multiplied by the value within the feature map overlapping the kernel and summed up.

This value is put into an activation function and output as a single value on the output position. This is done with each filter kernel, such that each kernel produces a single 2D output feature map. These 2D maps, one for each kernel, are then stacked and therefore create a 3D output feature map.

Visually the way a convolution works is depicted within Figure 2.4. The filter kernel gets positioned onto the input feature map and moved across it. On each position, all the input values within the filter kernel get multiplied with the kernel and summed up. This way an output channel gets calculated. This is done for each filter kernel available. This way n channels are created whereas n is equal to the number of kernels. These n channels are stacked onto each other and passed forward to the next layer. The stacking can be seen within Figure 2.5. Within this figure, each coloured filter kernel generates an output channel with the corresponding colour.

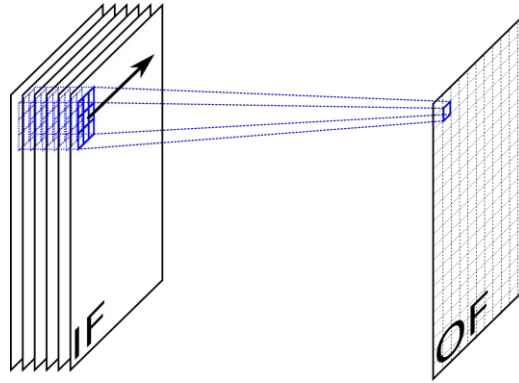


Figure 2.4: An exemplary convolution operation as they are often used in computer vision tasks.

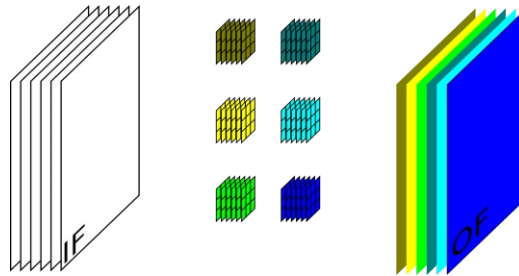


Figure 2.5: Example of the usage of a filter kernel on an input feature map.

In a mathematical sense, a convolution layer looks like the following [38]:

$$O[u][x][y] = B[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} I[k][Ux + i][Uy + j] \times W[k][i][j] \quad (2.2)$$

with O being the output feature map, I the input image respectively the input feature map, W describing the filters and B as the biases. W is a 4D matrix, due to describing multiple 3D filter kernels which are used as shown in Figure 2.5. The kernels W also come in different used sizes, such as 5×5 , 3×3 and 1×1 with the latter two being the most popular in networks such as DenseNet [20] and the MobileNet [18, 36, 17] variants.

The computational effort (within this work denoted as E) or also called the computational cost for a single convolutional layer is a possible metric to evaluate the effort it takes to compute the convolution. So for a single convolution layer this comes down to:

$$E = h_i \cdot w_i \cdot d_i \cdot d_j \cdot k_x \cdot k_y \quad (2.3)$$

with h_i , w_i and d_i being the height, width and depth (the number of channels) of the input feature map. d_j describes the number of filter kernels that are used within the given layer and k describes the kernel size. If $k_x = k_y$ then the kernel is symmetric.

Pooling

Pooling is a concept, which reduces the dimension of the input data by introducing pooling layers. Pooling is usually applied to each channel separately and helps the network to become more robust to slight shifts in variations. A pooling operation operates only on a small receptive field similar to convolutional kernels. The key difference is that pooling operates on non-overlapping blocks. Within Figure 2.6, three different pooling operations are shown. The leftmost shows a mean pooling operation. Mean pooling propagates the mean value of each input block. A similar approach is used with maximum pooling, which is taking only the maximum value and minimum pooling taking only the minimum value within a block.



Figure 2.6: Pooling operations and how they operate. Adopted from [26, 4]. Shown is mean, maximum and minimum pooling.

Residual Blocks

Residual networks were first introduced by [14] to tackle the problems of saturating accuracy values when stacking more layers. A residual network consists of residual blocks which contain the normal convolutional layers in case of an imaging problem. A residual block is not a layer type per se, but since these are commonly used in modern network architectures, they get addressed separately. A residual block is a means to stop the increasing training error when stacking more and more layers within a network. Within the network, the overall goal is to learn $\mathfrak{H}(x)$ and the residual of this is $\mathfrak{R}(x) = \mathfrak{H}(x) - x$ which equals after rearranging $\mathfrak{H}(x) = \mathfrak{R}(x) + x$ which then equals the residual. Due to the identity part x the layers are learning the residual, therefore the name. This holds further benefits since it is easier to learn the residual of the output and input and the network can now also learn the identity function. Also, the problems concerning vanishing gradients become less dominant due to the skip connections which propagate bigger gradients farther into the network.

Depthwise Separable Convolution

Depthwise separable convolutions [11, 36, 3] are a special case of normal convolutions. In this case, the convolution is split up into groups. Group in this context corresponds to a smaller kernel that uses a smaller portion of the input feature map. This is done to reduce the required parameters and computations. For a depth-wise separable convolution,

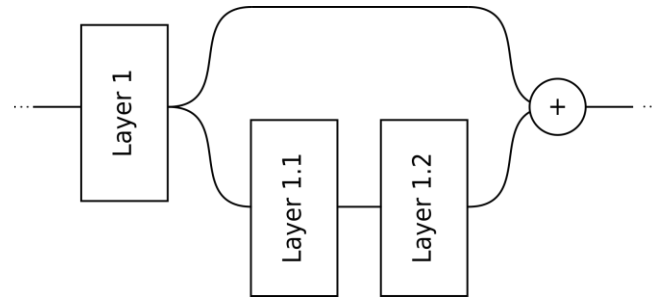


Figure 2.7: Example of residual blocks. The skip connection part is visible. Drawn after an idea of [14].

this group parameter is maximised. This means that each channel from the input feature map is convolved once with a single channel filter kernel. Within Figure 2.8b this is shown with corresponding colours.

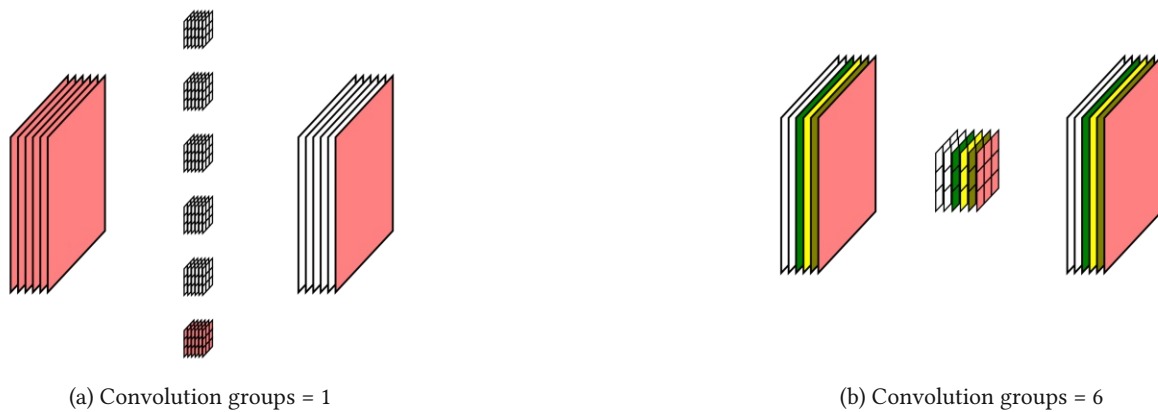


Figure 2.8: A standard convolution is visible on the left and a depthwise separable convolution on the right. The depthwise separable filter kernel uses a single channel for a single channel of the input feature map.

2.1.3 Neural Network Training

The weights and also the bias values are often referred to in Section 2.1.1, but it is not explained how these get determined. The inputs are forwarded through the network. The received output is then compared to the expected output via an error or loss function. This loss function evaluates the expected output versus the received output. Based on this deviation from the results, the weights are readjusted to reach the desired output more reliably. This step is repeated for each weight or parameter within the network. This process is called backpropagation. After all the required values are calculated, the change is applied corresponding to the learning rate. The learning rate determines, how severely the backpropagated values influence the recalculated weights. To avoid severely changing all weights all the time, learning rate schedulers are in use to decrease the learning rate over time. The time is measured in so-called epochs. One epoch equals one iteration of a whole training data set. Simple schedulers decrease the learning rate just based on the current epoch. More sophisticated also take into account if the accuracy gain has stagnated for some time and then decrease the learning rate further.

After a network is fully trained, by reaching a given accuracy target or requiring a given amount of time, the weights get fixed. In this state, the weights are not going to be changed anymore and the network is only used in forwarding direction to fulfil the trained task. These tasks include classifying images, detecting objects or generating text.

2.2 Pruning

Pruning is the process of removing operations from the network. This is done to save up on calculations and thus improving the processing speed. Furthermore, removing superfluous calculations also creates the advantage of omitting memory transactions. Due to removing not needed calculations, the corresponding data does not have to be fetched either. Pruning can be divided into two big groups. One of those groups being unstructured pruning, where values are pruned independently on their position and only based on some external metric. An example of the choice of a metric could be the absolute value of the weight. If the weight is small compared to the others or lies below a certain percentage threshold, the parameter gets removed. The other big group is structured pruning. Within structured pruning the position of the data within the network and its structure is included in the decision making. This results in a more coarse view on the weight matrix due to considering weights only in bigger groups such as a filter kernel.

Pruning is not a new concept as the idea was first introduced with Optimal Brain Damage [7] in 1990. With Optimal Brain Damage, weights are pruned by using a saliency measure. Later Optimal Brain Surgeon [13] is proposed, which uses second-order derivative information and calculates the inverse Hessian to get structural information from the network. This is done to prevent the pruning of small but important weights. These two methods were applied to fully connected networks such as shown in Figure 2.1. The concept of pruning weights to gain speed and create smaller networks was further progressed and also applied to CNNs.

Some of the main questions that arise are:

1. Is this applicable to all kind of networks?
2. Is this only possible for image classifiers?
3. How to select the weights to remove?

Unstructured Pruning

Unstructured pruning is the process of removing single weights without considering the position of the data points within the network. For example, removing all weights over the whole network which are smaller than a given value. Visualised in a fully connected network this would just remove values everywhere and affect the edges between the nodes.

For a convolution, this would set single values within a filter kernel to 0. With this approach higher pruning rates are achievable than with structured pruning. Despite this the benefits of this approach are sparse. Consider for example the basic fully connected network as depicted in Figure 2.1. If some of the edges are removed the network would then for example look like Figure 2.9a. Within this depiction, the basic network with selected edges marked in red is shown. These edges are selected based on some metric criteria. When these are removed the network looks as shown in Figure 2.9b. Theoretically, this would result in lesser calculations required, but the benefit of pruning with regards to inference speedup becomes negligible if one takes into consideration how these are processed. Special hardware and algorithms would be necessary to gain any kind of improvement. This is due to the need to constantly checking every value for zero and skipping said zero entries. Removing a value completely causes a different problem. The newly created mathematical structure is not usable by libraries like PyTorch, which is built on the premise of using valid matrices. The root cause of this design decision is the applicability of general matrix multiply (GEMM) [5] which is

optimised in processors and even more so in GPUs.

Even though unstructured pruning reaches greater values in sparsity [28, 47], the execution of such pruned networks is hard. Due to resulting in sparse matrices special hardware or algorithms would be needed to benefit. In the case of not switching to a sparse representation, simple removal of the weights is sometimes not possible due to destroying the GEMM form. A potential advantage might be a reduction in memory usage for storing when the network is compressed.

The benefits of unstructured pruning are sparse and only beneficial when reaching high pruning rates. For example [47] shows that using a sparse representation offers a speedup only at higher pruning rates in the range from 50% to 80% depending on the type of the pruned layer. This is not universally true as they also show that this is also highly dependent on the parallelism available in the underlying hardware. But for the general use case of using GPUs, the use of unstructured pruning is even discouraged by some [28].

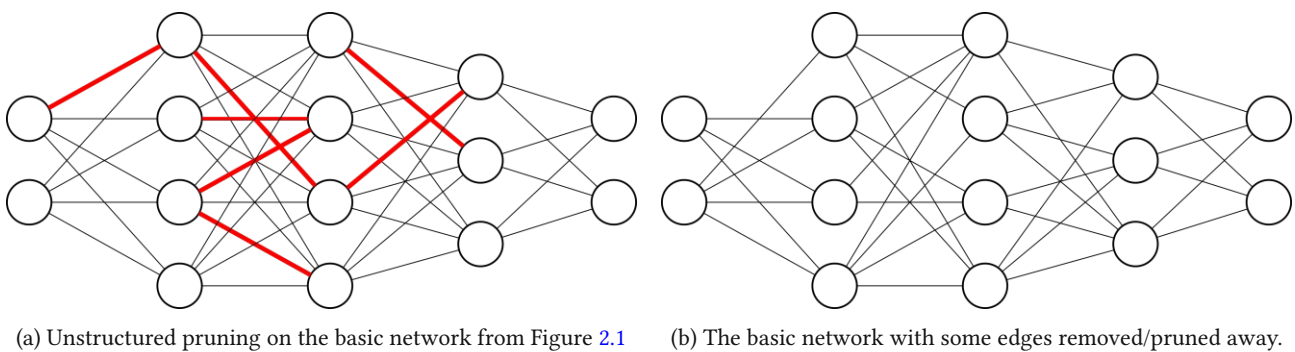


Figure 2.9: Example unstructured pruning on a fully connected network.

Structured Pruning

Structured pruning, as opposed to unstructured pruning, enables a more coarse pruning. This results in always removing bigger parts of the network, but these are removable from the computation. Unlike unstructured pruning, which would break the GEMM format in many cases, structured pruning will leave the matrix intact. To visualise, structured pruning would remove not the edges within a fully connected network but whole nodes. Exemplary this is visualised within Figure 2.10a. The selected neuron and all the corresponding connections to the neuron are marked in red. Figure 2.10b shows the neural network with the two highlighted nodes removed. This now demonstrates the effect the removed node has on the previous and the next layer. All the weights from the incoming edges from the previous layer can be removed. The bias value from the node itself also is removed and all outgoing edges and their corresponding weights from the node to the next layer also become unnecessary.

Convolutional networks are playing a big part in the task of image classification. The convolutional part is based on the classic computer vision problems where edge detection and so on happens via filters that get moved over the image. The input to such a network is most commonly a three-channel image. One channel for each colour: red, green and blue. Over each of these channels, a convolutional operation is applied.

The convolution operation is mapped into a GEMM. Such a mapping is depicted within Figure 2.11. Each row within this matrix stems from a filter kernel, whereas each column corresponds to a channel within a filter kernel. Structured pruning does remove only whole rows, columns or groups of columns. These correspond to filter, filter shape and

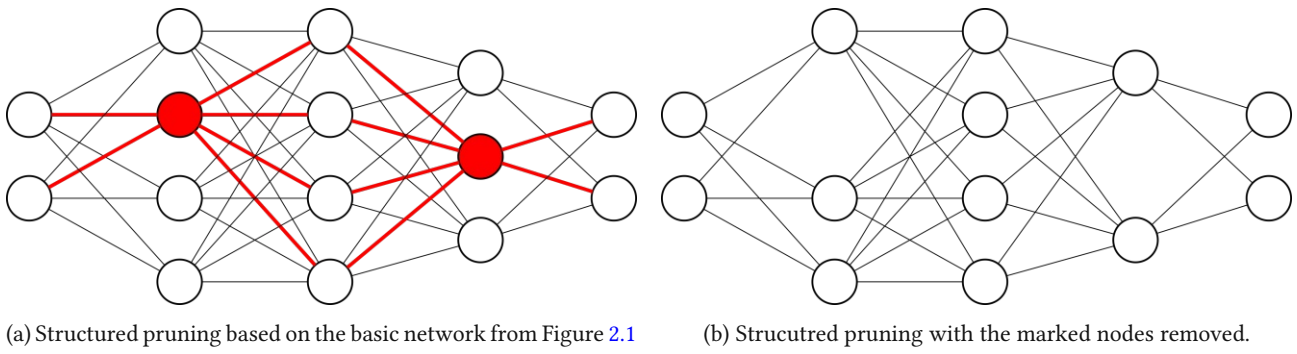


Figure 2.10: Example for structured pruning within a fully connected network.

channel pruning. The figure also shows the cases for filter and channel pruning. Filter pruning for example would remove the yellow marked row. Channel pruning on the other hand would remove the whole red marked area.

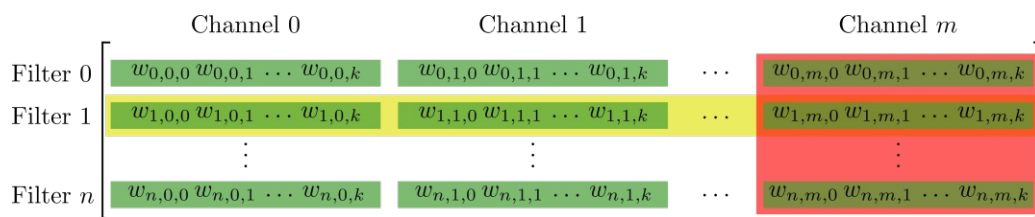


Figure 2.11: Convolution filter kernel to matrix mapping.

2.2.1 Related Work

Pruning has shown in multiple works [16, 27, 31, 44, 24] to being able to remove big parts of a network regarding either model size, multiply accumulates (MACs) or both. One can influence what is optimised by targeting different parts of the network for pruning. Convolutional layers are more MAC heavy, whilst fully connected layers contribute to the model size due to having more parameters than fully connected layers.

The removal of parts of the network to save on computations is not a new concept. It was first introduced by [23] and later on refined by [13]. The recent rise in machine learning and with this the rise of required computational power gave a new surge to finding more efficient networks [18, 36, 30]. MobileNets [36] are prominent networks that offer different configurations of the same network due to the use of depth multipliers. Also, other model compression techniques such as quantization reduces the required computations [28].

Pruning can be separated into multiple groups: unstructured and structured pruning. The main difference is that with unstructured pruning, only single points in the network get set to zero or removed. Due to mapping most common operations into a matrix multiplication format, this approach may result in a high sparsity, but only with a little speedup. It is possible to utilise this approach by using sparse matrices, but even then only with high pruning rates of up to 80% for convolution layers, a speedup is reached [47]. Thus they show, that pruning with the target hardware in mind yields better results.

To deal with the problems from unstructured pruning structured pruning uses a similar approach but looks at more coarse units within the matrix. With the general matrix multiply format in mind, only rows, columns or groups of columns are removed, which map to channels, partials of the filter kernels or whole filter kernels respectively. Due to this removal, a speedup better than by unstructured pruning is reached.

The natural approach in pruning is to remove weights that are close to zero due to the assumption that these weights do not hold any viable value to the calculation. The correctness of this assumption is shown in multiple works [12, 27, 15].

[12] uses L1 and L2 regularisation to determine which parts are to be removed within the convolution and the fully connected parts of AlexNet. They reduce the original model by a factor of 9 and even increase the top1 and top5 accuracy. [27] shows the applicability of this ranking approach on multiple convolutional networks. Furthermore, they compare the usage of L1 vs. L2 norm and conclude, that there are no notable differences in the applicability regarding the influence on the results. Within their work, they reduce the floating point operations (FLOPs) for a ResNet-110 by 38% and the parameters by 32%. They also show that a high reduction in FLOPs does not necessarily correlate with a high reduction in parameters. A ResNet-110 was reduced by 16% regarding the FLOPs, but only 2.3% of the parameters were removed, showing a high removal within the convolutional part of the network. [15] uses L2 norm in a soft filter pruning approach. Within their approach, they enable pruned filters to stay in the network and be updated by a backward pass. Thus they enable removal of up to 41% FLOPs within ResNet-110 with an accuracy loss of 3.1%.

Recent works on pruning are [41, 16, 31]. [41] utilises a similar approach like this work and builds upon [24]. They utilise a combined pruning approach removing up to 93% of the nodes within the network which is trained on the Cifar10 [21] classification task. They combine within their tool a structured and unstructured approach and the fact that they prune before the training sets them apart from other work. [31] takes the structured pruning approach a step further, based on the assumption that filter shape is also important and introduce filter stripe pruning. This even more fine-grained pruning allows them to remove 50% of the FLOPs in a ResNet18 trained on ImageNet while reducing the accuracy by 0.2%. [16] utilises pruning with the the help of a reinforcement learning agent to determine where and how much to prune. With this approach, they reduce 50% of the flops for a ResNet-56 trained on Cifar10 with only a marginal accuracy loss of 0.9% from the original model.

I use the tool Distiller [49] in pruning, as it is currently one of few frameworks to support thinning. At least for networks out-of-the-box and without changing the network definition and creating conversion methods between networks with different dimensions. Problems arise with Distiller when faced with layers that hold complex dependencies, mainly represented thru residual blocks. The main problem and how this was circumvented is described in Section 3.1.

Evaluating what to prune is done in multiple ways. Mainly differentiable in using measurement-based approaches or heuristic methods. Measurement-based approaches record for example the activation ratio of neurons and remove those with zero or a low ratio [19]. Other measurement-based approaches use sensitivity analysis [41, 27] to obtain data on how strongly each layer influences the total accuracy. This approach is even further improved by [16] which uses a measurement-based method together with a reinforcement learning agent to generate a pruning plan for a given network.

Also, a good approach to result at smaller networks is the assumption, that a better version of the network, exists as a subnetwork. This hypothesis is called the Lottery Ticket Hypothesis. It states, that within a randomly initialised network, all nodes exist with a perfect weight upon initialisation and therefore only pruning is necessary to end up at the better smaller network [29].

One of the few which also evaluate the pruning effectiveness not only in theory and theoretical values such as saved MACs is [9]. Elkerdawy et al. also evaluate their chosen networks with regards to savings at the inference together

with remained accuracy.

Another approach with similarities to pruning are networks that get trained and consist of subnetworks. Depending on the constraints the network gets shrunk or only partially evaluated. An example of this are slimmable networks [46] which use a slimming factor to evaluate each only up to a given percentage.

2.3 Distiller

Distiller is a framework created for the exploration of optimisation methods on neural networks. It builds the base for the further pruning algorithm. Therefore a base explanation is given to the original Distiller and the features present within it. For the explanation, a base knowledge in python is assumed due to Distiller being written in Python. It was created for testing the effectiveness of pruning on neural networks and exploration of the pruning space. With time further optimisation methods and algorithms were added. Some of these are:

1. Early-Exit
2. Quantization
3. Lottery-Ticket
4. Knowledge Distillation
5. Thinning

Distiller was created by the company Nervana Systems. They also offered an open-source deep learning framework. The company then was acquired by Intel in August 2016 and since then continues work on Distiller. [Nervana Wikipedia¹](#)

Distiller itself makes use of the PyTorch Neural Network Framework. Namely the version 1.3.1. Their choice fell on PyTorch due to the added flexibility with the training process and the ability to change how layers behave during training. More to why this is important in Section 2.3.1 in which the functioning of Distiller and the changes it makes to the standard training loop are explained in more detail. Furthermore, it is designed as an interactive framework that uses configuration files to determine beforehand what should happen to the loaded network.

2.3.1 Distiller Structure

To work as illustrated in Figure 2.12 Distiller makes only minor changes to the usual workflow. These changes encompass adding a so-called `compression_scheduler`. This compression scheduler consists of multiple policies such as

- a pruning policy,
- a regularization policy,
- a learning rate policy,
- and others ...

¹ https://en.wikipedia.org/wiki/Nervana_Systems

Each of these policies holds a task such as calculating the new learning rate or calculating which filters are to mask. The policy has then defined methods for each point of time which are defined within the compression schedule from Distiller.

```

for e in epoch:
    compression_scheduler.on_epoch_begin(e)
    train(e)
    validate()
    save_checkpoint()
    compression_scheduler.on_epoch_end(e)

train(epoch):
    for step in epoch:
        compression_scheduler.on_minibatch_begin(epoch)
        output = model(input)
        loss = criterion(output, target)
        compression_scheduler.before_backward_pass(epoch)
        loss.backward()
        compression_scheduler.before_parameter_optimization(epoch)
        optimizer.step()
        compression_scheduler.on_minibatch_end(epoch)

```

Figure 2.12: Training schedule with Distiller. The bold parts are new.

The `compression_scheduler` object then holds all these different policies and furthermore defines the times at which these policies shall get executed. Therefore the following points within the training loop are defined:

on_epoch_begin gets executed at the beginning of each epoch. This point in the execution is used for coarse masking of filters and channels. After the masking, the calculated values act like they were not there.

on_minibatch_begin is the beginning of a new data batch for training. This step is also used for pruning during training when a filter value moves below thresholds due to the weight adjustment.

before_backward_pass is the point at which a forward pass is finished and the next operation is the calculation of the optimiser. This calculation is also called the training pass. This point in time is used for regularization policies to calculate the policy loss.

before_parameter_optimization is the point after which the training pass has completed the backwards-pass and the optimiser is about to update the weights.

on_minibatch_end after a batch of data went through the network. Operations that happen here are for example the quantization. The quantization offered by Distiller is only a fake quantization with floating-point values but within the quantized ranges.

on_epoch_end at the end of each epoch. An operation that falls into this area is for example the learning rate scheduler. The learning rate scheduler adjusts the learning rate according to some meta information and depending on which learning rate scheduler is used.

The Distiller pruning process is also a rather complicated manner. Pruning does not happen by directly removing the pruned weights, filters or nodes. Rather the `compression_scheduler` holds an item called the `zeros_mask_dict`. Within this python dictionary, all the ‘pruned’ weights are marked. This marking is done with another tensor of the same shape. This tensor is a one-tensor and all weights which should be masked are set to zero within this additional tensor which is held by the dictionary. On the `on_minibatch_begin` step an element-wise in place multiplication between the real weights and the masking tensor from the dictionary is executed. These marked items are thus set to zero, to simulate

the removal of the corresponding nodes. This might seem like a complicated way to do it, but this is a drawback that comes from using PyTorch. PyTorch saves all models in the format of a class description and the corresponding weights. Therefore it is not possible to load a model only from the weights, as these saved weights only hold the raw data.

2.3.2 Functionalities

Distiller already presents quite an impressive amount of possibilities to evaluate a given network. As an example, some data relevant to know in the case of pruning would be which layers should be pruned. A possible approach is then to use for example weight maps to determine where much data transfer happens.

Weight Maps

Weight maps show the number of parameters within a layer. These consist of 3 participants. The first is the Input Feature Map (IFM) volume. The input feature map volume shows how many parameters are within the feature map that gets fed into the layer. The meaning of the Output Feature Map (OFM) is the same, but with the feature map after the calculation. The weights volume show how many parameters are used within the calculation for the filter kernels or the fully connected part.

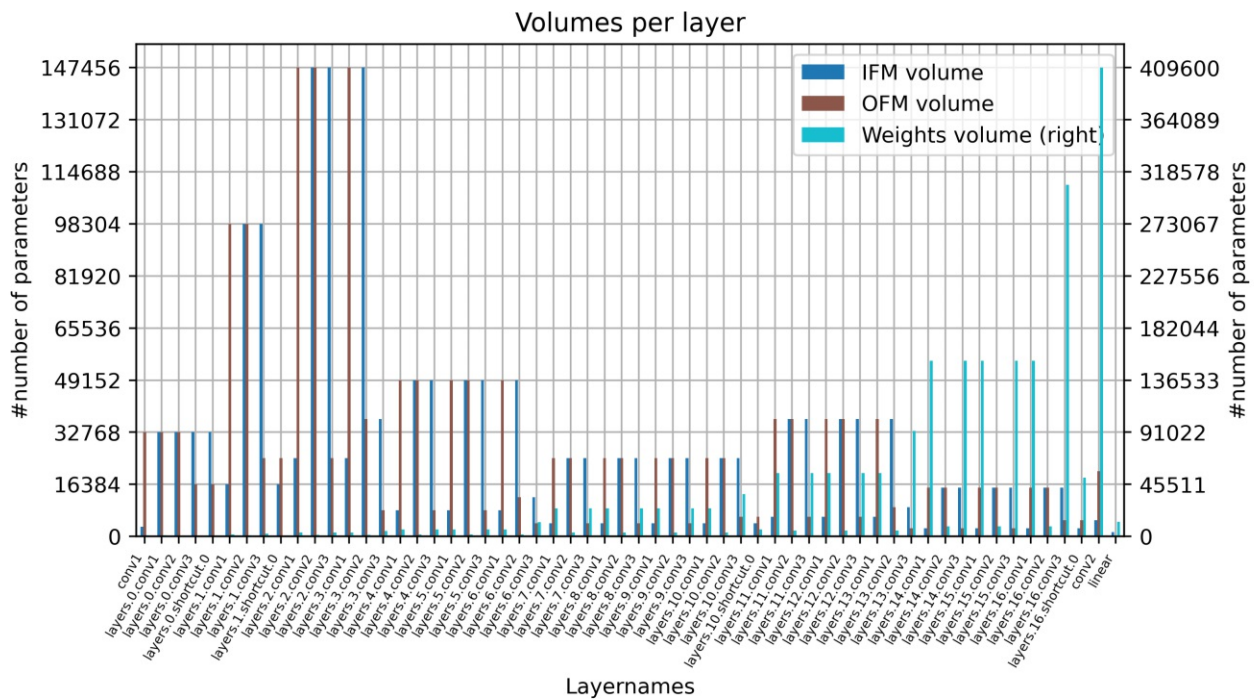


Figure 2.13: Weight Map for a MobileNetV2 trained on Cifar10.

Layers with a high amount of parameters might be susceptible to memory bottlenecks and are therefore worth taking note of pruning them. Furthermore, it is to mention, that the fully connected part (denoted as linear within Figure 2.13) usually contributes very strongly to the weight map. Due to this network being trained on Cifar10 it only has a small subset of a classifier that normally would be used.

For example, the convolution layer with most parameters is conv2 and has the following shape: $1280 \times 320 \times 1 \times 1$ and the fully connected layer for Cifar10, therefore, has the following shape: 1280×10 . A classifier for ImageNet with the same convolution layer in front would have the shape 1280×1000 . Thus resulting in factor 100 more parameters for

this single layer. This weight map would then be $\approx 200\%$ bigger than the weight map for conv2. This is to show, that the classifier does contribute more than it seems in this work. And this is only considering a single fully connected layer for the classifier. For example within MobileNets three fully connected layers are used for the classification.

Sensitivity Analysis

The sensitivity analysis is one of the main parts used in this work. The principle is simple: Each layer gets pruned separately and stepwise and then evaluated to show the influence this layer has on the total accuracy. This procedure enables pinpointing layers which contribute only little to the total network accuracy and can therefore be removed from the network.

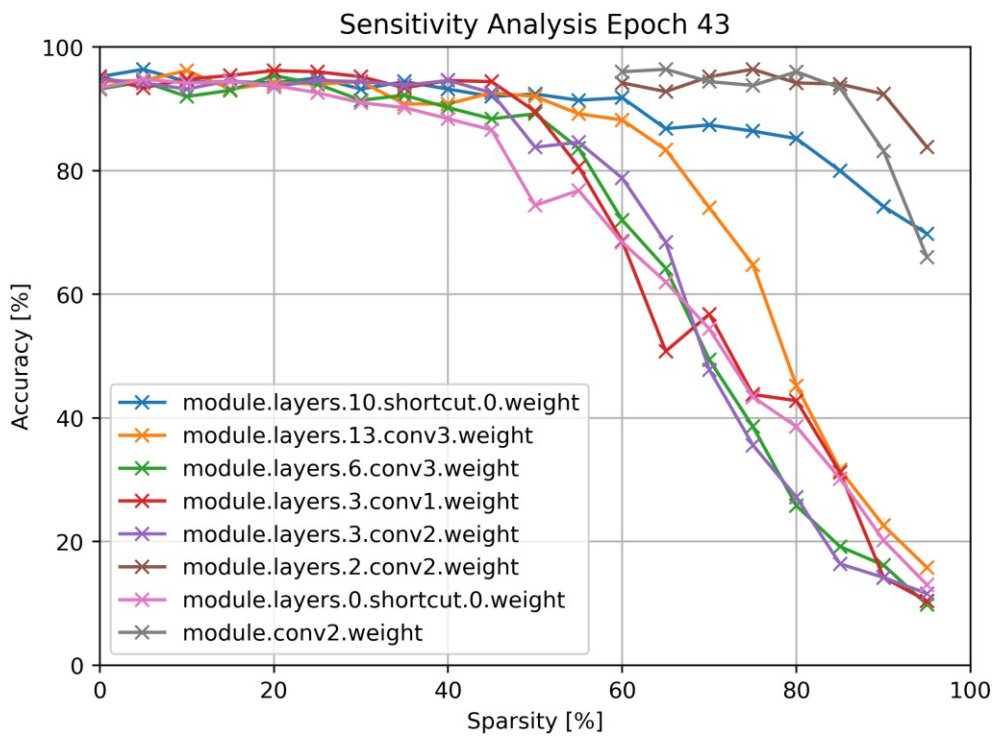


Figure 2.14: Sensitivity Analysis evaluation on a already pruned MobileNet.

Shown in Figure 2.14 is an evaluation of an already pruned MobileNetV2 in epoch 43. This Figure shows the 8 best performing layer out of the 57 layers within this network. The y-axis shows the current network accuracy on the classification task being the top1 accuracy. The x-axis shows the sparsity within a layer. Sparsity in this case is defined as the number of elements that are zero in proportion to the total number of elements within this layer. As in this work, the main focus is on filter pruning sparsity is therefore formulated as follows: The layer 13 convolution 3 has the shape $(160 \times 576 \times 1 \times 1)$. This corresponds to 160 filter kernels, of which each has 576 channels consisting of an 1×1 convolution kernel. A filter is zero when each parameter within 576 channels is zero. Therefore 8 filter kernels which are zero in all 576 values (channels times convolution kernel dimension) stand for 5% sparsity within this layer. The analysis used in this work does 5% steps and stops at 95%. This is due to a whole pruned layer is equivalent to disconnecting the network. This holds not true for networks with residual blocks due to residual connections and point-wise operations.

Within each step, the network gets evaluated to know the impact on the total accuracy. As can be seen in Figure 2.14 these give a representation of which layers are pruneworthy and which might not. To put epoch 43 in contrast with

an unpruned network Figure 2.15 can be seen. All of the 8 shown layers have close to no influence on the total accuracy.

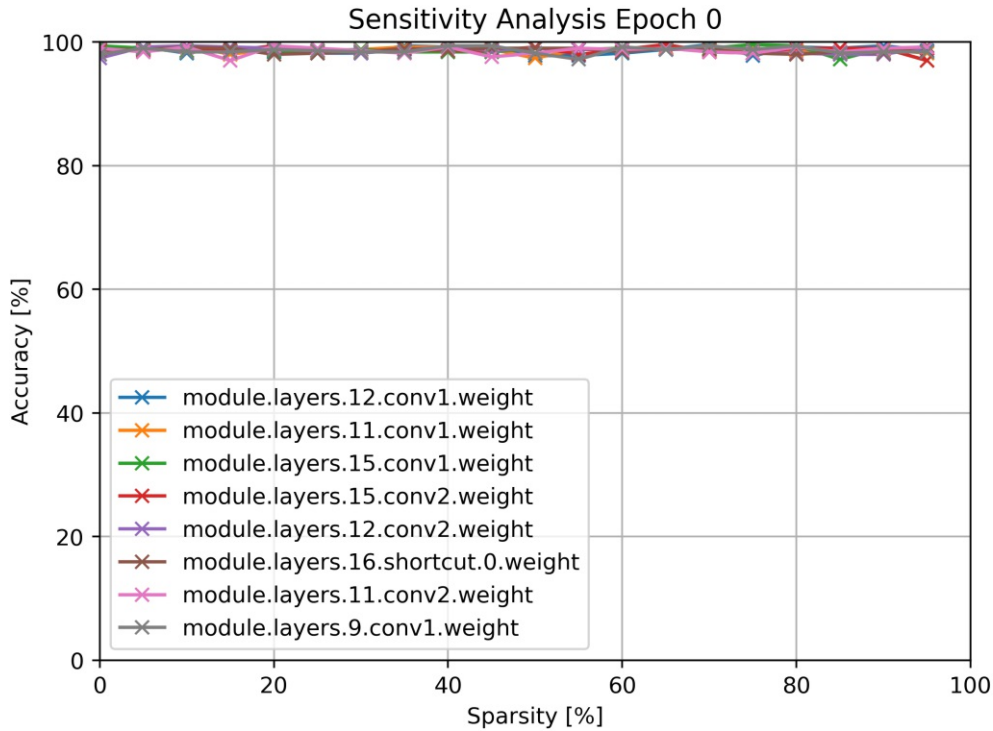


Figure 2.15: MobileNet V2 sensitivity analysis epoch 0.

MAC evaluation

Depending on the model architecture, Distiller calculates a MAC map which shows how many MACs happen within a layer. This is useful information to determine layers that are significant in terms of computational complexity. Within these layers, redundancy is common due to similar filters. Such a MAC mapping can be seen within Figure 2.16. From this information computation heavy layers can be derived and pruned accordingly for example in computational weaker platforms.

When comparing Figure 2.13 with Figure 2.16 a slight consistency is visible. But the assumption, that many weights equal many multiply accumulate is not correct. As an example serve the layer groups 11, 12 and 13. Within these three groups, the accumulated weights are small, but they rank third in terms of MACs. The reason for this is the usage of the point-wise and group-wise convolutions within MobileNet.

Thinning

A further and most important functionality is the ability to thin networks. Thinning refers in this case to the removal. More on this exact manner and how the thinning works are within Section 2.3.3. To show the difference between a thinned and a masked network see Figure 2.17. Within Figure 2.17a the masked network is represented. Masking just performs a point-wise multiplication on the weight matrix with zeros. Thus emulating a removed computation with zeros. When comparing this to Figure 2.17b it can be seen, that after almost every layer where filters are masked parameters are removed. This is due to the dependencies which occur in filter pruning. More on these dependencies is within Section 3.1. The resolution of these dependencies results in the removal of more parameters. Within these figures

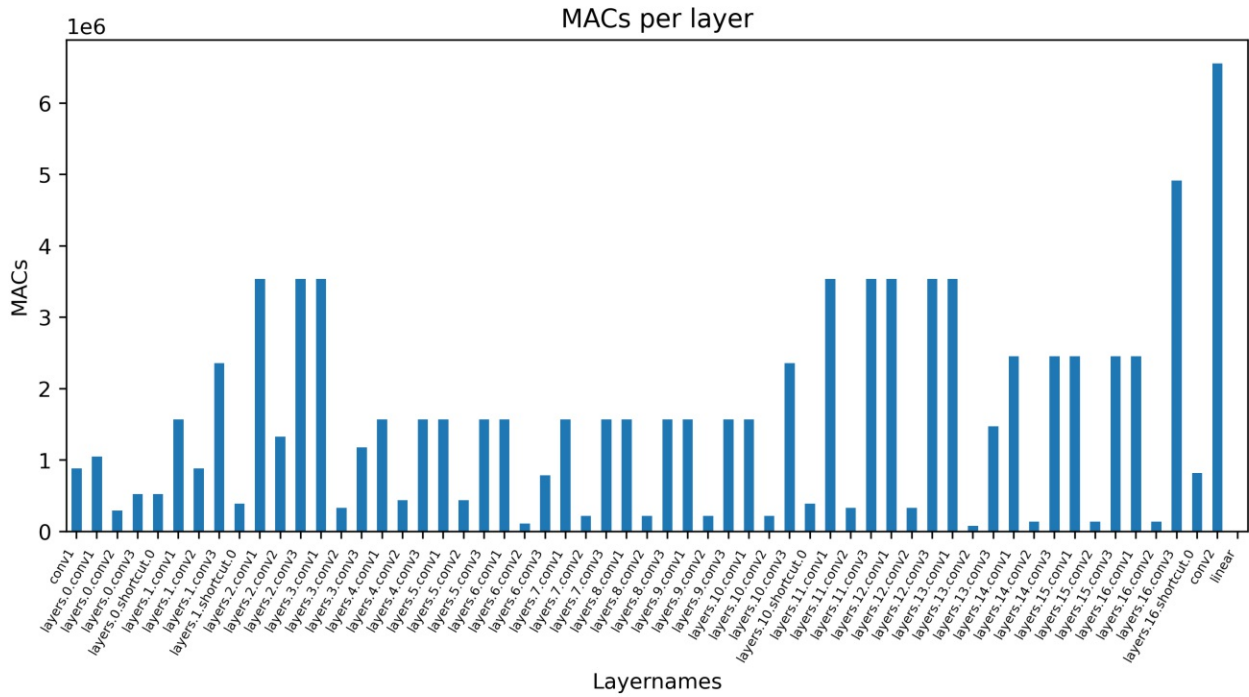


Figure 2.16: MAC count for a MobilenetV2 trained on Cifar10.

on the y-axis is the ratio of the original layer parameter count versus the parameters without the masked parameters (Figure 2.17a) and the original parameters versus the parameters of the thinned network (Figure 2.17b).

The cases within Figure 2.17b where the parameter ratio on the y-axis touches 0% is actually close to zero. A zero would namely relate to a disconnection between layers. As an example for the layer 9 convolution 3:

Original shape: $(64, 384, 1, 1) \rightarrow 24576$ parameters

Thinned shape: $(4, 20, 1, 1) \rightarrow 80$ parameters

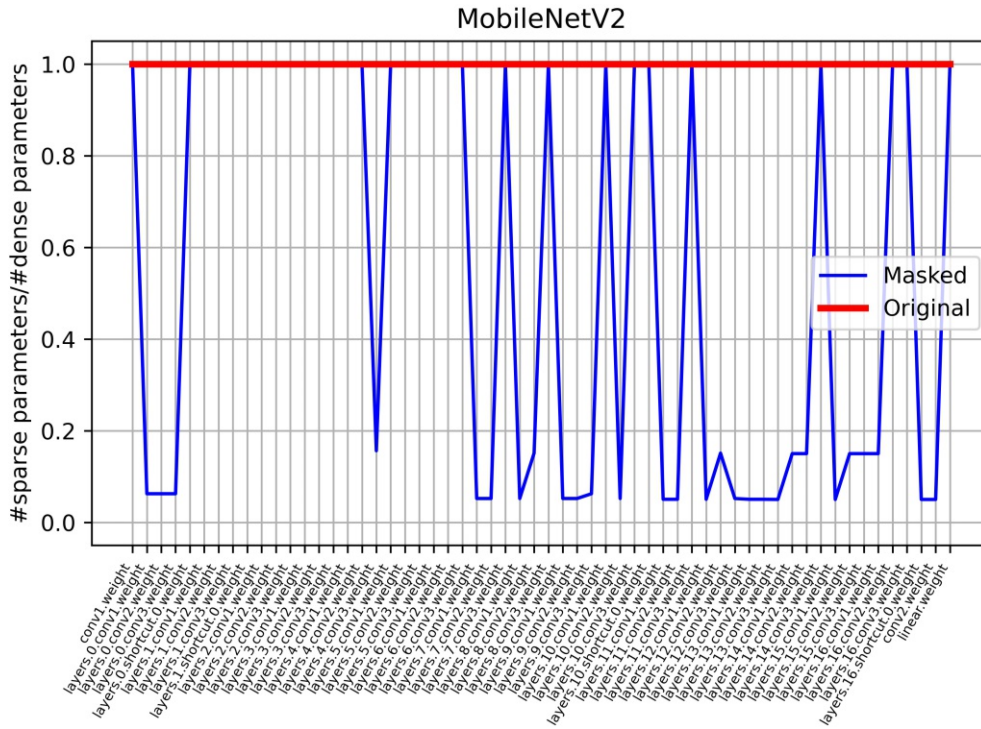
$$\frac{80}{24576} \approx 0.00325 = 0.3\%$$

2.3.3 Pruning

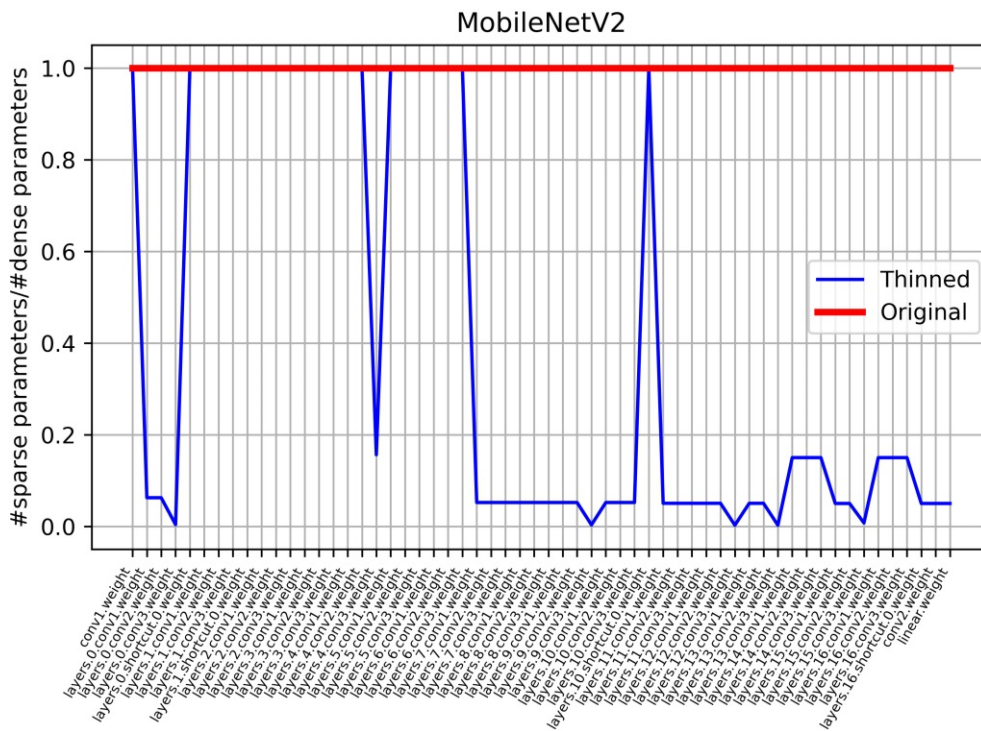
Distiller is at the time of writing one of the first frameworks which also give the possibility of really pruning the network. Real pruning in the sense of removing the computations and thus saving on computation time. To achieve this the thinning flow explained in Section 2.3.2 is used.

PyTorch Pruning

PyTorch offers some variation of pruning since the Release 1.4 [32]. The pruning part of PyTorch is also under active development and further techniques are added. The main caveat with these provided pruning techniques is the follow-



(a) Unthinned network.



(b) Thinning applied.

Figure 2.17: Comparison between a masked and a thinned network.

ing: It is only a masking of the network. With these methods, the original weights within the model get copied into a new parameter called weights_orig and an additional buffer with the computed mask gets added to the model. To not interrupt the usual workflow with PyTorch, an additional attribute gets added to the model which contains the weights used for the computation. This attribute gets calculated by using PyTorchs forward_pre_hook. The pruning can be

made permanent, but only in a sense of fusing the original weights and the buffered mask, to form a new weight tensor. Then the added `weights_orig` parameter gets removed together with the buffer. From this short explanation following things can be derived:

- Pruning in the sense of removing the weights is not possible as of version 1.6.
- This pruning procedure does neither speed up the computation nor make the model smaller.
- Contrary to what one would expect, the computation becomes even slower, due to the additional forward hook which computes the weights. Since this needs to be executed before every forward pass, it adds overhead to the required calculation.
- Every calculation which uses a pruned parameter is calculating it with the value zero. The pruning sets the value of the pruned weight to zero, to simulate removal of the calculation.
- In case of writing the pruned model into a file, it becomes bigger than the original model. This is due to the additional masks which are used for the calculation of the weights. These also need to be written to the model file, otherwise, the pruning would be lost.

This is not to say, that pruning with PyTorch is unusable. It has its respective use with researching the effects of pruning on the accuracy of the model. Furthermore, it can be used to test the effect, before making a permanent change in case a node removal might be added.

Keras Pruning

Pruning with Keras, which uses Tensorflow as a backend, is realized similarly as described in Section 2.3.3 [39]. The methods and the workings in the background might differ, but at the time of writing, neither PyTorch nor Keras offers pruning functionality with the removal of nodes (thinning as it is called within Distiller) out of the box. The only bonus is the saved space since the model becomes smaller, but only when compression algorithms are also used. For Keras, the same statement as above is valid, that for inference speed up this kind of pruning does not do anything. It still has its place in researching the effect of ‘removing’ nodes on accuracy. This is the state as of Keras 2.4 with Tensorflow 2.3 as backend.

Distiller Pruning

As of writing Distiller is the only framework known to the author which is capable of removing nodes from the model. Within Distiller, this is referred to as thinning to differentiate from pruning. Pruning is similar to the process described in Section 2.3.3. The network gets pruned by adding the so-called `zero_mask_dicts` which denotes the position of the pruning. The pruning then also gets executed in a similar sense by masking the used weights with this dictionary. This procedure is required to be done before each minibatch run. Otherwise, the zeroed weight would be affected by the computed change from the optimiser. Thinning as it is referred to, is a procedure that uses these `zero_mask_dicts` or the model directly to compute a thinning recipe. This recipe holds the layer parameters which get pruned and the layer names.

The basic pruning workflow which appears within Distiller happens in three major steps:

- creation of data graph
- creation of pruning recipe
- modify network according to recipe

Data Flow Graph The first important step is to determine the data dependencies within the network. Since this is not possible with the PyTorch model information, Distiller chooses to do this by passing dummy data into the network. While this data is traversing thru the network, the taken path gets recorded by replacing the usual forward hooks with additional methods which return the information on where they are located at the moment. With this information, a so-called Summary Graph is created.

Create Recipe Creation of the recipes happens in the following steps: In the first step, the created summary graph is iterated with the method `sgraph.named_params_layers()`. The 3 iteration variables are the following and used for the recipe creation:

layer name – The name of the currently iterated layer for example ‘features.module.14’

param_name – The name of the currently iterated parameter. This differs due to each layer might having multiple values depending on the layer configuration. An example for the parameters are the weights or the bias values.

param – The parameter itself. So for example in the case of the weights parameter this is a tensor that holds the weight values.

First, the currently iterated parameter is checked for 4 channels due to pruning only convolution layers. In the case of 4 channels, some common parameters get calculated such as the **nonzero_filters**. This value also needs to be greater than zero because otherwise there would be nothing to prune. The pruning recipe for the layer then gets created by using the `handle_layer` method. This recipe gets put into a special **ThinningRecipe** tuple which is the base for the thinning.

Within the `handle_layer` the channel count is checked again due to pooling layers might having also 4 channels. If this check passes a so-called module instruction is created and added to the **ThinningRecipe**. This module instruction consists of the layer name, what is removed (e.g. `in_channels` or `out_filters`) and how many of these are left after removal. This module instruction is mainly used to change the meta parameters of the layer within the PyTorch model graph.

Alongside the module instruction, a so-called param instruction is also created. While the module instruction holds the meta parameters for the PyTorch model, the param instruction holds the concrete indices which are left after removal. Within the current Distiller flow, a check happens if there is an already existing param instruction. If there is already an existing param instruction then the existing one gets ignored. These instruction get added to the **ThinningRecipe**-tuple which holds the module instructions and the param instructions.

With this, the first step of the recipe generation is finished. Theoretically, this is enough data gathering to prune the first layer. But as explained in more depth within Section 3.1 the dependencies need to be taken care of. First is checked if the same layer also has bias parameters. If yes an additional param instruction is created which differs in the param-name but has the same remaining indices. After this direct check, the graph gets checked explicitly for further convolutional

or fully connected successors using the `sgraph.successors_f` method. These successors get handled depending on their layer type.

After handling these dependencies the current layer gets also checked for an eventual BatchNormalization layer following it. The BatchNormalization needs to be handled differently due to the additional values this layer saves during training. For this layer then additional module instructions get created containing the new number of `num_features`. Furthermore are additional entries added containing the indices for `running_mean` and `running_var`. The param instruction for a BatchNormalization layer holds the indices for the weights and bias values within the layer.

Execute Recipe Within the `execute_recipe` method the created thinning recipe from Section 2.3.3 gets executed onto the given model. The execution happens in two main steps:

execute — Within this step the concrete recipe gets executed. The layers are changed according to the parameters from the recipe.

save — The recipe gets serialised and added to the model. This enables the pruning again in case of saving the network and continuing later on. The recipe gets added to a list in case of pruning multiple times.

The execution within the `execute_thinning_recipe` function is the core part of the thinning. The execution happens in two main loops. Within the first loop, all the module instructions are executed. These change the parameters, such as the number of output channels, as a whole for the selected layer.

Within the second loop, the dimension of the weight matrix gets changed. According to the param instruction from the `ThinningRecipe` the corresponding dimension gets selected. Together with this dimension value, the saved indices from the instruction are used to only select the layers which are not masked and put them into a new matrix.

Keras-surgeon

The Keras-surgeon [42] is similar in functionality to Distiller. It can remove channels, filters and layers within a given neural network. But it also has its limitations as it does not support all layer types and pruning dependencies which may affect an unsupported layer further down might hinder pruning. Keras-surgeon was neither used nor tested by the author and is listed here for completeness.

Chapter 3

Autopruner

The creation of an auto-pruner faces some main problems. The first one is pruning is exposed to a multitude of dependencies. Some of the easier ones exist only between two layers. More complex dependencies are far-reaching into the network and make the removal of layers hard or in other cases even not possible. The second problem an auto-pruner face is a determination of where to prune and how much to prune. A third problem is the pruning flow as it is presented within Distiller and described within Section 2.3. Since Distiller was created as a framework for research in a hands-on way, it is not inherently suited to fulfil the task of automatically pruning. It does offer an iterative pruning, but this is limited to a single layer, to which stepwise more sparsity is introduced. Another problem within Distiller is the utilisation of masking within the first part of pruning. Due to the masking, potential pruning problems are not visible until the thinning part of the workflow.

The problem regarding dependencies is closer explained within Section 3.1. How this problem is dealt with is explained in detail within Section 3.4.1. The determination of the pruning parameters is explained more clearly within Section 3.3.

3.1 Pruning Problems

Even though pruning has its advantages, it also bears one major disadvantage. The removal of a channel or layer poses a problem in that it changes the structure of the network. Therefore it leaves a hole in a place where data is expected to be. This is not in the sense of breaking up regular and optimised structures that are required for the mapping onto a GPU, but rather in a sense of dependencies as they are explained next.

3.1.1 Residual Pruning Problems

Distiller itself can handle simple dependencies, but simple dependencies are rather sparse, especially within more modern networks. For example, a residual block within ResNet which is explained in Section 2.1.2, looks as depicted in Figure 3.1.

Consider a residual block as shown in Figure 3.1. Going from this there are multiple places to prune.

With an arrangement as shown in Figure 3.1 there are now multiple possibilities to prune a filter. It is important to note,

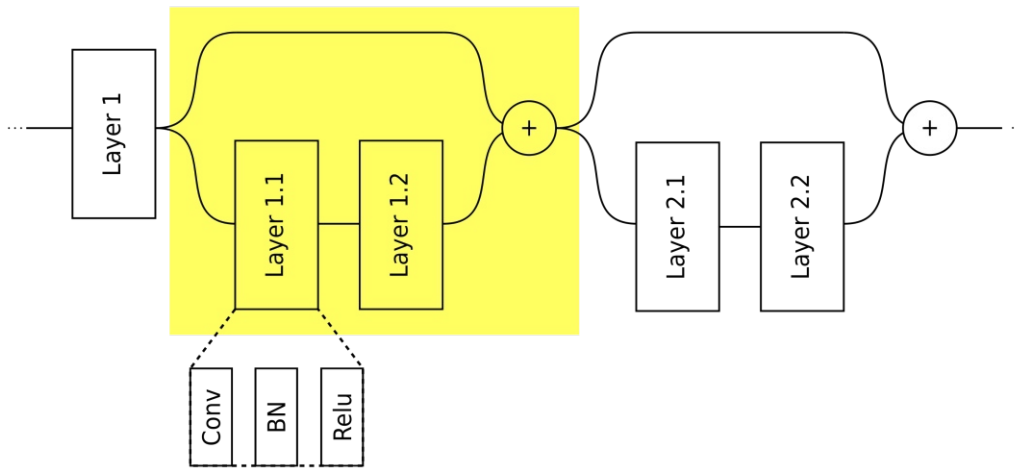


Figure 3.1: An exemplary block within ResNet. In this example, the residual block consists of an identity part, which is the upper path, and two convolution layers on the lower path.

that the elementwise addition expects two data matrices with identical shape. If this is not the case, the forward pass through the network will fail. To showcase the different cases, the base residual block is drawn as shown in Figure 3.2. This is to better show the removal and the effect this has on the feature maps. In this case, the input feature map is of size $1 \times 6 \times W \times H$. The 1 denotes a single batch with 6 channels of whereas each channel is W wide and H high.

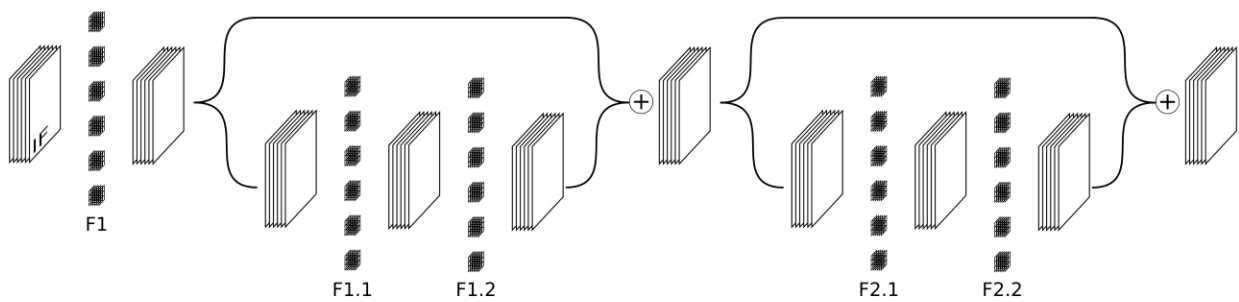


Figure 3.2: The redrawn residual block. This time showcasing the shapes of the filters and the feature maps.

One of the possibilities to prune is shown in Figure 3.3. A filter from F1.1 is removed. The removed filter is marked in red. The removal of this filter results in a missing channel within the input feature map for filter 1.2. To realign the shapes of the filters with the feature map, a change within each filter kernel is needed. The corresponding channel within the filters is removed. The removed channels are shown in Figure 3.3b in light blue. Since the filter count within F1.2 does not change, the output feature maps of F1.2 has the correct shape for the pointwise addition. The other feature map for the pointwise addition is the output of F1. This is the simple case, where no further dependencies arise aside from the channel removal.

The second possibility to prune is shown in Figure 3.4. In this case, the selected filter to prune lies within F1.2. This filter removal results in a missing channel in the output feature map of F1.2. Due to this missing channel, the shapes for the pointwise addition between this and the output feature map of F1 do no longer fit. The now missing channel is marked in red within Figure 3.4a. To resolve this, the corresponding filter kernel within F1 needs to be removed. As a result of this removal, the output feature map of F1 now fits with the shape being similar to F1.2. But two other problems arise with this. The first one is that now the shape of the input feature map for F1.1 is not anymore what was expected. Therefore the corresponding channel within all filter kernels in F1.1 need to be removed. The second

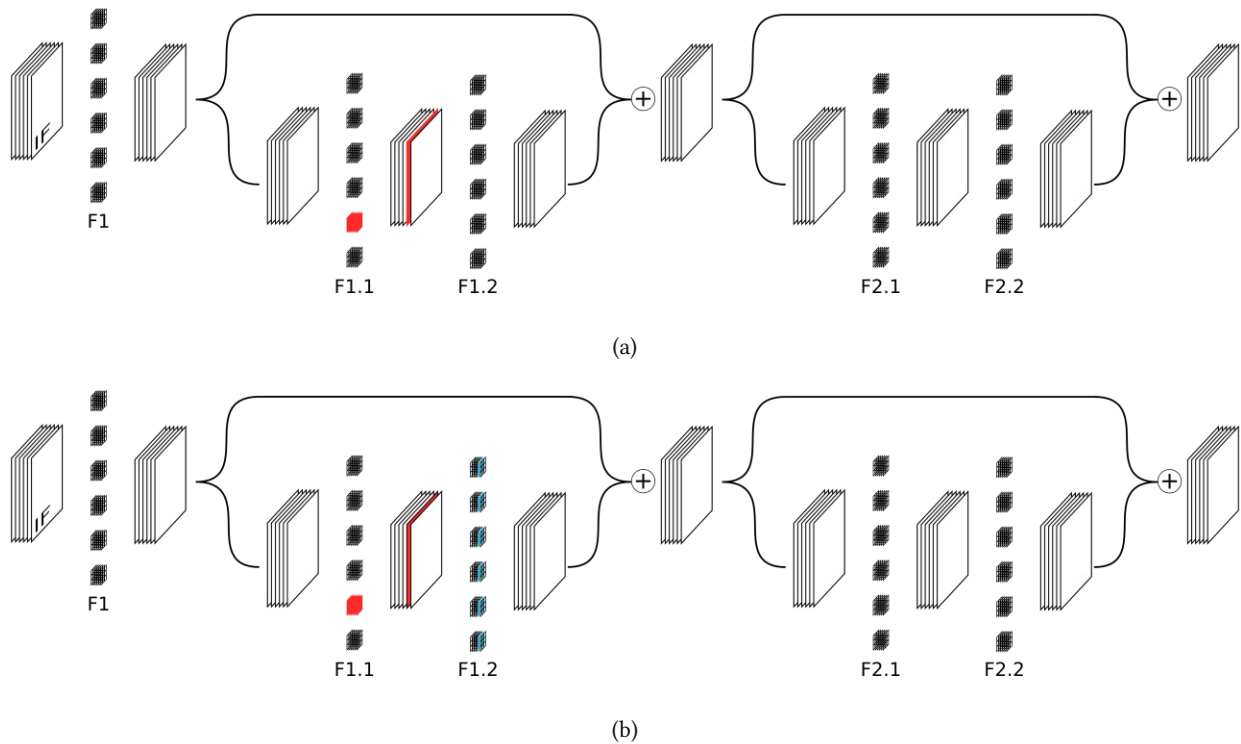


Figure 3.3: Pruning a filter from F1.1. The resulting dependency is drawn in red and the channels which need to be removed in light blue.

problem is essentially the same, but at a different place in the block. The shape which F2.1 expects now also do not fit anymore. Therefore a channel removal within the filter kernels in this place is also required. Furthermore to now fit the shapes for the pointwise addition, a filter kernel within F2.2 needs also to be removed. This problem now may or may not reproduce itself further into the network, depending on which layer comes next. In the case of further residual blocks, as described above, it will propagate. If the next layer is a convolution layer, then the propagation ends with a channel removal in this layer.

From this example, it becomes apparent, that pruning within a residual structure might hold big losses regarding accuracy. This is due to the additional removal required to again match the shapes.

The third and last possibility is removing a filter within F1. This is the last possibility due to pruning within the F2.x part is similar to pruning in the F1.x part of the network. The filter removal results in a change of shape for the pointwise addition between the outputs of F1 and F1.2. Furthermore, the channels now do not fit for layer F1.1. To cope with these two issues, a channel within the filter kernels of F1.1 needs to be removed and further a filter from F1.2 needs to be removed. The pointwise addition now has a different shape than the original network. Therefore further adjustments need to be made to F2.1 which uses the feature map directly and F2.2 where the output feature map needs to be of the same shape. All these direct dependencies are marked in red within Figure 3.5a. To resolve this the channels within the filter kernels in F2.1 need adjustment and one filter kernel in F2.2 needs to be removed. As before with Figure 3.4 this dependency can reach farther into the network depending on the layers in between.

The second and the third result in an identical network at the end. Even though the pruning would have started with completely different filters. The dependencies can also reach backwards in the network. Take for example the network as shown in Figure 3.2. If one filter within F2.2 is removed this would result in the same network at the end as if a filter

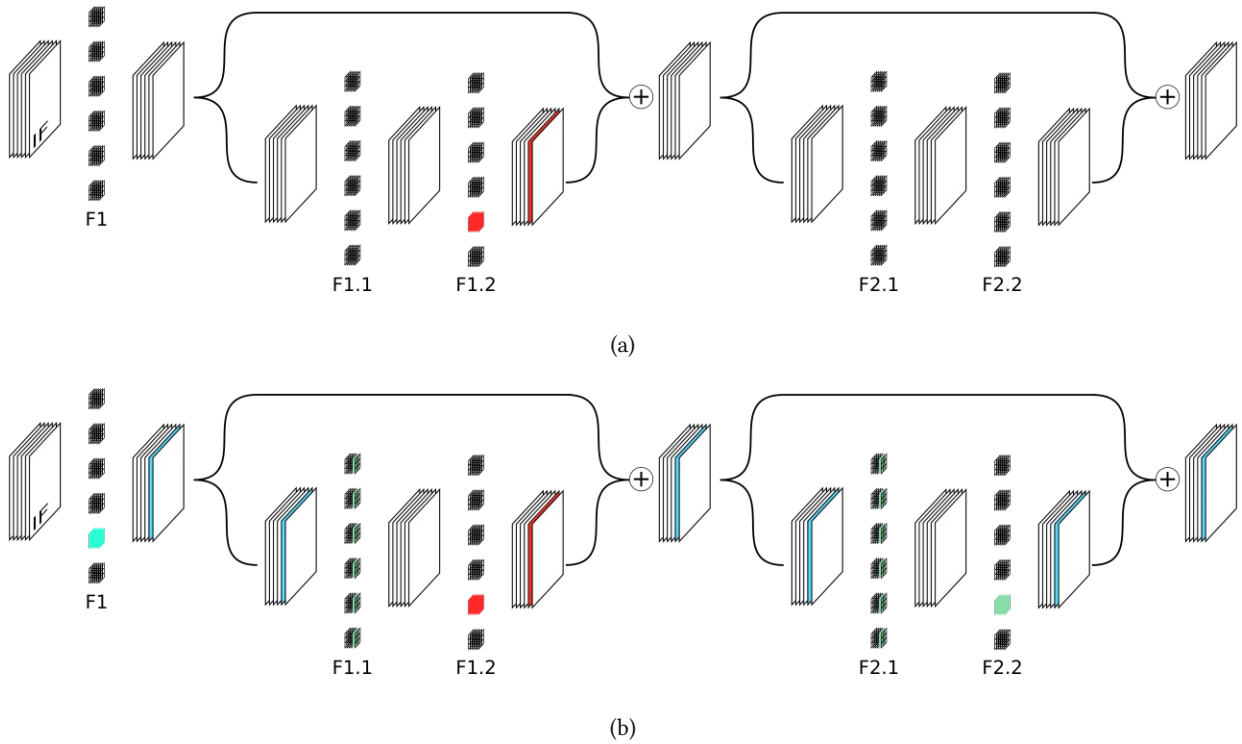


Figure 3.4: Pruning a filter within F1.2 results in an unexpectedly high amount of dependencies to resolve. These dependencies degrade the accuracy of the original net.

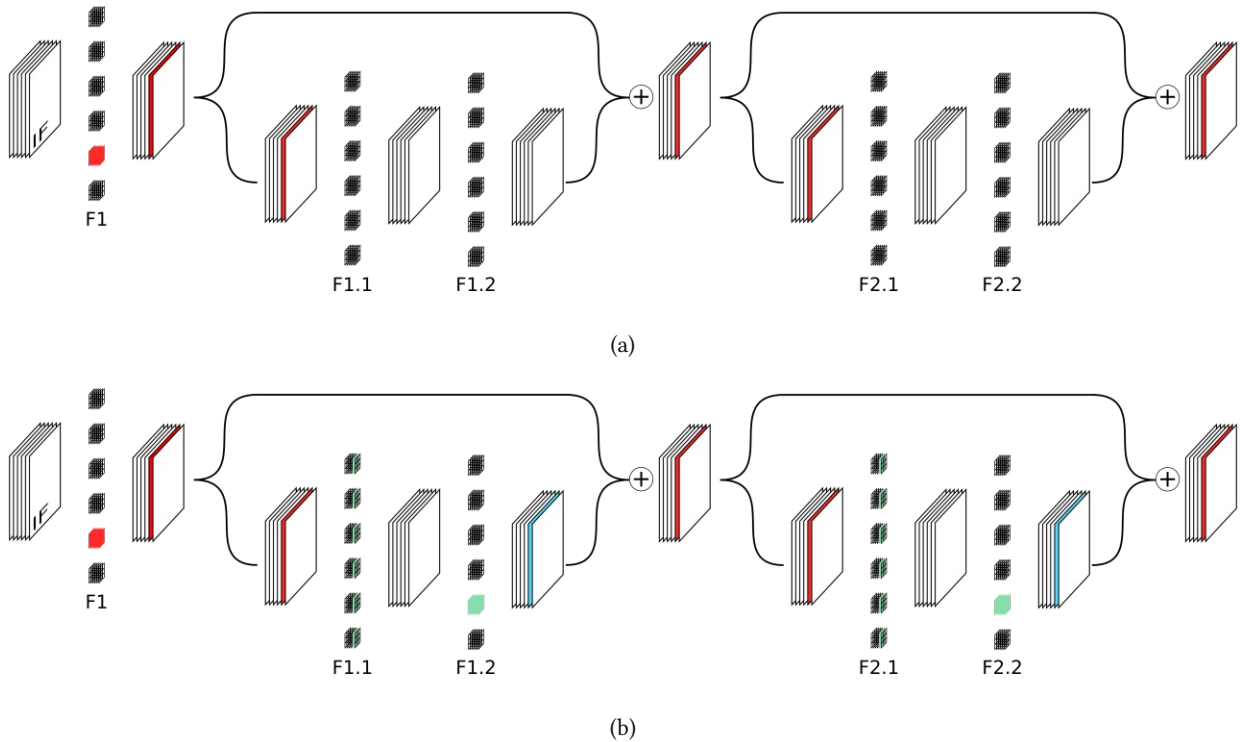


Figure 3.5: Pruning a filter within F1 affects multiple Feature Maps (FMs). This results in influencing all other depicted filters in various ways.

within F1 or F1.2 is removed. The dependencies would propagate back and require a change there. This also showcases a problem with the currently used approaches to pruning present in PyTorch or Keras as discussed in Section 2.3.3. Due to only masking all these dependencies which would also need to be removed are ignored, since zero-masking the values

does not influence the structure of the filters or the feature-maps respectively.

3.1.2 Groupwise Convolution Problems

Groupwise Convolutions bear a whole other level of problems. Due to the nature of the groupwise convolutions, the pruning becomes dependant on the filter combinations. Since it is not possible to remove a filter and keep the remaining filter operating on the original patch of input data. This problem is untreated within the machine learning domain. To further illustrate why there is a problem with groupwise convolutions, the working of a groupwise convolution will be explained.

Within groupwise convolutions, the input feature map is not used with the convolution as a whole but separated into groups. These groups divide the channels and for each subchannel, a smaller convolution kernel is required. With this lesser parameters are required within each filter kernel and thus lesser computations.

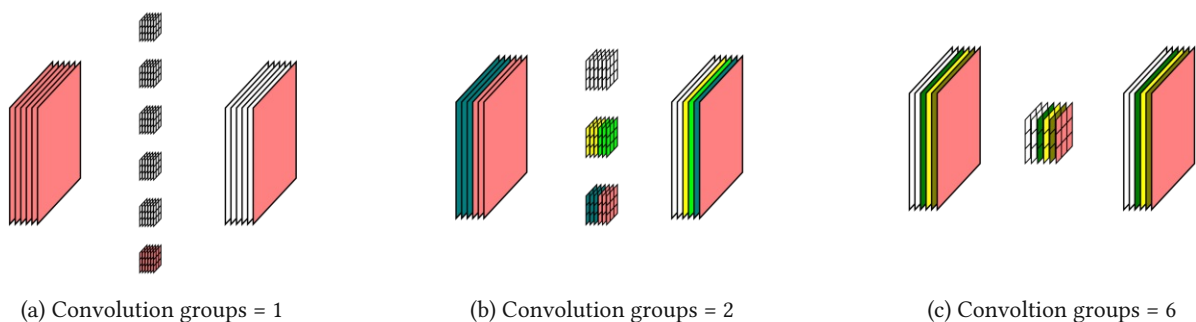


Figure 3.6: Illustration showing the working of groupwise convolutions. The used channels from the input feature map are depicted in the same colour as the filters which are using it. In the case of multiple filters using the same input features only one colour of a used filter is shown.

Within Figure 3.6 a convolution with 3 different group parameters is depicted. The input feature map is of the size $B \times C \times W \times H$ with $B = 1$ and $C = 6$. B and C stand for the batch size and the channel count respectively. The groups parameter has some constraints. The groups parameter controls the connections between the inputs and the outputs. Furthermore, the input channels and the output channels must both be divisible by the groups parameter.

The first Figure 3.6a shows the classic convolution. Within this case, each filter kernel is convolved over the whole input feature map and thus creating an output feature channel. The filter matrix is of size $6 \times 6 \times 3 \times 3$ meaning that there exist 6 different filters with 6 channels each. This is also coloured in light red. The red-coloured filter kernel uses all six input channels and creates a single output channel.

Within the second Figure 3.6b a setting with two convolution groups is depicted. Each filter kernel now only has three channels for their respective three input channels. This reduces the initial size by a factor of 2. The filter matrix now is of size $6 \times 3 \times 3 \times 3$. Again with six output channels, but each filter kernel now has only 3 channels. Removing a single output channel is not feasible at the moment of writing within PyTorch. Doing so would break up the convolution nature. A possible solution would be to remove all input channel of the first half (depicted in green) and also remove all corresponding filters which use this part of the input feature map (green and yellow in the coloured kernels and the latter three channels of the white kernel). Doing so would probably result in a big drop in accuracy.

The third Figure 3.6c shows the extreme case of groupwise convolutions. This is the case when the groups parameter is equal to the count of input channels. Thus it results in a single channel filter for a single channel of the input. In the

example with groups equals 6, each channel now has its own 3×3 filter kernel. And each of these filter kernels creates its own output feature map. Different to the aforementioned version in which the groups lie in between 1 and the channel count, this version is in fact prunable. Not only that but it is also named especially. Grouped convolutions where each input channel has its own 3×3 filter for a separate output channel are called depthwise separable convolutions. In the case of removing a filter, this means a reduction of the input channels and the output channels.

Pruning is therefore only possible in the case of removing a whole filter group and not a single filter kernel. Therefore resulting in more coarse pruning and might removing filters that hold valuable data.

3.1.3 MobileNet Problems

The problems within MobileNet are similar to the problems in Section 3.1.1 combined with the problems from Section 3.1.2. Due to the optimisation of the conventional 3×3 convolution and splitting these up into the depthwise separable convolutions combined with pointwise convolutions in MobileNet [18]. In MobileNetV2 these parts are also used within residual blocks [36], thus increasing the pruning problems. Especially the pointwise and the depthwise convolution result in a rather high dependency, depending on which layer is chosen for the pruning.

To showcase the high dependencies, consider a standard block within MobileNetV2 as can be seen in Figure 3.7. These standard blocks are used in a residual manner throughout MobileNetV2 with varying channel counts. Within this standard block, there are 3 possible places to apply filter pruning. Meaning removing a filter to create an output feature map less.

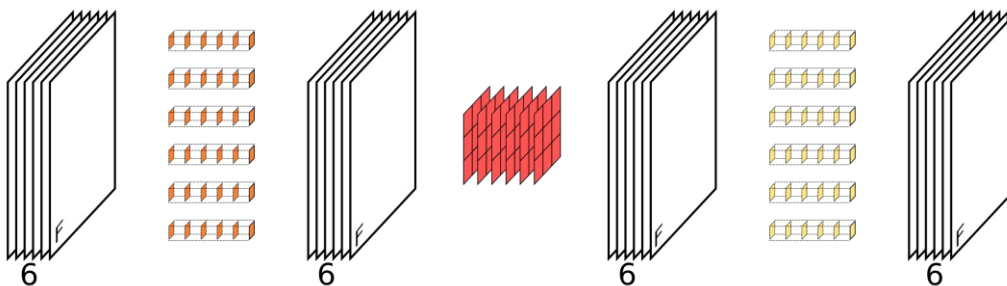


Figure 3.7: MobileNetV2 Standard Block consisting of a pointwise convolution, followed by a depthwise convolution followed by a pointwise convolution.

The first possible place to apply filter pruning is the first pointwise convolution. This filter kernel marked in red is the one that is going to be removed (Fig. 3.8a). Removing this kernel would lead to a reduction in the channels of the corresponding output feature map. This is marked in light blue within Figure 3.8b. Furthermore, the kernels requiring the data of this missing output feature map also need to be removed (also marked in blue). Since this is a groupwise convolution the constraint of output channels equals input channels equals filter count needs to hold up. Therefore a whole filter is removed. The removed filter would create the channel marked in green (illustrated within Figure 3.8c). Figure 3.8d shows the pruned network with updated channel counts.

The second place to apply pruning would be the groupwise convolution illustrated within Figure 3.9. In this case, a groupwise convolution marked orange in Figure 3.9a is pruned. This direct removal results in one lesser output channel and due to it being a groupwise convolution also necessitates a reduction within the input channels. These direct dependencies are marked in light blue within Figure 3.9b. To accommodate these needed removals, both pointwise convolutions, the one before and the one after need to be changed. The first pointwise convolution needs the filter removed

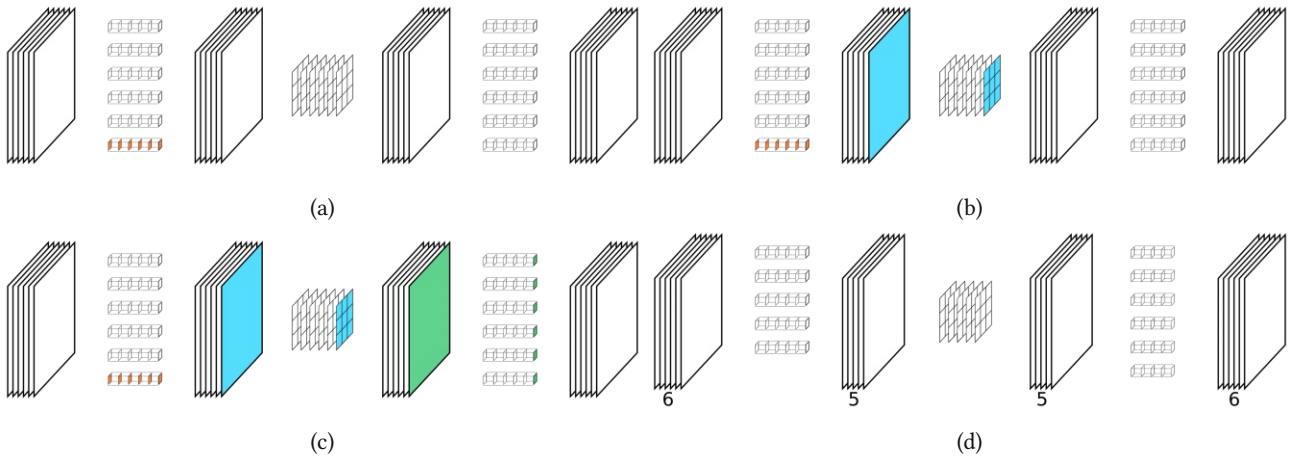


Figure 3.8: MobileNetV2 Pruning Variant 1

which would generate this data. The second pointwise convolution requires a change in all filter removing the channel which would use the removed channel. If one compares Figure 3.9c with Figure 3.8c it becomes apparent, that these are identical. This showcases that it would be better to use a combination of both filters to evaluate the impact on the accuracy. Furthermore, this holds potential speedup in the evaluation phase which is explained in Section 3.3.1.

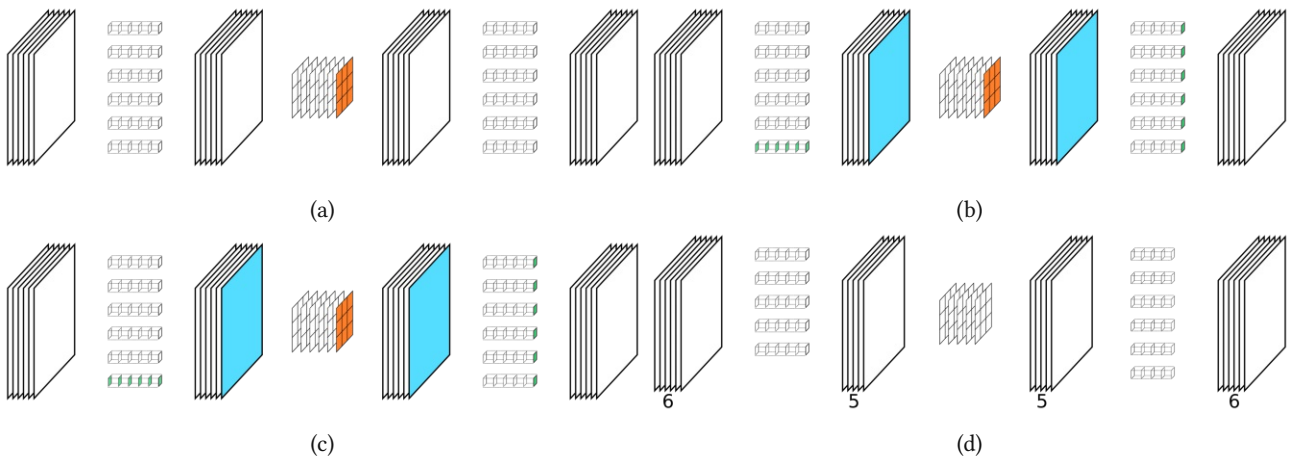


Figure 3.9: MobileNetV2 Pruning Variant 2

The third possibility would be the pruning of the second pointwise convolution which is marked orange within Figure 3.10a. The removal of a filter on this position results in one lesser channel within the fourth feature map. To accommodate this change other parts, not within this block need to be changed. For example, a convolution layer after this block requires a change in its channels or changes within BatchNormalization layers afterwards. The reach of pruning a filter on the last place within a residual block is further explained within Section 3.1.1. This variant also results in a different network than the first two approaches which target the first pointwise convolution or the depthwise convolution.

Even though this is a well-defined problem, the implications of removing unwanted channels in places, which then might propagate before the targeted kernel, makes pruning in such places rather unwanted. To elaborate further on why these are hard to remove, all while ignoring the fact that this might have unwanted side effects in the accuracy, for example, the pruning process itself needs to be better understood. The Distiller pruning process, works on one layer after another, by resolving the zero_mask_dict and creating the corresponding thinning recipes. Due to this, an order

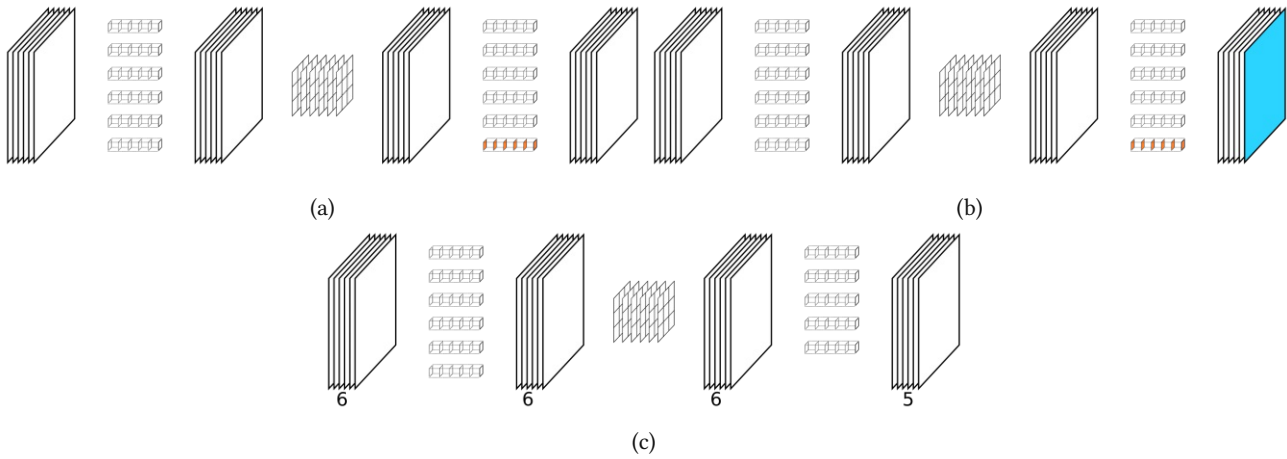


Figure 3.10: MobileNetV2 Pruning Variant 3

is made within the network. Now in the case of filter pruning which would change the network layout in a layer that has already been pruned might become complicated to track and therefore results in a failed pruning recipe.

3.2 Solutions

To these presented problems, solutions were also created and implemented within Distiller. Distiller is already offering a solid base for extension but was unable to prune residual blocks and also had problems with groupwise convolutions.

3.2.1 Residual Solution

To solve the presented problem within Section 3.1.1 two approaches were envisioned. The first one was to consider the layer position within the network. This would mean, that layers that are at an unpruneable position would not be pruned. This approach is rather inappropriate due to essentially losing the capability to prune big parts of the network. In a residual block as presented in Figure 2.7 this would remove Layer1.2 from the list of possible prunings. Considering the fact, that for example, ResNet consists mostly of such presented blocks, this would mean a severe impact on the reachable sparsity. Furthermore, in the case of more sparse convolution matrices within these layers, this sparsity could not be leveraged and would simply waste energy and time during convolution.

The second more valuable approach is rather simple. The reason for these dependencies is due to the convolutions. These convolutions require fitting shapes between the kernels and the feature maps. Changing a kernel implies changing the feature map, therefore requiring a change within the kernels of the next layer. Therefore the reason is the change within the feature map. To avoid this change, the original feature map shape should be maintained. This conservation of the original shape is acquired by creating a zero matrix with the shape from the original convolution. The pruned convolution values are then embedded within this zero matrix to transform it back into the correct shape. Even though this approach sounds rather costly at first, it can be shown to be quite feasible and also effective. Within Figure 3.11 the working of the zero-padding is shown. The shown block is similar to these within Figure 3.2, but with a filter removed within the second convolution. To avoid going down the dependency chain as shown in Section 3.1, the output is padded back to shape. It is not completely correctly shown within this Figure, but essentially the input feature map to the pointwise addition operation receives a feature map with one channel consisting fully of zeros.

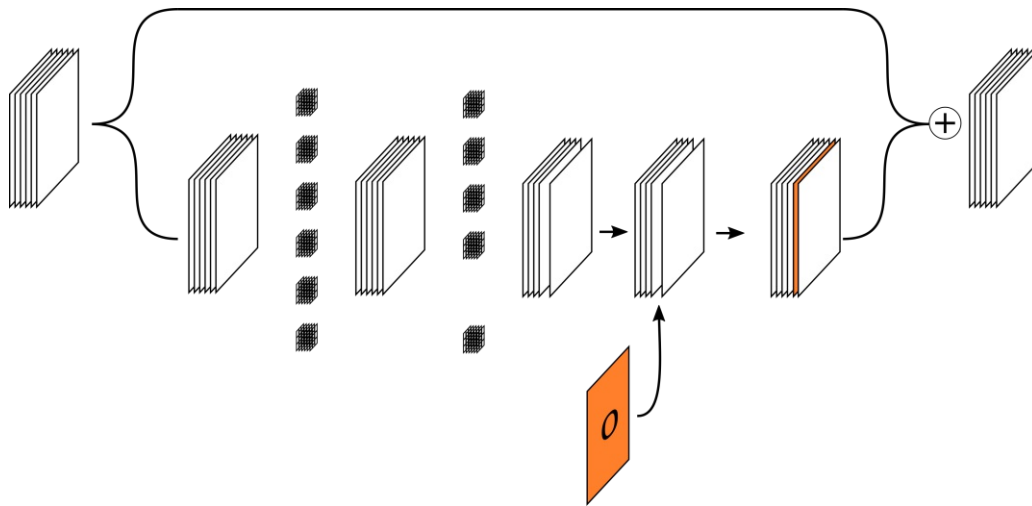


Figure 3.11: Zero-padding to get the output back into the correct shape.

3.2.2 Groupwise Solution

Since pruning groupwise convolutions are not possible, only the case of the groups parameter equals to the channel count is considered. To be more specific the depthwise convolutions. Distiller is almost inherently able to prune groupwise convolutions. In fact, almost everything is there in place. But for some reason, the groupwise convolution dependencies are only resolved correctly in the case of pruning the first convolution in a groupwise pair. In the case of choosing the depthwise convolutions for pruning, this case does not get resolved correctly. To consider this case too, the pruning flow gets changed slightly, to also check if the convolution layer is a depthwise convolution. The refined pruning flow is presented in Figure 3.12.

3.2.3 MobileNet Solutions

The presented solutions within Section 3.2.1 and Section 3.2.2 automatically solve the problems for depthwise-separable convolutions existing within with MobileNet. The groupwise part is taken care of with the addition in the pruning flow. The case of the removal of a filter kernel before a pointwise operation is also considered by the zero-padding.

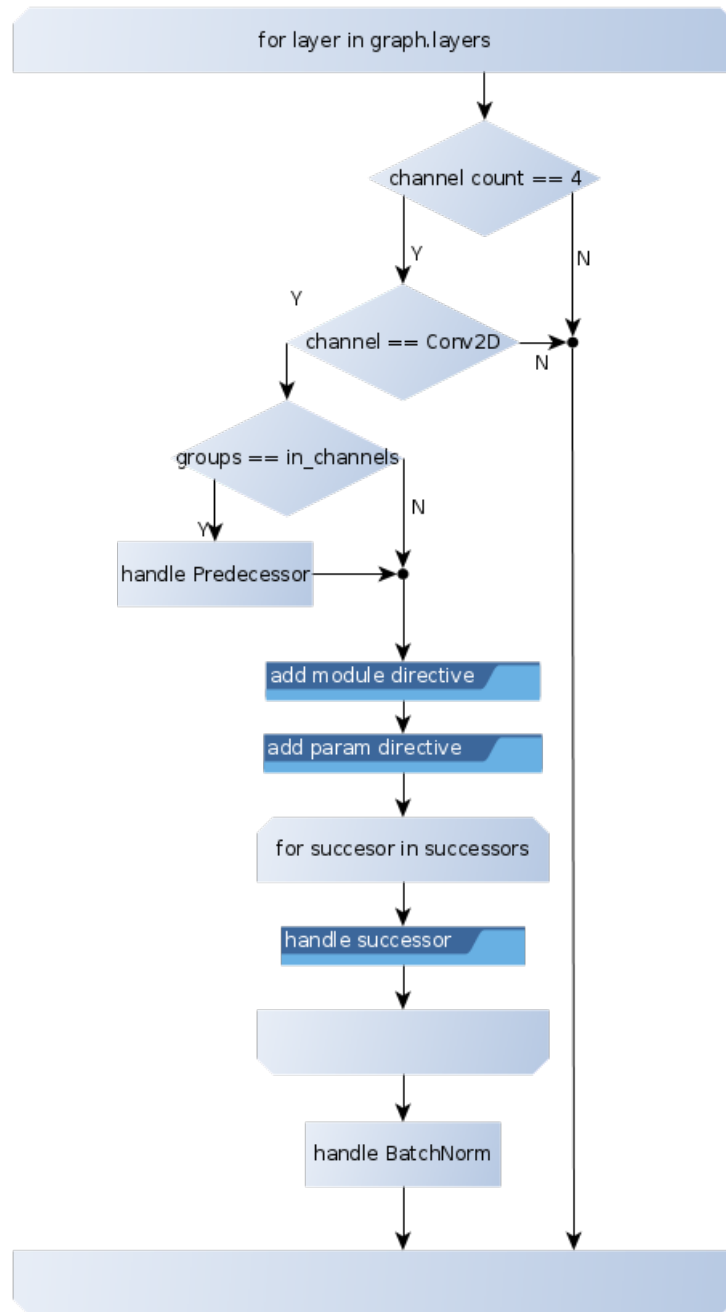


Figure 3.12: The refined pruning recipe generation. The newly added blocks are the decision and the handle predecessor blocks.

3.3 Pruning algorithm

The basic workflow for the pruning algorithm is shown in Figure 3.13. Each of these blocks is explained in the sections to come. Most of the tools used within the workflow were available within Distiller or are built upon those available. The output of this algorithm is a masked model which is then fed into a thinning function. This has the downside of not speeding up the training process. But due to working on the image classification problem, the training speed is not too high and thus this downside is ignored.

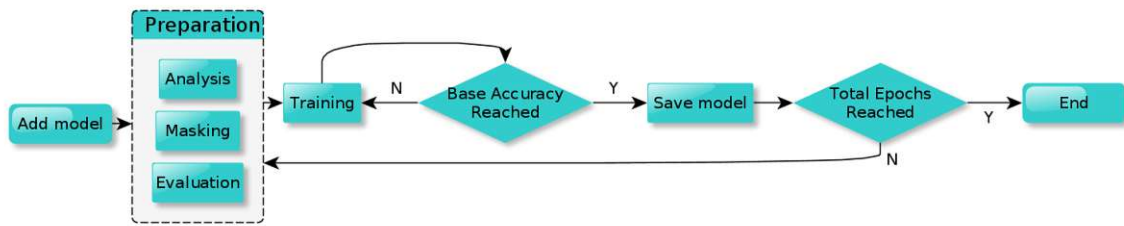


Figure 3.13: The created and used pruning workflow.

3.3.1 Analyse

Within the Analyse-block, the network gets analysed with regards to pruning and the effect of the pruned layers onto the final accuracy. This is done by utilising the presented capabilities of Distiller within Section 2.3. The main core of this block is the sensitivity analysis. The sensitivity analysis prunes each convolutional layer in 5% steps and runs inference on the validation set. With this data, the following graphs can be created as can be seen in Figure 2.14. In the graph can be seen the analysis of a network, with all convolutional layers, each pruned independently. The values range from 0% sparsity up to 95% sparsity. The reason it goes only up to 95% is due to a layer pruned to 100% would be equivalent to removing the whole layer which would leave a hole in the network. Distiller is not able to handle such a case. But it is to mention, that this is not unrealistic, as it is possible due to the residual nature of some networks. Within such multiple paths which split and merge exist and therefore a single such path would be able to be completely removed.

This data serves as a good base for pruning. The problem is to find which layers are more pruneworthy and which are not. This problem is more present, if the network already contains pruned layers. This is due to the effect of the pruning now needs to be considered against more factors which are:

- Accuracy
- MAC count
- Weight removal

A downside to this measurement-based approach is the following: It takes an insane amount of time to complete. The required time for testing the influence of pruning on the model is directly depending on the size of the dataset used for the evaluation. As an example, using the evaluation set for evaluating the accuracy influences on a network with Cifar10, each layer evaluation takes around 500 seconds. This time requirement scales with the dataset, the bigger the dataset, the longer it takes. As a further example, a single layer for a MobileNetV2 trained on ImageNet took around 3600 seconds. To speed this process up, a small subset of the original data was used for the evaluation. It has shown, that 1% of the original data is in the case of Cifar10 and ImageNet enough to get good enough evaluations. The deviation between these is visualised within Figure 3.14. On the y-axis, the difference between the evaluated accuracy difference with the small and the full test set is visualised. Along the x-axis the different sparsity steps. As can be seen, a mentionable deviation starts to happen at around 80%. Even after 80%, the median difference stays below 2%.

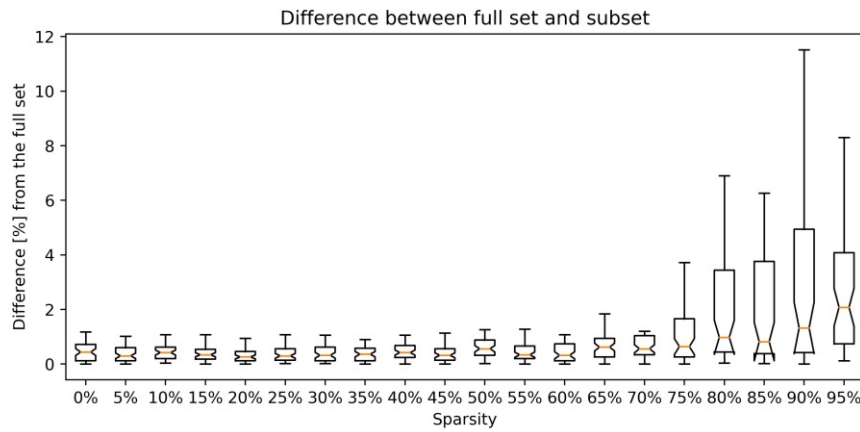


Figure 3.14: Showing the deviation of the full set for evaluation and a small subset.

Layer Sorting

The main sorting happens with the implemented close-accuracy points (CAP) Close-Accuracy-Points sorting. With CAP sorting, the mean sum of all layers gets accumulated and multiplied by a factor. This factor is determined, by counting how many accuracy points fall within a given margin around the base accuracy. In the case of an already pruned layer, this layer has fewer accuracy points within its evaluation and would thus score more badly with CAP sorting. CAP sorting is therefore used as a sorting function with the built-in python sort function. The return values of the sorting is the mean value multiplied by the factor determined by the number of values within the threshold.

Point Search

For the determination of which sparsity points are a good fit, an approach using the gradient is implemented. This approach is used instead of simply searching for a given deviation from the base accuracy. Finding the right and fitting constant is hard and all accuracy lines follow the same principle. They have a flat slope and most of them end in a sudden steep slope. Finding the point before the sudden steep slope is the point aimed for. For this purpose, the gradient is calculated and checked if the gradient is greater than a given value since this would indicate a sudden decrease in the slope.

3.3.2 Prune Best Layers

This part of the algorithm deals with the actual pruning. The Analyse-block explained in Section 3.3.1 just evaluates all layers. It returns what according to the CAP-sorting seem like the best layers to prune. The network stays untouched. This block takes a list of pairs consisting of the layer name and the desired sparsity within this layer. This leads up to the next decision to make. As explained in Section 2 the filter kernels which shall be removed need to be selected. There are multiple ways to select the kernels to remove. The chosen variant within this work is the usage of the ℓ_1 norm. This is also the method that is applied by [27] to prune different networks by using sensitivity analysis and one-shot pruning. A short review of the used filter pruning is given in the next section.

Filter Pruning

As already mentioned, the used method to sort and prune the filters is by measuring the importance of using

$$\sum_c^C \sum_i^I \sum_j^J |F_{c,i,j}| \quad (3.1)$$

which equals the ℓ_1 -norm. C, I, J stand for the channel dimension, kernel width and kernel height respectively. This bases on the assumption, that filters that hold little to no value have many values which are close to zero. Therefore they contribute little information and thus are not noteworthy when removed. This is also visualised in Figure 3.15. As can be seen, there is a multitude of filters. These are from the first convolutional layer within a MobileNetV2. Of these filters only a few carry distinction. The others, which have close to no distinction and information, are those who are later removed by the pruning process.

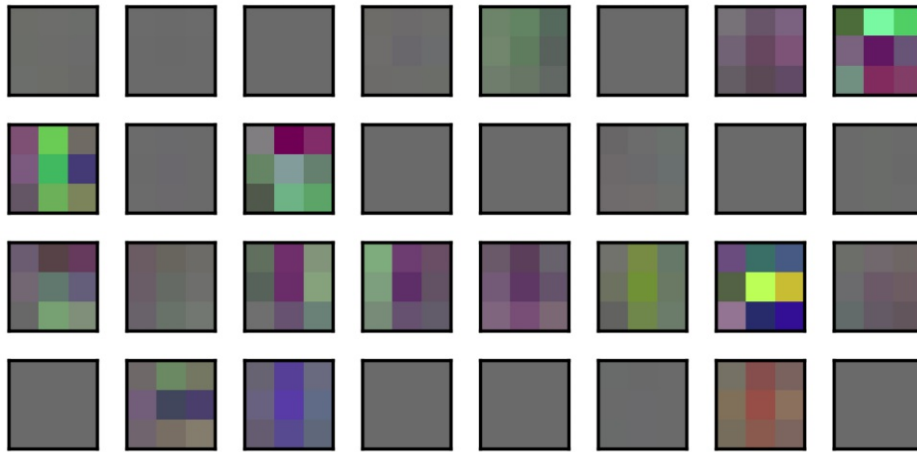


Figure 3.15: Filter Visualisation for a MobileNetV2 trained on Cifar10.

Distiller Integration

Since Distiller in its original form only supports config files, the original flow was analysed. From the gathered information, a wrapper was created which calls the pruner creation functions and add these to the model. The adding happens via the `compression_scheduler` object present within the scheduler. This scheduler object holds all pruning policies required for the process.

Dependency Masking

The original Distiller pruning flow just masks the weights. This masking is in the case of normal convolutional layers on top of each other no problem. But in the case of combinations with bias and groupwise convolutions a problem arises. Namely, values that should be zero after a layer are not zero anymore due to masking only the convolution part. This problem presents itself during the whole pruning algorithm. Due to areas being not zero, which should be zero, the retraining is based on a wrong basis. This results in a high training accuracy, but after the thinning is completed the accuracy is butchered.

For mitigation, the dependency resolving which happens at the thinning part is extended to also work for the created filter masks during the masking. The pruning algorithm with the pruner creation was expanded to also resolve these to gain accurate data on the influence of removing filters. This expansion also includes resolving all the problems mentioned within Section 3.1.

This dependency masking is built upon the already existing methods, which are used in the Distiller flow. The layer masks are held within the `compression_scheduler` object and a zero corresponds to a masked value. This mask is built from a view generated on the calculated filter norms. This generation has a downside. It is in the normal matrix not possible to select filters and channels within one mask. This is due to the mask being only a view and thus acting like it has the correct dimension, whereas it is of lower dimension and projected up to fit the original dimension. This saves on space within memory but disables the possibility to set specific entries within the mask. To circumvent this problem, the matrices were generated as one matrix with equal shape and the corresponding rows and columns, for the filter- and channel-pruning respectively, set to 0 for the masking operation. The masking happens with the usage of an in-place multiplication offered by PyTorch with the `_mul` method for a tensor.

3.4 Implementation

This section describes the implementation process of the zero-padding in greater detail. Measurements used in conjunction with zero-padding are located within Section 4 as this Section only focuses on the implementation details.

3.4.1 Zero-Padding

Due to the usage of PyTorch, the implementation of zero-padding is not as straightforward as one would assume. PyTorch operates on the principle of the code models the graph. Due to this concept, PyTorch does not allow changing the graph in a way that differentiates it from the original graph. Although it enables expanding the graph as in adding more layers during runtime. But the use case of opening an existing connection and inserting a layer in between is not possible with the standard `torch.nn` packages. What PyTorch does support are hooks. With these hooks, a function can be added which is then executed at a pre-determined point in time during the execution. So the fact that changing the computation graph is not possible in the desired way is circumvented by the usage of hooks.

There exist up to 3 possible hook locations for each `nn.module` created layer. These 3 points are:

Backward hook is called whenever the gradients corresponding to the inputs are computed.

Forward hook is called every time after the forward method of an `nn.module` has been called. This is the point in time when the computation for this layer has ended. But before the outcome is passed on to the next layer. A hook using this point in time has access to both the input and the output of a layer. Therefore this hooks is usable for calculating statistics for a single layer. Furthermore it offers the ability to change the output.

Forward pre hook is called before the forward method for the `nn.module` layer is called. These hooks can be used to modify the input to the layer.

Each function for a hook has a defined signature. These are defined within the documentation [4] for the `torch.nn` module. The forward hook is used due to its ability to change the output of a layer. Therefore it fits in terms of

functionality and corresponding to Figure 3.11.

This solves the problem of how to implement this functionality without changing the original network description. Due to the usage of a forward hook, the signature for the function is also given and not changeable. The function has access to the module or layer itself, the input tensor and the output tensor. Zero-padding requires knowledge about the indices which are removed. To give this knowledge to the function an additional attribute is added to the module layer itself by using the python builtin `__setattr__`. With this method, the module is expanded with the indices which are determined during the thinning process. Therefore giving the hook access to the required indices information.

3.4.2 PyTorch-Implementation

With this, all required information is now accessible for the function. The concrete implementation happened in 4 versions. The first version was implemented as a for-loop. The loop then iterates over the tensor along the second dimension. At each iteration is checked if the current index is within the remaining indices. For the first match, the OFM is sliced, the corresponding output channel is inserted and the tensor is then put back together. The second matching indices slices again the whole feature map two times and puts it back together. The downside of this implementation is blatantly obvious. For loops are not an efficient way for this kind of operation. Furthermore were the slices copied using the PyTorch colon notation and with this further problems were introduced which are the focus of Section 3.4.3. The main reason for this approach being slow is visible within Figure 3.16. For each added iteration a yellow block is added to the network. In this case, an iteration is similar to a removed filter. Therefore meaning each filter removed is adding additional blocks to the graph.

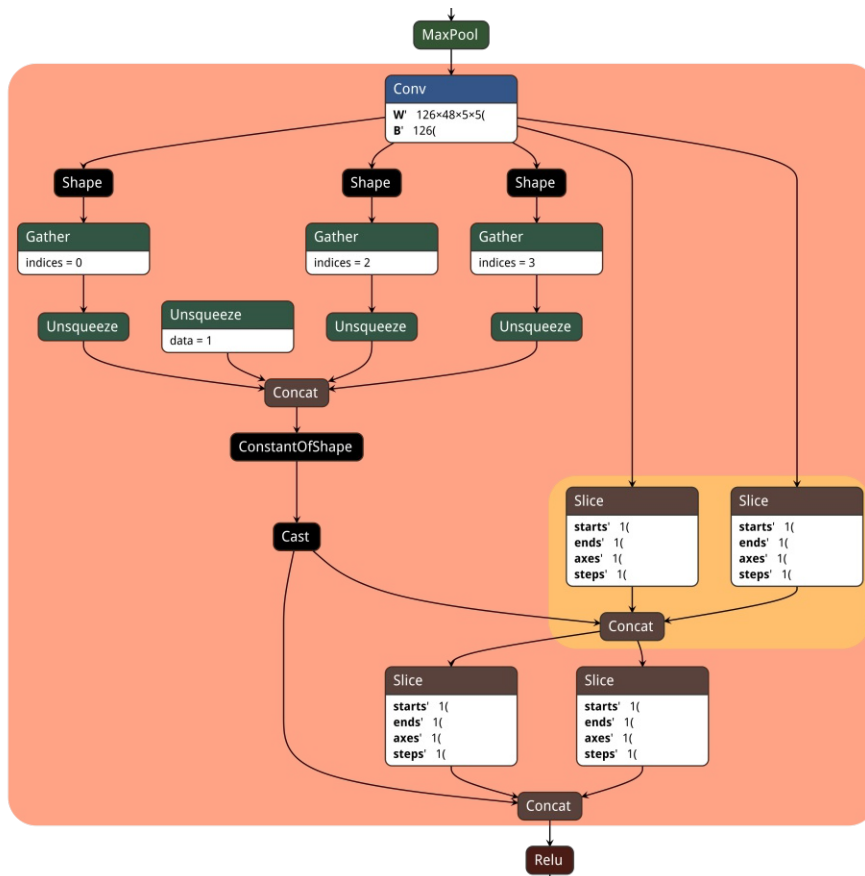


Figure 3.16: NeTuron network graph for the for loop implementation.

To cope with the first issue of being slow, the for loop was reworked. Instead of a loop, the benefit of using tensors was utilised. Meaning the implementation was changed to only use PyTorch tensors for indexing. This changed implementation is visible within Listing 3.1.

Listing 3.1: Zero-Padding implementation version 1

```
zero_tensor = torch.zeros(
    (o_tensor.shape[0], module.out_channels,
     o_tensor.shape[2], o_tensor.shape[3])
).to(o_tensor.device)
zero_tensor[:, module.keep_idxs, :, :] = o_tensor
return zero_tensor
```

This version creates in the first row the zero-tensor. The shape of this tensor is based on the original output shape in the dimension 0,2 and 3. The first dimension equals the batch size, the second and third correspond to the height and width of the original output feature map. The fifth row is the place that uses the torch indexing. For this to work, all the tensors used have to be on the same device and the keep_idxs variable has to be a torch tensor.

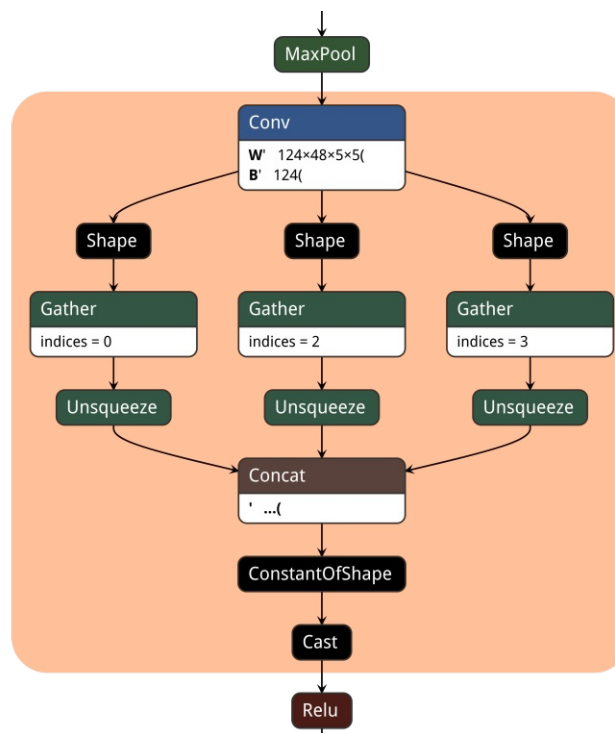


Figure 3.17: Netron network graph for the second implementation.

3.4.3 ONNX-Implementation

The downside of the implementation shown in the previous Section is that Open Neural Network Exchange (ONNX)¹ does not support the torch slicing notation. But the ONNX exporter does not create a warning or anything similar. Instead, a ONNX graph is exported which is also usable for computation. The downside is that the graph just continues with the created zero tensor and neglects all computations which happened beforehand. To showcase this an image of the disconnected graph was created using Netron [35]. The disconnected graph is visible within Figure 3.17. As can

¹ <https://onnx.ai>

be seen, the data coming from the convolution layer is fed into three shape blocks which extract the shape of the OFM. With a gather block at each edge one value of the shape is taken, then concatenated and fed into a constant shape to create the zero feature map. There does not exist a data connection from the output of the convolution to the succeeding blocks.

The list of compatible functions [1] shows multiple possible functions to use. For the third implementation, the compatible `index_copy_` function is used. The implementation in code looks exactly as in Listing 3.1, except instead of the torch indexing the `index_copy_` function is called.

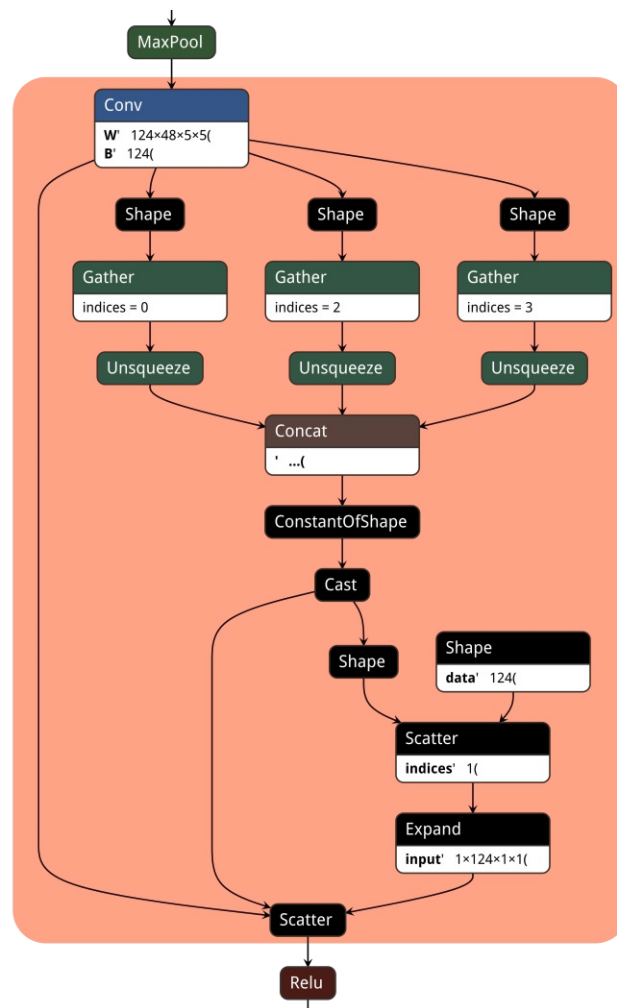


Figure 3.18: Netron network graph for the third implementation.

As can be seen within Figure 3.18 there is now an existing path from the output data to the scatter operation. The scatter operation is responsible for creating the output. This is done by taking in a tensor block and updating the contents. The update locations are a second input to the block. This input holds the indices which should be placed into the output block. These indices are used in conjunction with the third input, which is another tensor. The output data, therefore, has the same shape as the input data.

3.4.4 TRT-Implementation

The approach described within the previous Section works fine for PyTorch and ONNX. But according to the NVIDIA TensorRT documentation ², the TensorRT parser does not support the scatter operation, thus rendering this approach

² <https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html#supported-ops>

useless. To deal with this problem the subset of supported operations from the TensorRT parser was searched within the ONNX operators. Then a comparison was conducted which PyTorch functions translate too which ONNX operators to achieve a working implementation. After these steps, the implementation was changed to be usable with the TensorRT parser. For this, the implementation was changed according to Listing 3.2.

Listing 3.2: TRT compatible zero-padding implementation

```
zero_tensor = torch.zeros(
    (o_tensor.shape[0], 1, o_tensor.shape[2], o_tensor.shape[3])
).to(o_tensor.device)
zero_tensor = torch.cat((zero_tensor, o_tensor), 1)
zero_tensor = torch.index_select(zero_tensor, 1, module.keep_idxs)
return zero_tensor
```

This function is similar to Listing 3.1 with creating a zero tensor. A difference is, that the created tensor is only a single channel wide neglecting the batch size. This zero tensor is then concatenated onto the original OFM. The correct output feature map is then recreated by using the `index_select` function. This function returns a newly created tensor in which each channel corresponds to a channel from the original tensor. The channels which are selected are the non-masked channels from masked convolution. Noteworthy is the possibility of using the same channel multiple times, therefore creating the possibility of duplicating channels. This is used to fill the OFM with zero channels.

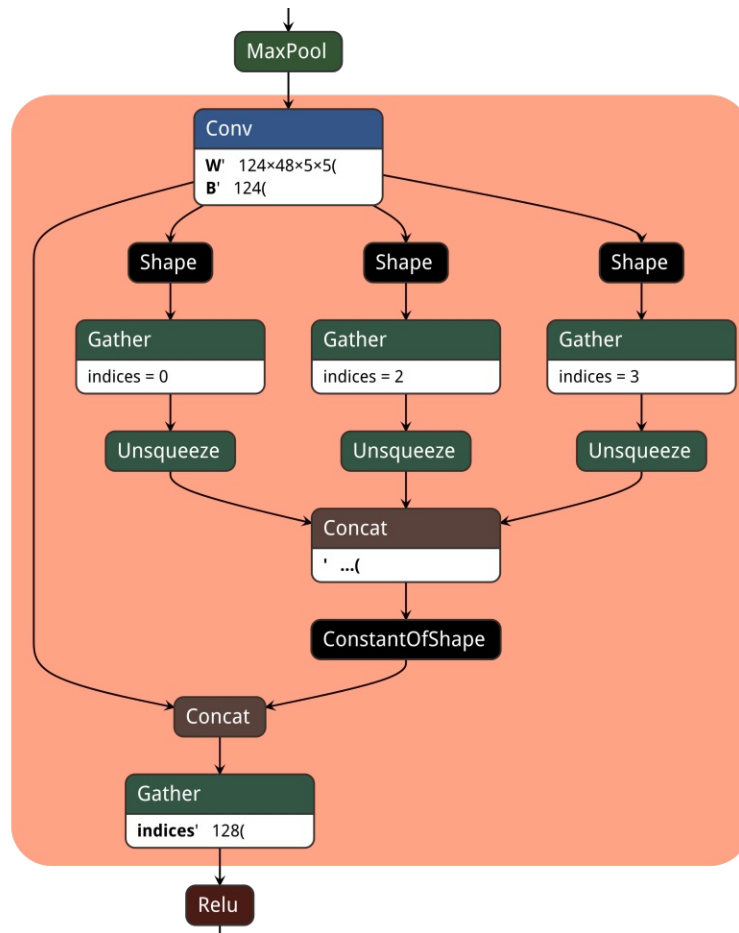


Figure 3.19: TensorRT compatible zero-padding network graph.

The created network graph can be seen within Figure 3.19. The functionality within this implementation is similar to

Figure 3.16. The main differences are an existing data connection that gets concatenated with the zero channel tensor. From this concatenated tensor the corresponding channels are extracted respectively duplicated in case of the zero channel. This part is done by the gather block within the network.

Furthermore, a noteworthy mention is, that the TensorRT network parser detected the broken data path which was created due to the implementation from Section 3.4.2. Even though the warning itself was not enough to determine the root cause, it helped in further pinning down the problem.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 4

Evaluation

To show the usability and applicability of zero-padding as well as the pruning algorithm this chapter is split into multiple sections. Within the first section, the program flow and measurement setup are explained. The second section holds measurements regarding the applicability and effects of zero-padding. The third section holds concrete measurements using networks trained on Cifar10 and ImageNet.

4.1 Measurement Setup

For the evaluation, an embedded platform, namely the Jetson Xavier AGX, was used. The Jetson Xavier was invented by NVIDIA and is the successor of the Jetson TX2 platform. NVIDIA is known for their GPUs and their advances in improving machine learning methods and speed up calculations. For this purpose, they introduced the tensor cores to significantly speed up the whole calculation process. These tensor cores are built with hardware support for these type of calculations. NVIDIA also invented the Jetson platforms with embedded machine learning in mind. These are specialised systems that utilise GPUs with tensor cores to improve throughput.



Figure 4.1: Jetson Xavier AGX from NVIDIA

The Jetson platform furthermore is assembled with power measurement sensors to monitor the power consumption of different parts for the hardware. Furthermore different power modes, which are called `nvpmodels` [6], are supported and available. In the basic configuration, the Xavier AGX offers up to 8 different modes. These modes change the maximum

clock frequency of the CPU, GPU and the eMMC. Furthermore, they can determine how many CPU cores are activated and available for computation. These offer a finer choice between speed and power consumption.

The Linux4Tegra or L4T shipped with the platform offers reading the power consumption via the usage of Linux device files. The original sensor is an IIC-sensor. A python program that logs these sensors and is started as a separate process was created to create the readings for the power consumption. The inference speed was measured by using the python time module.

All models are given as onnx model files. These are executed using the ONNX execution provider present within the onnxruntime. This runtime is utilised for the execution on the CPU and GPU. For execution utilising TensorRT the TensorRT-parser is used to parse the onnx description into an NVIDIA TensorRT engine file. The data is loaded in all cases using PyTorch data loaders and the corresponding functionality these present.

To further specify the exact hardware and software used look in Table 4.1.

Table 4.1: Used Hardware and Software on the Jetson Xavier AGX

Hardware	
Jetson Xavier AGX 16GB	
Software	
NSight Systems	2020.2.3
Jetpack	4.4
CUDA	10.2
TensorRT	7
Python	3.6.9
PyTorch	1.3.1

4.2 Test Network Evaluation

As a first step, the impact of zero-padding onto the network was measured. This was done by creating sample networks which are then pruned, measured and evaluated. This step was done to measure the impact of zero-padding on network performance. The tested implementations are described within Section 3.4 regarding their differences on an implementation level.

4.2.1 Small Test Network

The showcasing network is described within Table 4.2. This test network consists of 2 convolution layer, whereas each of them is pooled and put into a ReLU activation function. The convolutions utilise a 5×5 filter kernel. The inputs are from the Cifar10 dataset, therefore the input shape is $32 \times 32 \times 3$.

Inference Speed

The effect of zero-padding onto these small test networks regarding the inference speed is depicted within Figures 4.2-4.4. Each of these figures shows the number of removed filters along the x-axis with no filters removed on the leftmost and 126 filters removed on the right side. This leaves the test network with only 2 filters remaining. The y-axis is showing the recorded inference speed in milliseconds.

Table 4.2: Test Network

Layer Type	
Conv2d	$48 \times 3 \times 5 \times 5$
MaxPool2d	2×2
Conv2d	$128 \times 48 \times 5 \times 5$
MaxPool2d	2×2
FullyConnected	120×3200
FullyConnected	84×120
FullyConnected	10×84

Within Figure 4.2 the test network is executed with the 4 different available implementations on the CPU. As expected in Section 3.4, the implementation utilising the for loop is slowest and also scaling worst of the 4 implementations. Whereas the other three become faster the more filters are removed, the first version becomes slower. Furthermore of interest is, the sudden speedup in inference time. This speedup happens in all configurations as soon as 28 filters are removed.

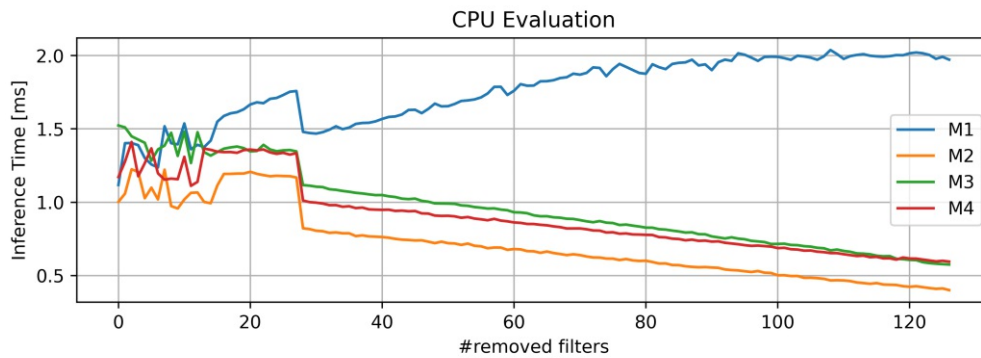


Figure 4.2: Testnetwork evaluation executed on the CPU.

The for-loop version scales even worse on the GPU execution as can be seen within Figure 4.3. Within this figure, two graphs are visible which show the same data. The first graph shows how bad the first version is scaling whereas the second version is zoomed in to show the difference between the versions. Due to the disconnecting nature of method M2, these networks execute considerably faster. The implementation M4 performs slightly worse than the implementation M3. All in all these two methods show no difference in inference time regarding the amount of pruned filters. This is due to the small mathematical complexity these networks hold.

The optimised execution using the TensorRT engine is shown within Figure 4.4. Due to only method M4 being compatible with TensorRT, the methods M3 and M4 on the GPU are used as a reference. The method M4 which was only slightly slower than M3 is speedup by $\approx 28\%$. Other than this additional speedup the behaviour is similar to the execution on the GPU.

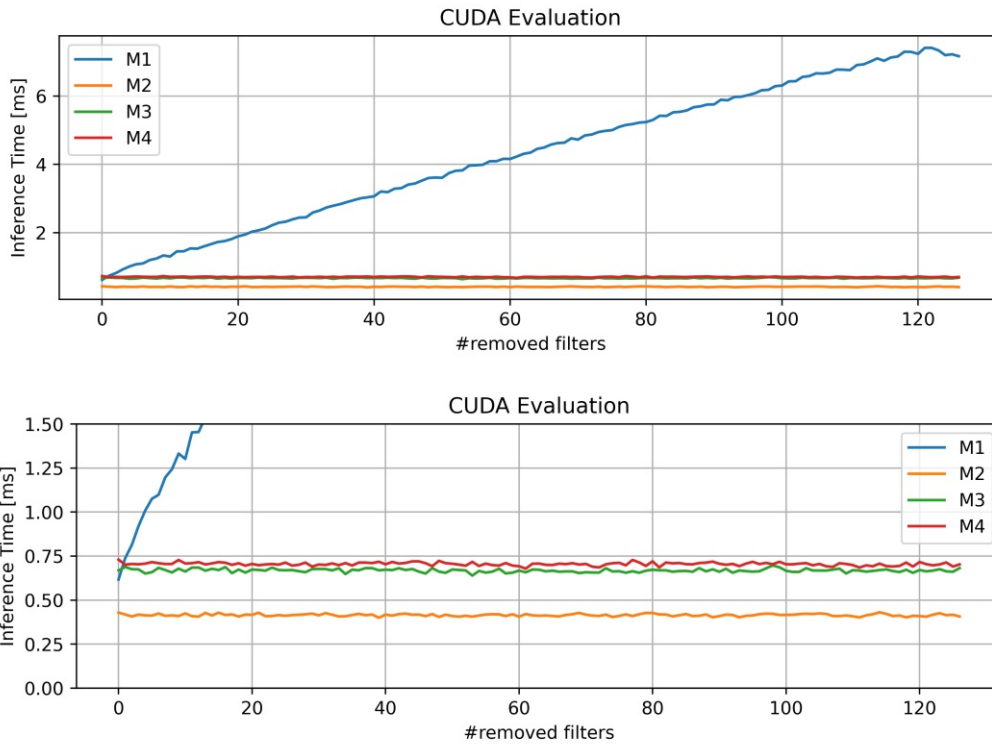


Figure 4.3: Testnetwork evaluation on the GPU. Figure b is zoomed.

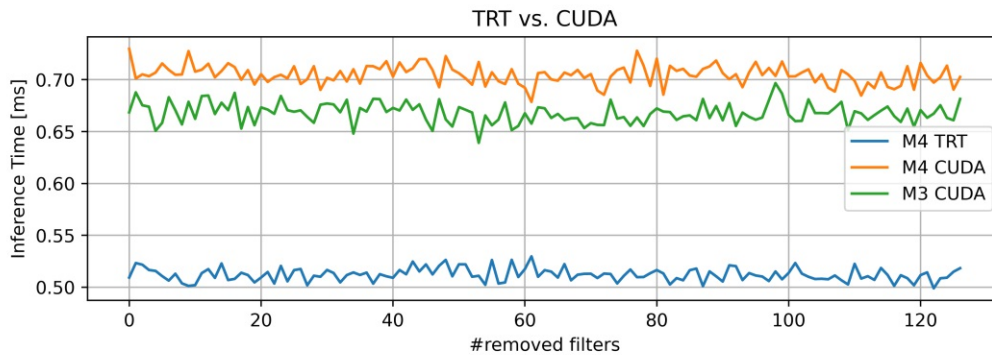


Figure 4.4: Evaluation zero-padding onto the test network. Comparison between CUDA and TRT execution providers.

Power Consumption

Even though the pruning shows no influence on the inference time except the CPU execution, there are trends visible in the power consumption. Within Figure 4.5 the different power consumptions are plotted. The values themselves are the mean value from all measurement points within an execution. Therefore they only serve as a rough outline. Furthermore, they also include all other computations done on the Xavier. The Xavier itself had no other processes running, but the OS is still working in the background and therefore skewing the values. The figure itself shows in the first column the mean power consumption on the CPU and on the second graph column the mean consumption on the GPU. Each graph row corresponds to one of the used execution providers, which are the CPU, the GPU and the TensorRT engine. Within each graph, the x-axis shows the number of removed filters ranging from 0 to 126. Along the y-axis, the mean power consumption as recorded during the execution is plotted in milliwatt.

As can be seen in the graph in the top left corner, the power usage goes down as the number of removed filters rises. The exception for this is the for loop version. This is only expected, due to directly requiring lesser calculations. In the case of the for-loop, the copying, slicing and back-copying overhead negate the effect of the removed calculations. Method M2 is in this case the method requiring the least power, but considering it disconnected the graph and just skipped on data copying altogether, this is not surprising. The power consumption for the GPU in the top right corner is just an anomaly. The GPU consumes power when used in the nvpmodel mode 0 with jetson clocks activated. As can be seen, the constant power consumption is around 1230 mW. The slight deviations in the case of Method M2 are deviations in the size of 6 mW and are therefore negligible.

The second row shows the power consumption when executed on the GPU using the Compute Unified Device Architecture (CUDA) execution provider. There are two notable phenomena within the CPU power consumption. The first is the decline in power consumption when executed with Method M1. This decline is due to the increase in inference time. Since the whole image preprocessing, which is, in this case, limited to image loading and transforming into a tensor, happens on the CPU. Due to the increase in inference time, the CPU has more downtime since lesser images are required in the same amount of time, therefore decreasing the power consumption. The second phenomena are the increased power consumption for Method M2. This increase is similar to the decrease in Method M1. Method M2 has a higher throughput, when neglecting the fact that it is unusable, and therefore requires the CPU to process more images. The case of executing on the GPU is also interesting. Method M1 has an increase in power consumption, most probably due to the increased time the GPU is handling the slicing and copying of the data for each iteration. The other 3 methods have similar behaviour. A step function is visible with a step size of 32. The first step happens at 32 removed filters, which corresponds to 96 filters remaining. The next step happens at 64 removed filters which correspond to 64 remaining filters. A further step is slightly indicated at method M2 around 96 removed filters, which correspond to 32 remaining filters. In method M3 and M4 is no such step visible, which is rooted in the fact that the data movement is covering this step up. From this plot it is therefore concluded that the minimum allocation size on the GPU regarding the convolution layer is 32 filters.

The third row showing the power consumption when executing the TensorRT engine. Since only method M4 is compatible with TensorRT no further comparison between the implementations is possible. Nonetheless, a trend similar to the GPU execution is visible in the GPU power consumption. There is no first step visible, but a clear step happens at 96 filters removed, which corresponds to 32 remaining filters. This would therefore back up the conclusion from the previous paragraph. The unsteady behaviour within the CPU power consumption might look like some phenomena, but compared to the other CPU consumption graphs, these spikes are well in the normal range. Therefore this is only due to the scaling and missing comparison possibilities.

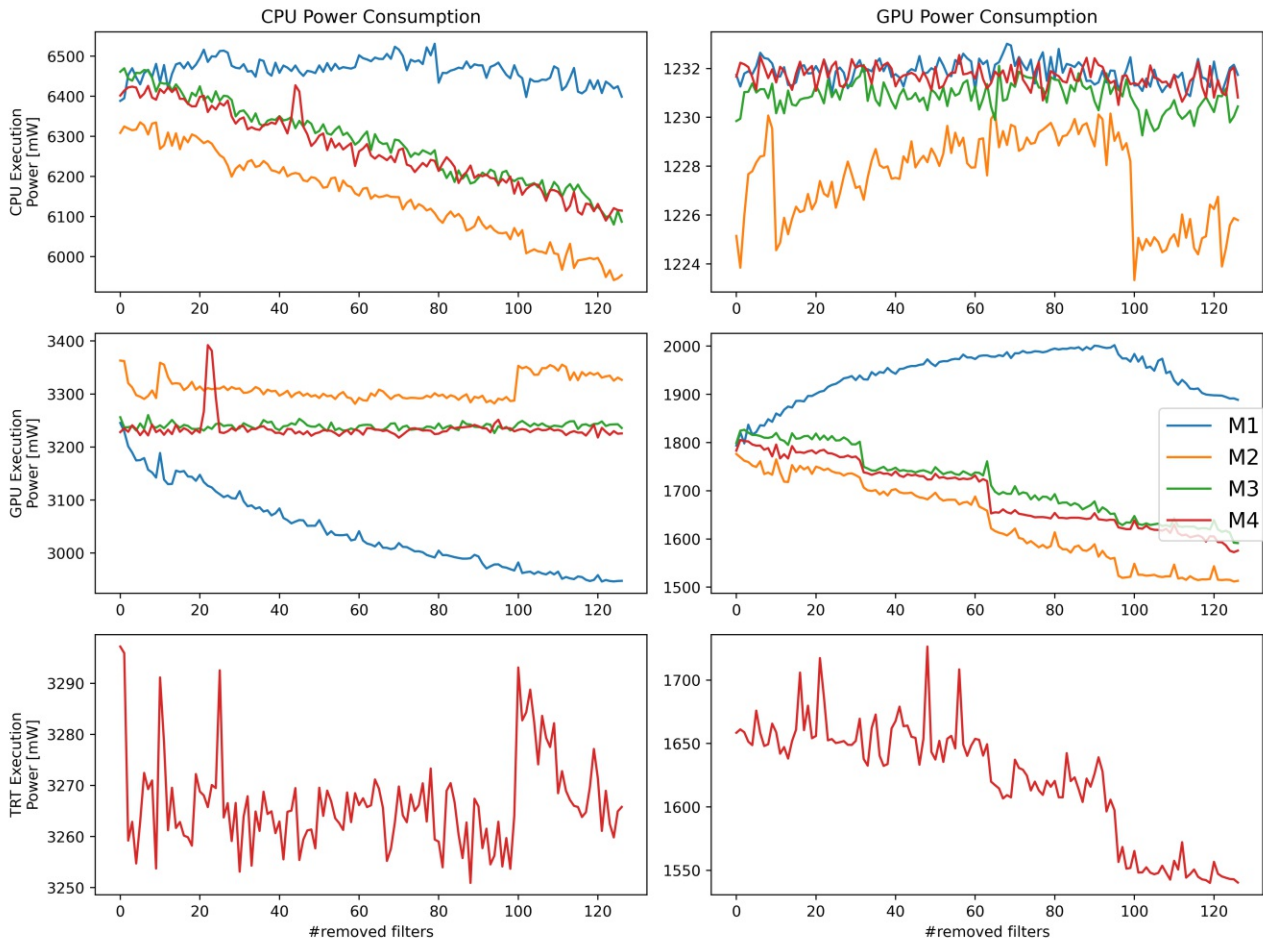


Figure 4.5: Plot for the testnetwork power consumption.

4.2.2 Big Test Network

As mentioned, there were no speedups regarding the inference time measurable with the small test network. To compensate for this, another test run with a bigger network with ImageNet input shape was created and tested. This bigger network mainly differs in the input image shape and the therefore adjusted linear layer. The exact network composition is shown in Table 4.3. Furthermore, the convolution layer was changed to work with 3 different filter sizes. Furthermore, now only the implementation method M4 is tested, which is described within Section 3.4.4. The equivalence between method M3 and M4 has been shown in the previous chapter. From these similarities, an equal behaviour is assumed going onward for bigger input shapes.

Table 4.3: Big Test Networks

Layer Type	Network 1	Network 2	Network 3
Conv2d	$48 \times 3 \times 3 \times 3$	$48 \times 3 \times 3 \times 3$	$48 \times 3 \times 3 \times 3$
MaxPool2d	2×2	2×2	2×2
Conv2d	$128 \times 48 \times 5 \times 5$	$128 \times 48 \times 3 \times 3$	$128 \times 48 \times 1 \times 1$
MaxPool2d	2×2	2×2	2×2
FullyConnected	120×359552	120×373248	120×387200
FullyConnected	84×120	84×120	84×120
FullyConnected	10×84	10×84	10×84

Inference Speed

As can be seen within Figure 4.6 the behaviour is identical to the smaller test network regarding the inference time. With each removed filter the required amount of computation shrinks and is replaced with cheaper data copying. Also in the computation heavy version, the decline is steady and without any anomalies, which are seen within Figure 4.2.

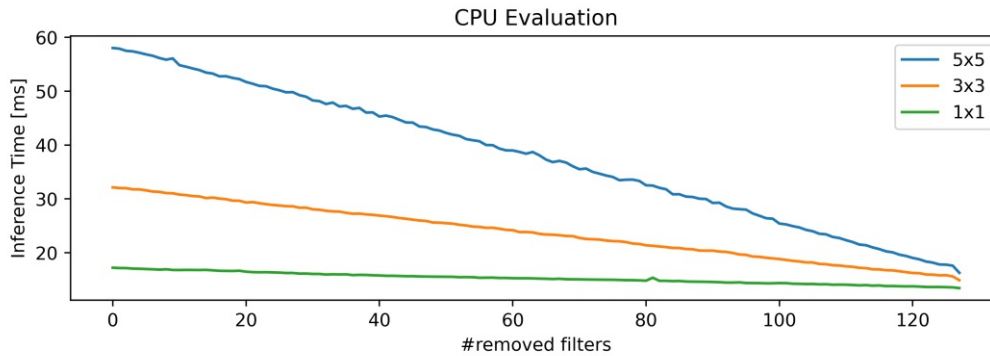


Figure 4.6: Zero-padding inference time evaluation using the CPU execution provider.

A rather interesting behaviour is visible within Figure 4.7. This behaviour is fitting to the assumption of an allocation size onto the GPU. Now that the execution takes longer, the effect of the pruning itself is also visible. Fitting to the assumed behaviour which is shown in Figure 4.5, now there are also steps visible. Most notable in this case is the effect on the 5×5 convolution which is the computation heaviest operation of the 3 available networks. These first two steps are also visible within the 3×3 convolution network. The third step might be too shallow to determine within the recorded data. The step size of 32 is determined to be equal to the step size determined with the power consumption on the smaller network. The spike within the 5×5 at 0 filters removed is due to a measurement run that had lags during the first iterations and is therefore skewing the mean value. For the 1×1 case a step is visible at 64 removed filters and a slight step is implicated at 96 removed filters. But the steps within this network are not visible due to the faster execution time.

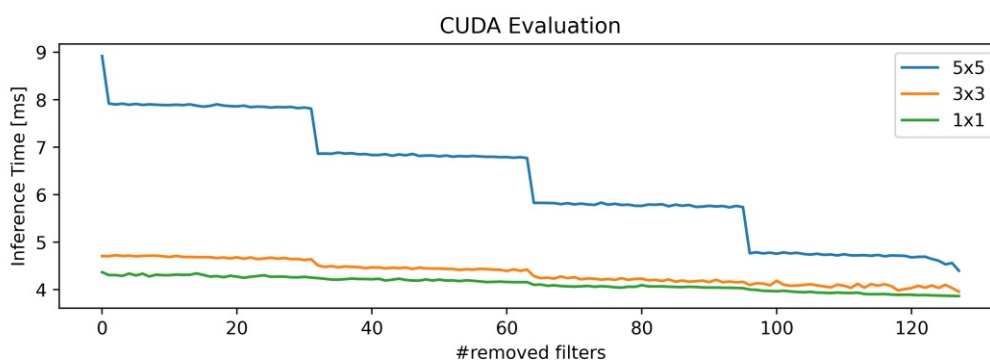


Figure 4.7: Zero-padding inference time evaluation using the CUDA execution provider.

The third test case which uses the TensorRT engine is showing the same behaviour as the GPU case. Within the TensorRT engine the execution time is again around $\approx 30\%$ faster than the CUDA engine. Also, the step size of 32 is observable within this network. These steps are visible within all 3 networks at the same positions as in the CUDA execution engine. The main difference between the TensorRT engine network is with the 5×5 network, which shows unusual behaviour at 28 remaining filters. After this point, the inference speed jumps from ≈ 3.8 ms to ≈ 4.18 ms in steps of 4

filters removed. This happens most probable due to a different mapping done by the TensorRT engine.

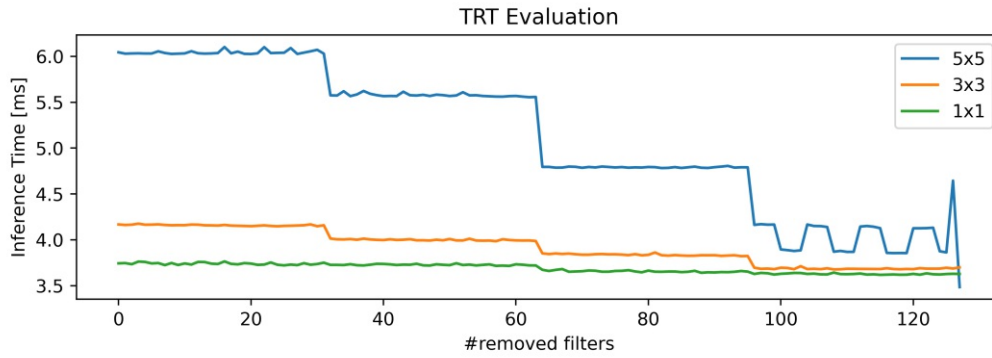


Figure 4.8: Zero-padding inference time evaluation using the CPU execution provider.

Power Consumption

Similar to Section 4.2.1 a decrease in the GPU power consumption when executed on the GPU is visible. But contrary to this the CPU power increases especially visible for the 5×5 case. This is explained via Figure 4.10. Due to the increased inference speed which is visible within the last section, the CPU has to do more preprocessing of the input images, therefore, resulting in a higher CPU utilisation and thus higher power consumption. This behaviour is similar to the TRT execution. A notable difference is within the GPU power consumption when comparing the GPU execution with the TRT execution. There is only one distinct step visible contrary to the GPU execution which has 4 steps with a step size of 32. The CPU power consumption increases fastest with the 5×5 execution. The reason for this more drastic increase is again the faster inference speed and therefore the requirement of more preprocessing.

Within Figure 4.10 the increase in power usage concerning pruning becomes visible. Within this image two combined NSight reports are visible. Within the top half of the image, a network profiling is visible for a network that has 10 filters and removed. On the lower half, a network is depicted with 100 filters removed. Both of these networks were executed within the CUDA execution provider. Within Table 4.4 the CPU utilisation for both tests is recorded. This utilisation is calculated to fit 100% within the profiled application. It becomes apparent from these values, that the 10 removed filters version is consuming more time within the main process and the data loading has more downtime. This behaviour is also depicted within Figure 4.10. Within this figure, only one of two data loading processes is shown. These normally run interleaved two minimise the need for waiting on data loading. The orange marked part within the top half of the image shows the execution of the network on the GPU. When comparing this to the lower half marked in blue the speedup is visible. The data loading is marked with green in both cases. From this, it becomes apparent that each data loading process consists of 3 threads, with two of them for signalling and one thread to deal with the bulk work of loading and preprocessing the image. In red the downtime areas are marked in both cases. When comparing the red areas in both cases, it becomes visible, that the 10 removed filters version has considerable downtime and is therefore reducing the overall power consumption. The 100 removed filter version has due to the faster inference speed lesser downtime and therefore accommodates to the steps visible within Figure 4.9.

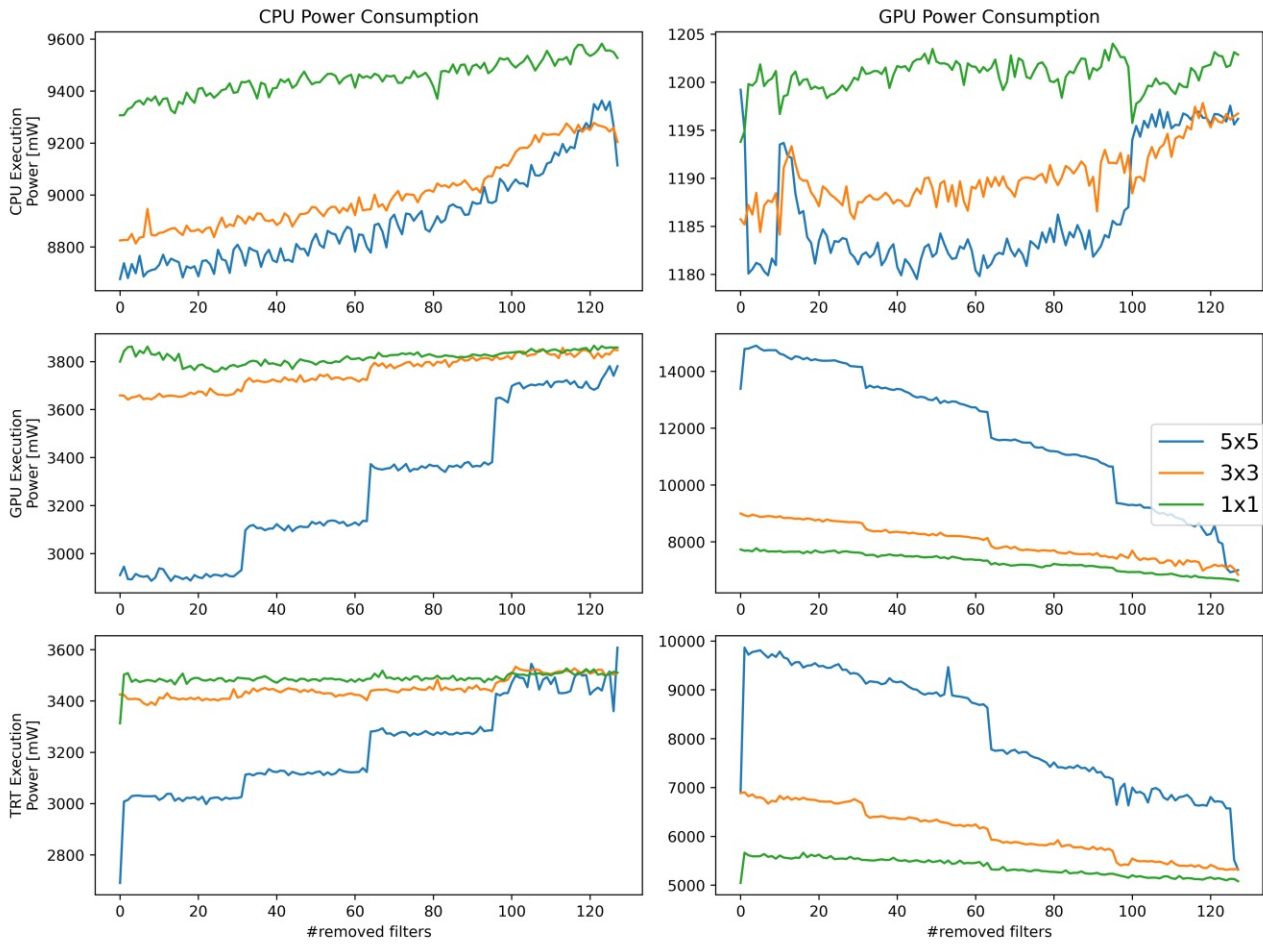


Figure 4.9: Big testnetwork power consumption across different execution providers.

Table 4.4: CPU utilisation

Process	Filters removed	
	10	100
Main	41.00%	31.95%
Dataloader 1	26.85%	30.65%
Dataloader 2	26.60%	30.26%
Profiling/Logging	5.50%	7.10%



Figure 4.10: NVIDIA Nsight Systems Report

4.3 Pruning Evaluation

The evaluation of the pruning workflow itself is primarily characterised by the total epoch count until the pruned model was achieved, the reached sparsity and model metadata such as the removed MACs and weights.

4.3.1 Cifar10

Multiple smaller networks were trained on the Cifar10 classification task. Then they were pruned and the effect of the pruning was recorded. These smaller networks were then executed on the Jetson Xavier on the CPU, the GPU and using TRT.

Models

All these models presented in Tab.4.5 were trained on Cifar10 and compressed using the presented approach.

Table 4.5: Model Metadata

Network	Accuracy [%]		MACs (%-Removed) [$\times 10^6$]		Parameters (%-Removed) [$\times 10^6$]	
	Original	Pruned	Original	Pruned	Original	Pruned
MobileNetV2	92.41	90.05	91.15	19.87 (78.20%)	2.26	0.29 (87.03%)
RegNetX 200MF	95.18	89.70	223.31	131.62 (41.06%)	2.30	1.23 (46.25%)
SimpleDLA	94.74	92.44	913.49	324.14 (64.52%)	15.12	3.48 (76.95%)
DenseNet121	74.44	81.94	6.87	1.75 (74.52%)	888.35	440.01 (50.47%)

ResNet50

To evaluate the effect of the pruning, a ResNet50 was pruned. Due to the pruning flow resulting in multiple possible networks with different pruning steps, all of these were further evaluated and recorded. ResNets mainly consist of residual blocks which do make pruning complicated. All these networks utilise zero-padding. Within Figure 4.11 the effect of the pruning on the accuracy, the number of parameters and the MACs are plotted. And within Table 4.6 all the values are also recorded. As can be seen, the first pruning step influences the network little in terms of accuracy, while already reducing the parameters by $\approx 27\%$ and the MACs by $\approx 20\%$.

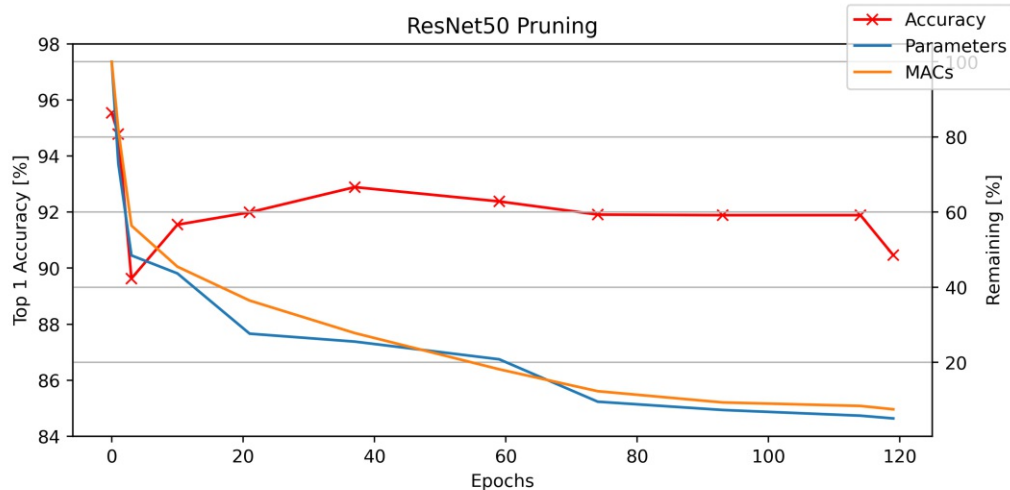


Figure 4.11: Pruningflow executed on a ResNet50

Table 4.6: ResNet50 Pruning Steps

Epochs	Accuracy [%]	Parameters [$\times 10^6$]	MACs [$\times 10^6$]
0	95.54	23.46 (00.00%)	1297.82 (00.00%)
1	94.78	17.11 (27.08%)	1063.38 (18.06%)
3	89.63	11.37 (51.52%)	732.08 (43.59%)
10	91.55	10.25 (56.31%)	590.20 (54.52%)
21	91.99	6.49 (72.34%)	473.30 (63.53%)
37	92.89	5.99 (74.45%)	361.09 (72.18%)
59	92.38	4.89 (79.16%)	235.98 (81.82%)
74	91.91	2.24 (90.44%)	160.24 (87.65%)
93	91.89	1.72 (92.64%)	121.52 (90.64%)
114	91.89	1.36 (94.16%)	109.39 (91.57%)
119	90.46	1.19 (94.89%)	97.81 (92.46%)

DenseNet121

DenseNet first introduced by [20] builds upon observations made by previous researchers and introduces so-called Denseblocks. Within such a block the output is directly forwarded to the next and all subsequent blocks within this dense block. Within a block consisting of 6 layers the sixth layer has 5 concatenated input feature maps, from layers 1 to 5 within the dense block. This helps to further alleviate vanishing gradient problems. Due to this structure DenseNets are a prime example of dependencies within the network structure.

As can be seen within Figure 4.12, the evaluation speeds up by 73% in the case of execution on the CPU and 29% when executing using CUDA. Even though the inference is sped up in these modes, when using the TensorRT optimisation, the inference gets slowed down by approximately 3%. Even though these are considerable speedups within the first two modes, regarding the power consumption, depicted within Figure 4.13 no improvement is observable regarding the first two execution modes. In the case of the TRT optimised execution, the power consumption is worsened by $\approx 10\%$ from 3001 mW consumption within the original network to 3275 mW within the thinned network.

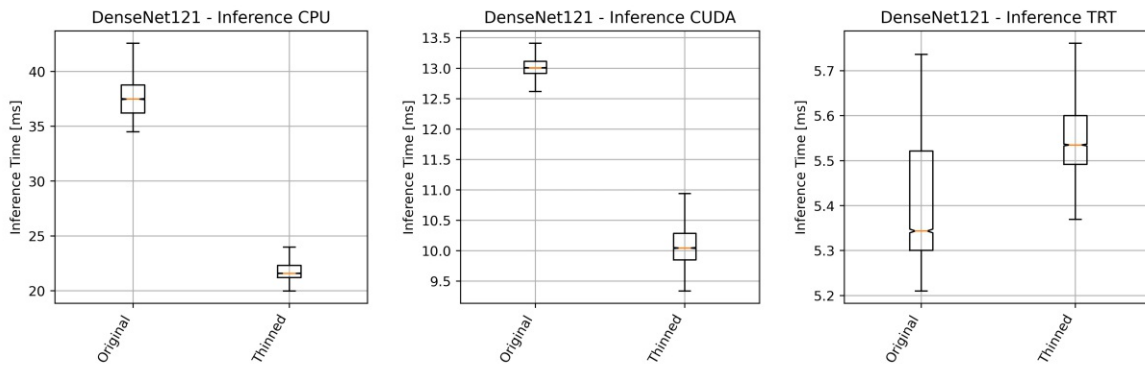


Figure 4.12: Inference measurements for DenseNet121

RegNetX

A network taken from [33] was also taken for measurements. These networks, also called RegNets, consist of simple and low-dimensional networks. Since they are built only with defined base blocks, these networks are used to explore the design space and the influence and interplay of these and how they contribute to the total accuracy. Furthermore is RegNet a network, which is hard to prune due to it prominently consisting of grouped convolutions. This problem is circumvented by using the aforementioned zero-padding.

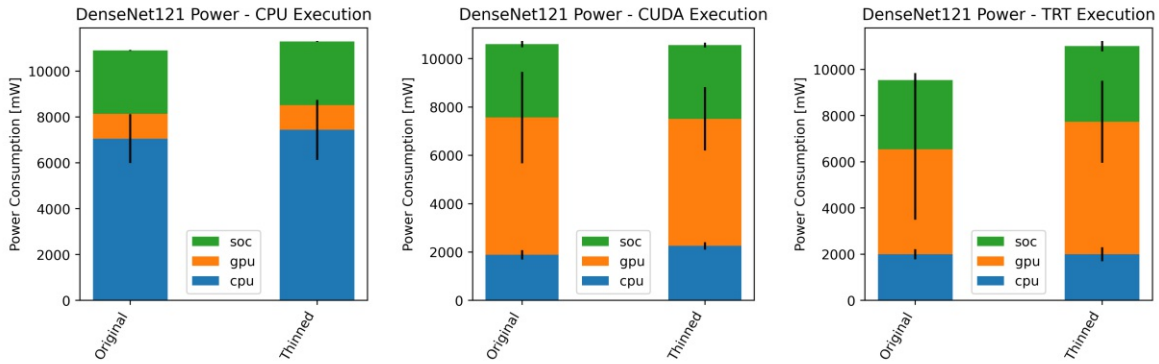


Figure 4.13: Power measurements for DenseNet121

Unlike with DenseNet, the speedup considering inference is only limited to the execution on the CPU, as can be seen within Figure 4.14. In the case of execution on CUDA, the pruned variant performs about 0.4 ms worse than the original version. Although worse the deviation regarding the inference speed is smaller than the original. When comparing the two TensorRT optimised versions, the original version is approximately 0.6 ms faster than the thinned version. The power consumption is unfazed regarding the pruning. A small improvement for the thinned version over the original version can be seen for the execution on CUDA within Figure 4.15 and a worsening for TRT, but these are minimal.

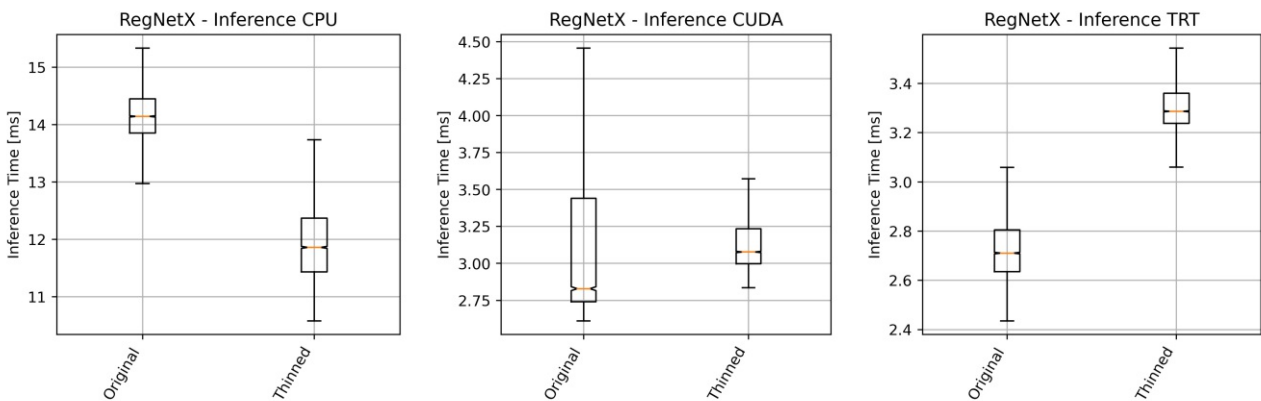


Figure 4.14: Inference measurements for RegNetX

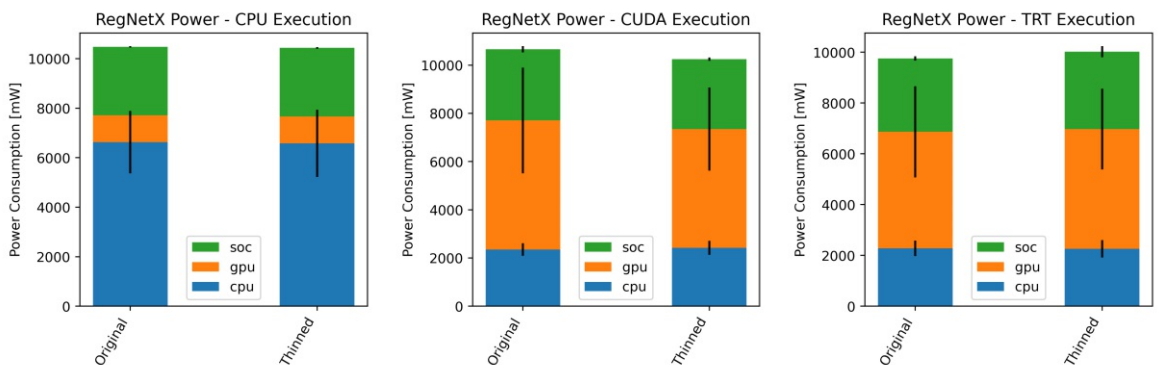


Figure 4.15: Power measurements for RegNetX

SimpleDLA

Deep Layer Aggregation [45] aims to optimise the Feature Map (FM) aggregation within networks. This is done to more optimal extract the information from a network. By changing the implementation of shortcut connections to also take into consideration how shallow the connection is and or reordering these shortcut connections an improvement in recognition is achieved. As these networks also create unusual implementations e.g. residual connections are also ideal to evaluate pruning.

As can be seen within Figure 4.16, an improvement in all cases is achieved. These improvements range from 117% in case of executing on the CPU, over 30% when execution on CUDA to 6% when using the TensorRT optimised network. When also considering the power consumption, then there are no improvements regarding the execution on the CPU. On the other hand, there are improvements when considering the execution on the GPU and TensorRT.

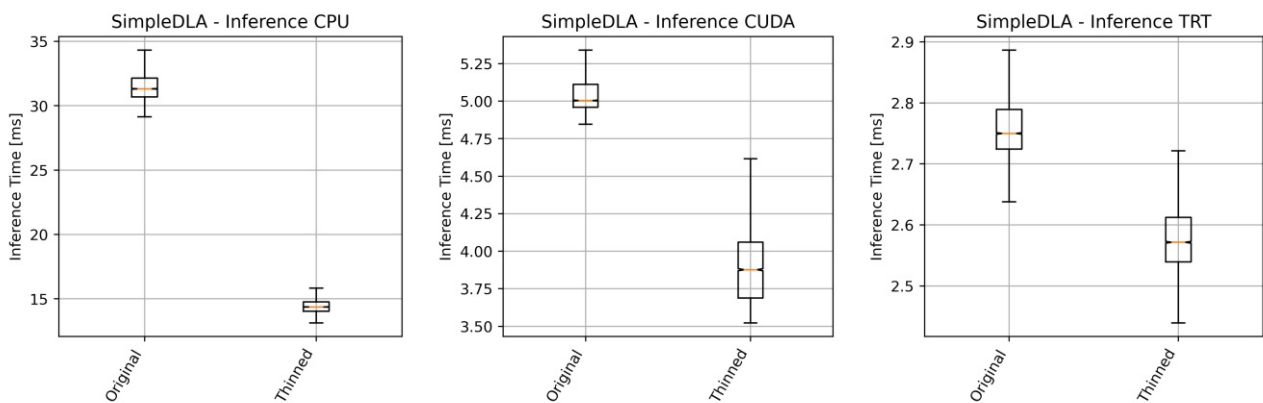


Figure 4.16: Inference measurements for SimpleDLA

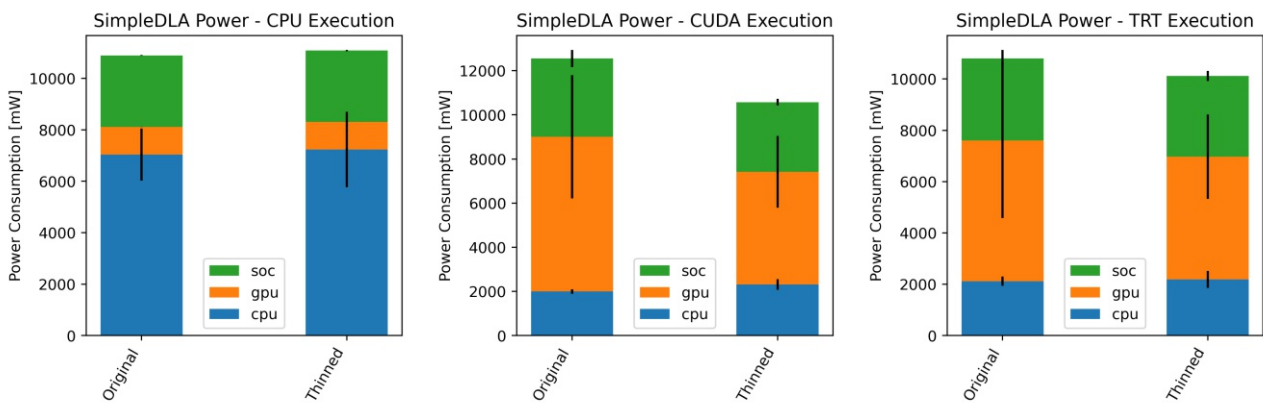


Figure 4.17: Power measurements for SimpleDLA

4.3.2 ImageNet

A trained MobileNetV2 which was originally taken from [25] has been pruned to show the positive effect of pruning onto the network. By pruning, about 25% of the MACs within the network were removed. Therefore the network does correspond to an original MobileNetV2 with an 0.75 depth multiplier. This can be seen in Table 4.7. Furthermore is notable that the total amount of parameters stays roughly the same. MobileNetV2 is consisting of residual blocks with the standard configuration of pointwise-convolution→groupwise-convolution→pointwise-convolution.

Table 4.7: Model Metadata

Name	Accuracy	MACs	Parameters	MACs Reduction	Parameter Reduction
MobileNetV2 1.0	72.19 / 90.53	300.79	3.504	-	-
MobileNetV2	69.86 / 89.06	227.03	3.103	25%	11%
MobileNetV2 0.75	69.88 / 88.98	209.08	2.636	30%	25%

The effect of the pruning with regards to the inference speed is depicted in Figure 4.18. In this figure are 5 networks shown. Three times MobileNetV2 with different depth multiplier and two times the pruned version. The pruned versions differ by one time utilising zero-padding and one time only classical pruning with avoidance of unprunable layers. As is to be expected, the inference speed when on the CPU is almost similar to the inference speed of the version using the 0.75 depth multiplier. This trend drifts apart when inspecting the version which is running on CUDA. Within this execution provider, the thinned version without zero-padding is slightly faster than the original version (3.9 ms vs. 4.07 ms). The zero-padded version is also slower than the 0.75 version (4.03 ms vs. 4.07 ms). This trend of drifting apart continues when using the TensorRT optimisation. Within MobileNet the optimiser can drastically improve the original networks, but the structure within the zero-padded version does break the number of possible layers to use layer fusion on. When inspecting the power within Figure 4.19, the decrease in MACs becomes visible within the GPU power consumption when comparing the 5 versions. Within Table 4.8 the reduction of power regarding the original 1.0 version in contrast to the pruned versions becomes visible. Furthermore, do the two pruned versions have higher power consumption when compared to the 0.75 version within the CPU execution provider.

Table 4.8: MobileNetV2 Power Measurements

Execution Provider	Network	Consumption	
		CPU [mW]	GPU [mW]
cpu	0.5	7879.74	1079.19
	0.75	7772.45	1079.48
	1.0	7737.41	1079.75
	No padding	7810.2	1079.2
	Zero-padded	7807.64	1079.23
cuda	0.5	4138.7	5261.05
	0.75	3939.57	7264.97
	1.0	3767.0	8172.14
	No padding	3982.16	7307.54
	Zero-padded	4040.05	7174.37
trt	0.5	4160.66	3777.19
	0.75	4031.81	4947.02
	1.0	3962.68	5609.76
	No padding	3954.27	4871.9
	Zero-padded	3933.77	4979.17

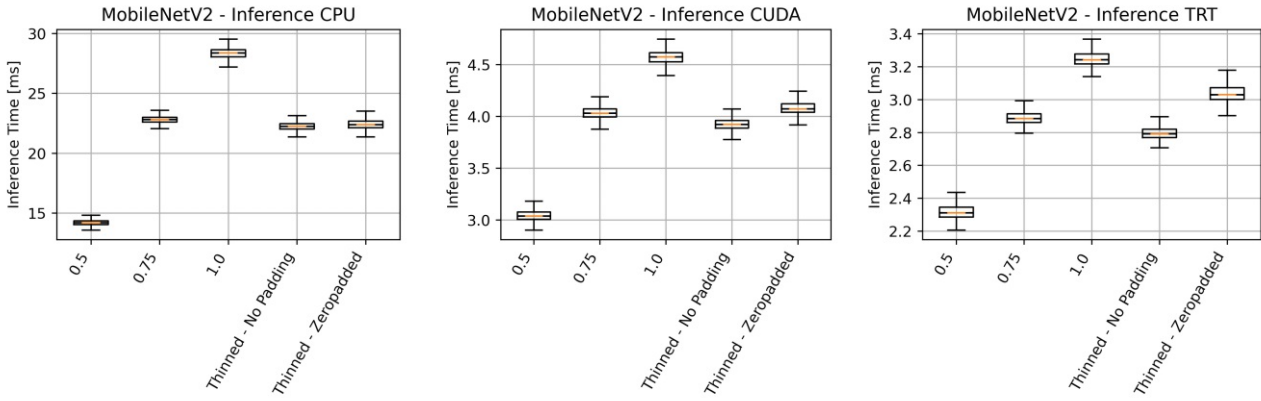


Figure 4.18: Inference measurements for MobileNetV2

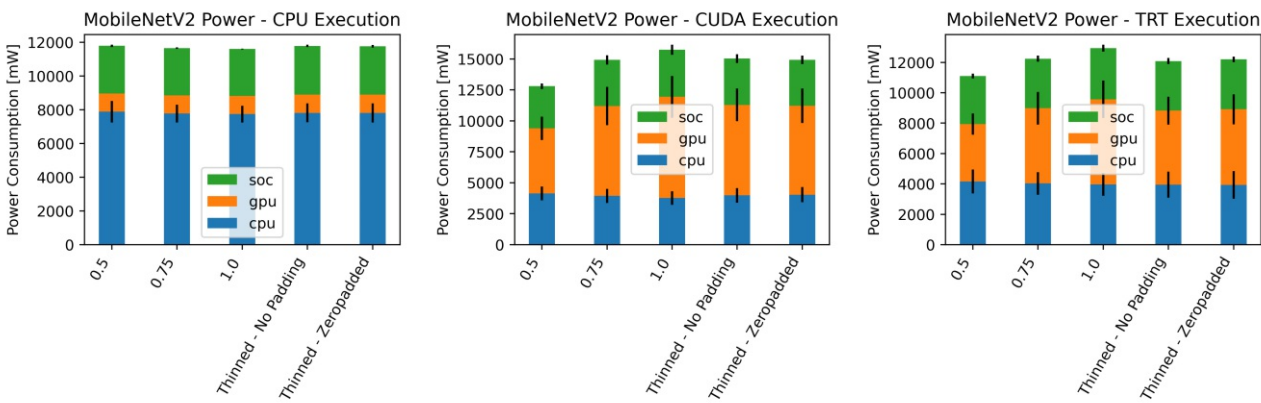


Figure 4.19: Power measurements for MobileNetV2

4.4 Pruning together with ONNX Simplifier

During this work, the onnx-simplifier [8] was also used to test the effectiveness on networks. The usefulness ranges from improving the inference time, having no impact on the inference time or even worsening it. The cases of no impact or worsening happened mostly for the execution on the CPU or the CUDA execution provider. An example of worsening the execution time is depicted within Figure 4.20. The exact reason for this worsening was not further researched.

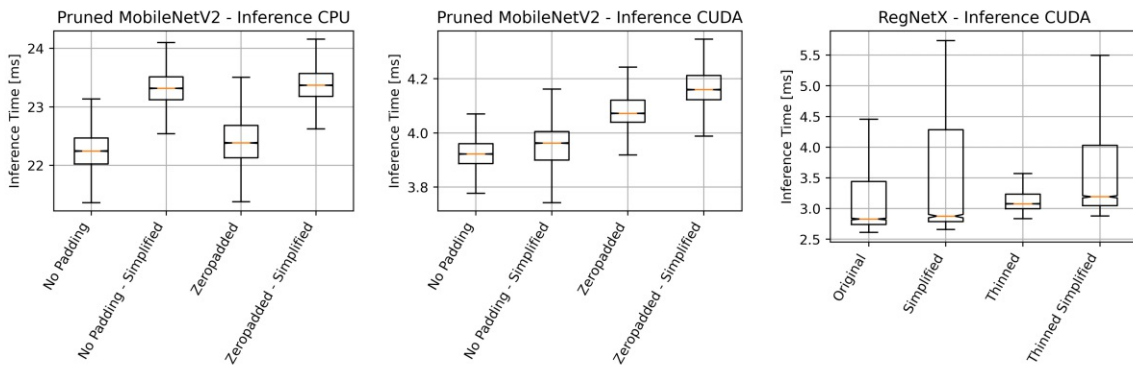


Figure 4.20: Example of onnx-simplifier worsening inference time.

In the case of execution using TensorRT, the onnx-simplifier improved the performance in close to all cases. There was only a single case where no improvement was achieved, but in this case, the simplifier did not change anything.

An example of the improvement is shown within Figure 4.21. In the case of DenseNet121, the simplifier improved the thinned network and the result was better than the original TRT version even though the thinned version performs worse. In the case of MobileNetV2 the simplifier improved both versions, but the thinned version still performs worse than the version without zero-padding and only thinning removable sections.

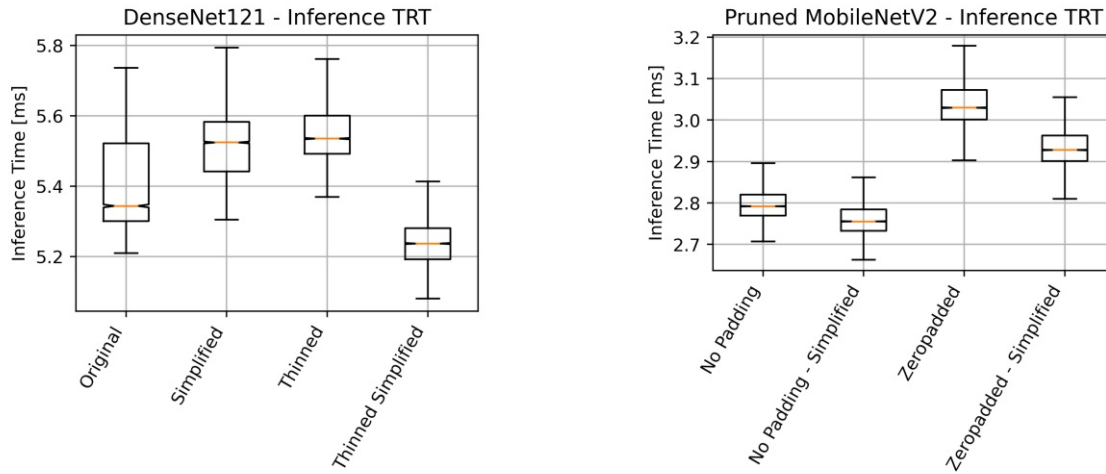


Figure 4.21: Example of onnx-simplifier improving inference time.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The goal of this work was an evaluation regarding the usability of pruning for a Jetson Xavier AGX and the implemented pruning workflow. As has been shown, pruning has a positive effect regarding the inference time when being executed purely on the CPU or the GPU. Furthermore, a decrease in power consumption is also observable. These effects do change when comparing the effects with the usage of the TensorRT optimiser from NVIDIA. Due to the effect of zero-padding onto the network, the possible optimisations locations for TensorRT gets reduced. The result, therefore, vary widely depending on the network and the possible optimisation locations TensorRT has.

Concerning power consumption, pruning has shown a positive effect in reducing the power consumption across all tested networks when executed with CUDA. When comparing the power consumption when being executed on the CPU, the power consumption in most cases is unfazed. This is mainly due to the increase in power consumption of the CPU when decreasing the inference time.

Finally, this work shows the implementation of zero-padding. Zero-padding shows to be a successful approach in increasing prunability. This is shown with the evaluation of the test networks and the trend is also visible in the networks. Especially mentionable is the RegNet pruning. This network is consisting mainly of blocks represented by pointwise- and grouped convolutions. This combination is not prunable due to the pointwise convolutions propagating the dependency and grouped convolutions being unprunable. With zero-padding, pruning was enabled and a speedup achieved in both cases the CPU and the GPU execution. As mentioned before, the zero-padding does take away possibilities for optimisation which the TensorRT optimiser would otherwise utilise and leverage on them. This downside is visible in both the inference time and the power consumption.

The conclusion is that the effect of pruning is highly dependant on the network. Regarding zero-padding is yielding a positive effect on both the execution on the CPU and the GPU. Due to it changing the network into an unusual pattern it can hinder the TensorRT optimiser. Therefore it has usability, but it would be wrong to say it is a better option.

5.2 Future Work

This work shows an iterative pruning approach based on measurement data. It also shows that a speedup for a measurement-based approach is easily reachable by decreasing the measurement data, but it lacks in some aspects. For further usability, the dependency on Distiller is rather big and it also ties the PyTorch version to 1.3.1. This is something hindering further usage, due to improvements that get built into PyTorch are out of reach. Circumventing this would be possible by adapting the PyTorch built-in pruning since version 1.4 which also relies on masking. To work with the thinning part of the whole workflow, the tracer which is used for data graph generation needs to be extracted from Distiller.

A further big improvement which would be beneficial is the usage of hardware information. This would incorporate an estimator to the pruning workflow within the Prune Best Layer block in Figure 3.13. Rules would need to be defined, to find optimal spots on the target hardware, therefore, leveraging the pruning to hardware aware pruning. Hand in hand with the implementation of an estimator would be the reformulation of the pruning as an optimisation problem finding the sweet spots regarding the accuracy loss, the ideal layer to prune, the ideal position within a layer and the decrease in MACs.

Bibliography

- [1] J. Bai, F. Lu, K. Zhang, et al. *ONNX Operators*. URL: <https://github.com/onnx/onnx/blob/master/docs/Operators.md>.
- [2] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020.
- [3] F. Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: *arXiv:1610.02357 [cs]* (Apr. 4, 2017). (Visited on 01/14/2021).
- [4] T. Contributors. *PyTorch Documentation*. 2020. URL: <https://pytorch.org/docs/stable/index.html>.
- [5] T. Contributors. *PyTorch Source Code*. 2020. URL: <https://github.com/pytorch/pytorch/>.
- [6] N. Corporation. *NVIDIA Linux for Tegra Documentation*. 2021. URL: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%5C%2520Linux%5C%2520Driver%5C%2520Package%5C%2520Development%5C%2520Guide/power_management_jetson_xavier.html#wpID0EOYFOHA.
- [7] Y. L. Cun, J. S. Denker, and S. A. Solla. “Optimal Brain Damage”. In: *Advances in Neural Information Processing Systems 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 598–605.
- [8] daquexian. *onnx-simplifier*. Version 0.3.3. Feb. 27, 2021.
- [9] S. Elkerdawy, M. Elhoushi, A. Singh, H. Zhang, and N. Ray. “To filter prune, or to layer prune, that is the question”. In: *arXiv:2007.05667 [cs]* (July 10, 2020). (Visited on 09/24/2020).
- [10] J. Guo, S. Lu, H. Cai, W. Zhang, Y. Yu, and J. Wang. “Long Text Generation via Adversarial Training with Leaked Information”. In: *CoRR abs/1709.08624* (2017).
- [11] Y. Guo, Y. Li, R. Feris, L. Wang, and T. Rosing. “Depthwise Convolution is All You Need for Learning Multiple Visual Domains”. In: *arXiv:1902.00927 [cs]* (Feb. 19, 2019). (Visited on 10/21/2020).
- [12] S. Han, J. Pool, J. Tran, and W. J. Dally. “Learning both Weights and Connections for Efficient Neural Networks”. In: *arXiv:1506.02626 [cs]* (Oct. 30, 2015). (Visited on 05/01/2020).
- [13] B. Hassibi, D. G. Stork, and G. J. Wolff. “Optimal Brain Surgeon and general network pruning”. In: *IEEE International Conference on Neural Networks*. 1993, 293–299 vol.1.
- [14] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: *arXiv:1512.03385 [cs]* (Dec. 10, 2015). (Visited on 10/29/2020).
- [15] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang. “Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks”. In: *arXiv:1808.06866 [cs]* (Aug. 21, 2018). (Visited on 03/18/2021).
- [16] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. “AMC: AutoML for Model Compression and Acceleration on Mobile Devices”. In: *arXiv:1802.03494 [cs]* (Jan. 15, 2019). (Visited on 02/17/2021).

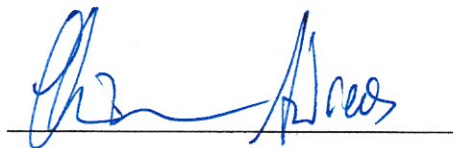
- [17] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. “Searching for MobileNetV3”. In: *arXiv:1905.02244 [cs]* (Nov. 20, 2019). (Visited on 10/21/2020).
- [18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *arXiv:1704.04861 [cs]* (Apr. 16, 2017). (Visited on 10/21/2020).
- [19] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang. “Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures”. In: *arXiv:1607.03250 [cs]* (July 12, 2016). (Visited on 02/23/2021).
- [20] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. “Densely Connected Convolutional Networks”. In: *arXiv:1608.06993 [cs]* (Jan. 28, 2018). (Visited on 10/29/2020).
- [21] A. Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: 2009.
- [22] 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.
- [23] Y. LeCun, J. Denker, and S. Solla. “Optimal Brain Damage”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1990.
- [24] N. Lee, T. Ajanthan, and P. H. S. Torr. “SNIP: Single-shot Network Pruning based on Connection Sensitivity”. In: *arXiv:1810.02340 [cs]* (Feb. 23, 2019). (Visited on 03/18/2021).
- [25] D. Li, A. Zhou, and A. Yao. “HBONet: Harmonious Bottleneck on Two Orthogonal Dimensions”. In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2019.
- [26] F.-F. Li, A. Karpathy, and J. Johnson. *CS231n: Convolutional Neural Networks for Visual Recognition 2016*. 2019. URL: <http://cs231n.stanford.edu/>.
- [27] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. “Pruning Filters for Efficient ConvNets”. In: *arXiv:1608.08710 [cs]* (Mar. 10, 2017). (Visited on 03/19/2020).
- [28] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, X. Lin, K. Ma, and Y. Wang. “Non-Structured DNN Weight Pruning – Is It Beneficial in Any Platform?” In: *arXiv:1907.02124 [cs, stat]* (Jan. 7, 2020). (Visited on 07/09/2020).
- [29] E. Malach, G. Yehudai, S. Shalev-Shwartz, and O. Shamir. “Proving the Lottery Ticket Hypothesis: Pruning is All You Need”. In: *arXiv:2002.00585 [cs, stat]* (Feb. 3, 2020). (Visited on 07/01/2020).
- [30] S. Mehta, M. Rastegari, L. Shapiro, and H. Hajishirzi. “ESPNetv2: A Light-weight, Power Efficient, and General Purpose Convolutional Neural Network”. In: *arXiv:1811.11431 [cs]* (Mar. 30, 2019). (Visited on 01/05/2021).
- [31] F. Meng, H. Cheng, K. Li, H. Luo, X. Guo, G. Lu, and X. Sun. “Pruning Filter in Filter”. In: *arXiv:2009.14410 [cs]* (Dec. 9, 2020). (Visited on 03/18/2021).
- [32] *PyTorch Release Notes 1.4*. URL: <https://github.com/pytorch/pytorch/releases/tag/v1.4.0>.
- [33] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár. “Designing Network Design Spaces”. In: *arXiv:2003.13678 [cs]* (Mar. 30, 2020). (Visited on 05/08/2021).
- [34] P. Ramachandran, B. Zoph, and Q. V. Le. “Searching for Activation Functions”. In: *arXiv:1710.05941 [cs]* (Oct. 27, 2017). (Visited on 01/14/2021).
- [35] L. Roeder. *Netron*. URL: <https://github.com/lutzroeder/netron>.
- [36] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *arXiv:1801.04381 [cs]* (Mar. 21, 2019). (Visited on 10/21/2020).

- [37] M. Shoenybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism”. In: *CoRR abs/1909.08053* (2019).
- [38] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *arXiv:1703.09039 [cs]* (Aug. 13, 2017). (Visited on 10/29/2020).
- [39] *TensorFlow Keras Comprehensive Pruning*. URL: https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide.
- [40] Tesla. *Tesla AI*. URL: <https://www.tesla.com/autopilotAI>.
- [41] S. Verdenius, M. Stol, and P. Forré. “Pruning via Iterative Ranking of Sensitivity Statistics”. In: *arXiv:2006.00896 [cs, stat]* (June 14, 2020). (Visited on 09/24/2020).
- [42] B. Whetton. *Keras-surgeon*. May 17, 2017. URL: <https://github.com/BenWhetton/keras-surgeon> (visited on 10/27/2020).
- [43] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: *arXiv:1609.08144 [cs]* (Oct. 8, 2016). (Visited on 10/27/2020).
- [44] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang. “Gate Decorator: Global Filter Pruning Method for Accelerating Deep Convolutional Neural Networks”. In: *arXiv:1909.08174 [cs, eess]* (Sept. 17, 2019). (Visited on 03/18/2021).
- [45] F. Yu, D. Wang, E. Shelhamer, and T. Darrell. “Deep Layer Aggregation”. In: *arXiv:1707.06484 [cs]* (Jan. 4, 2019). (Visited on 05/08/2021).
- [46] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. “Slimmable Neural Networks”. In: *arXiv:1812.08928 [cs]* (Dec. 20, 2018). (Visited on 03/18/2020).
- [47] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. “Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture. ISCA ’17: The 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, June 24, 2017, pp. 548–560. (Visited on 05/20/2020).
- [48] S. Zhang, L. Wen, X. Bian, Z. Lei, and S. Z. Li. “Single-Shot Refinement Neural Network for Object Detection”. In: *CoRR abs/1711.06897* (2017).
- [49] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik. “Neural Network Distiller: A Python Package For DNN Compression Research”. In: *arXiv:1910.12232 [cs, stat]* (Oct. 27, 2019). (Visited on 02/18/2021).

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 der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria 2021



Andreas Glinserer