UNIVERSITY OF PERUGIA
UNIVERSITÀ DI PERUGIA

Department of Mathematics and Computer Science
Dipartimento di Matematica ed Informatica

Master Degree in Computer Science
Laurea specialistica in Informatica

Master's Degree Thesis:

Per-Layer Pruning with Heuristic

Pruning layers of a neural network with a heuristic-based approach.

Author:
Diego Russo, 231423
me@diegor.it

Supervisors:
Prof. Alfredo Milani
PhD. Anton Katchaktou

Academic Year 2019/2020

Contents

List of Figures

Listings

"Now is better than never."

from "The Zen of Python"

This thesis represents the end of a long journey started more than a decade ago. Studying part-time has not been easy but it is definitively doable if one has the right amount of perseverance, patience and determination: when I enrolled to the master's degree I was already working as software developer and I have been working since then.

It has been a bumpy ride, full of unexpected turns of events which slowed down the progress of my studies but at the same time they contributed to my personal and professional life and they brought me where I am now: 2 more spoken languages, new home country, new culture, new experiences, plenty of working experience and, the most important, working on AI/ML field, a long-awaited move.

I have always seen the master course as way to learn new things (even from the "boring" topics), to improve myself and to keep my mind in constant training. Although stressful, and sometimes hard, I would do it again.

This has been possible thanks to the support of many people who in a way or another have helped me to go through difficult times.

First of all, to my family who has always believed and trusted me: their support has been vital to get to the point where I am today.

My girlfriend Carmen who unconditionally has supported me during the last few years for the final sprint of exams and thesis. Gracias mi amor!

Friends and people who, even with small words or actions, have given me the strength to continue and to finish this journey.

All the professors who, during all these years, have understood my working abroad situation demonstrating flexibility and accommodating my needs. Amongst them, a special thanks to Professor Alfredo Milani, the supervisor of this thesis.

Last but not least, a huge thank you to my employer Arm Ltd. They have been very supportive with my studies during all these years allowing me extra days whenever I needed to sit exams and for allowing me to work part-time during my final sprint.

Within Arm, I would like to thank my current line manager and supervisor of this thesis, Anton Kachatkou: his support has been fundamental for the execution of this work. Anton supported and helped me to get the right approvals within Arm, IP reviewed every single word/code of this thesis and spent some of his time to discuss about this work.

I also wish to thank my team members, Mohamed Nour Abouelseoud, Ruomei Yan and Johan Gras for reviewing my code and helping with experiments.

After all, it is a team effort, and the excellent results of this thesis are because of everyone's contribution.

Thank you all.

Abstract

In the last decade *Deep Learning* has had an incredible success due to the increasing availability of big data and the decreasing cost of computational power. A massive amount of research effort has regarded the investigation of new neural network models for application to different domains, and an increasing interested has arisen techniques on for optimizing deployment of such models to edge devices with limited resources. The aim of these techniques is to optimize the generated deep learning models to be more efficient with respect to computational power and memory requirements while having no or little loss in accuracy, thus allowing efficient deployment of a neural network model to edge devices. The aim of this thesis is to investigate and validate a specific pruning technique: layers of a network model are pruned based on a heuristic whilst respecting the target sparsity of the network model. After introducing the theory behind this approach, the application design is presented, and its implementation experimented to a simple MNIST based network model using TensorFlow Model Optimization. Experiments have been held on MobileNet v1 architecture using CIFAR-10 and ImageNet 2012 as datasets. Experimental results of the pruning technique are presented, in particular we observe that, the heuristic distribution of weights behave more robustly compared to the uniform distribution especially with higher sparsity levels.

1 Introduction

In the recent years we have been witnessing an incredible increase of Artificial Intelligence (AI) applications and services. Machine Learning (ML) and Deep Learning (DL) have been outperforming other classical AI algorithms in many fields, like face recognition and natural language processing.

This has been possible thanks to an exceptional increase of data availability and computational power, both critical for realizing the "training" phase of neural networks. Once the neural network has been trained, it can be used for the actual purpose that it has been trained for: this phase is called "inference", and it consists in providing the network with the input and computing the output response, e.g., recognition, classification, regression, prediction, etc...

The training phase is the most computationally intensive, and usually it is done off-line on special purpose powerful hardware resources where the amount of "big data" required for training is available. On the other hand, the *inference* phase is held where the intelligence service is made available, and the computational and memory requirements of this phase, although less than those required for training are neither negligible nor small. In section 1.1 these phases, their characteristics and relationships are explained more in details.

Applications that require to apply intelligent inference are therefore computationally intensive and quite demanding CPUs, GPUs, and memory power out of the device where the application is deployed and this does not always make possible to deliver efficiently the application to edge devices, i.e., devices close to the end user. In the last years the performance of those edge devices has been getting better and better but still not sufficient for all more computationally intensive applications.

A common solution to overcome this limitation is to adopt a centralised data management and processing approach (see 1.1): the device collects data, send the data to the cloud for processing and it receives back a response to their request. We refer to this approach as ***centralised intelligence***. Although this design is the basis of many applications, it presents some drawbacks:

- **amount of data**: the data generated by the device is sent to the cloud for processing. The amount of this data is not trivial, and this could overload communication channels.
- **persistent connection**: in order to ensure a robust and reliable appli-

cation a persistent network connection between the device and the cloud computing is required.

- **real-time**: some applications cannot function properly when in presence of latency introduced by the communication between the device and cloud and any delay introduced by the cloud in processing the request.
- **data privacy**: the device might send personal data in transit to the cloud, arising privacy issues in case of hacking or accidental leaking of information, on the other hand cryptographic encoding of channels can introduce a relevant computational overhead.

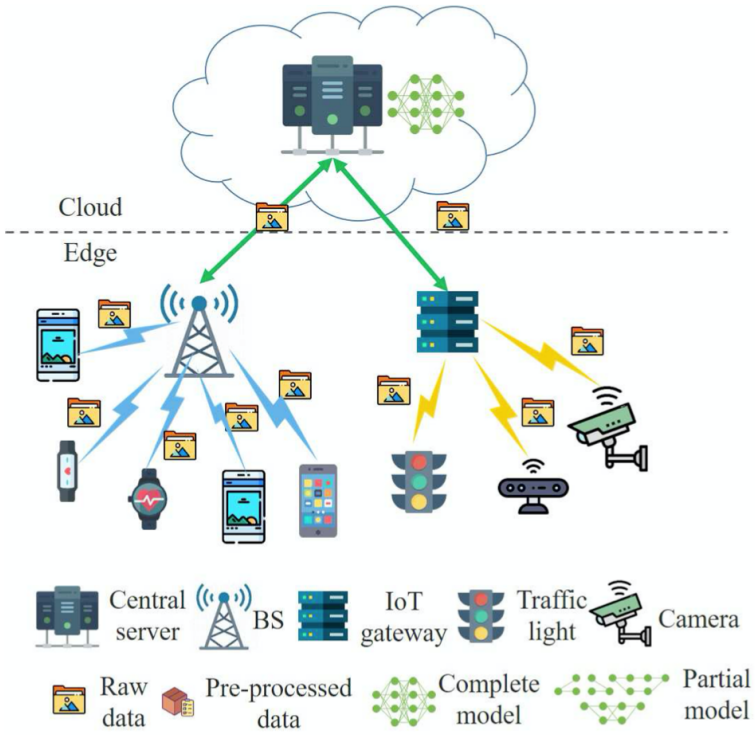


Figure 1.1: Centralised intelligence

In order to overcome to the above issues, a different approach is needed where the data processing is not centralised but distributed closer where it is generated: this different approach is called ***edge computing***, Computing, storage and networking resources are located at the edge of the network (IoT gateways, routers, etc...) whilst end devices like mobile phones and IoT devices request services from edge servers are called edge devices. It is easy to understand how this approach can address some of the issues arising in a centralised intelligence architecture: low latency between edge devices and edge servers and data exchange and data privacy are somehow mitigated. It is worth noticing that edge computing is not a replacement for cloud computing. On the contrary, it complements cloud computing, and both are targeting different kind of applications.

If we extend a little more according to this design perspective, the basis of edge computing combined with AI creates what is called **edge or mobile intelligence** (see 1.2).

This means that the data collection, caching, processing and analysis can take place locally, i.e., on the device where the data is generated. With this architectural model: latency, data privacy, network and communication load are all contained and improved, giving a better experience to the end user and eventually making the final applications and the whole system architecture more reliable.

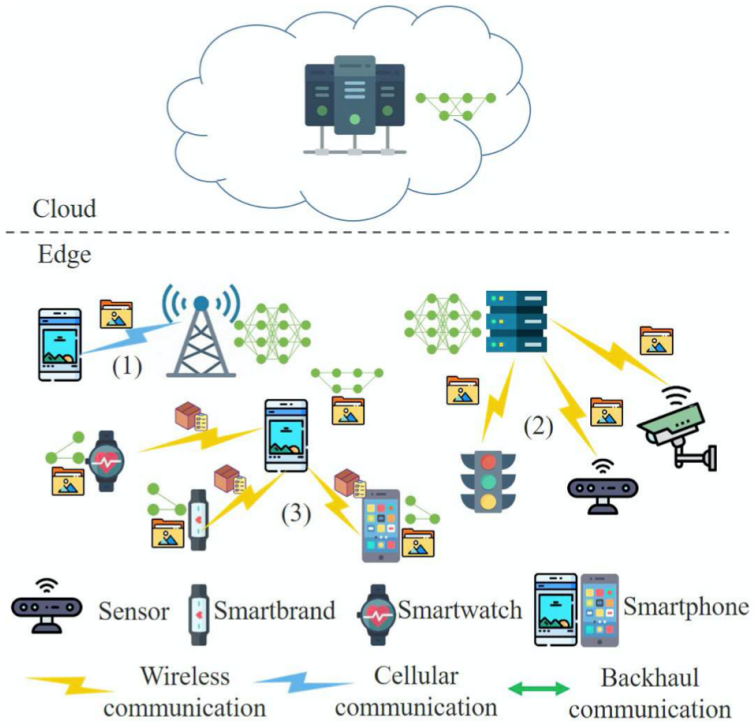


Figure 1.2: Edge intelligence

Having more powerful end devices allows to bring edge intelligence applications in end users "pockets", like smart suggestions on the keyboards based on the context of the text, photos application with integrated face recognition, based on contacts stored on the mobile and voice recognition commands for offline translation, and it is worth to point out that in all these cases: no personal data as sent or stored in the cloud, like it is generally happening now. Other examples are also self-driving cars, real-time applications and medical devices but also noise cancellation on video call application (interestingly enough, as counter example, Google Meet uses its noise cancellation model on the cloud leveraging on Google TPU infrastructure, acquiring sample of user ambience sounds [6]).

This thesis focuses mainly on the ***edge inference*** (1.3) and specifically investigated on how the models can be optimised in order to reduce memory

footprint, size and improve compression without losing accuracy in the prediction.

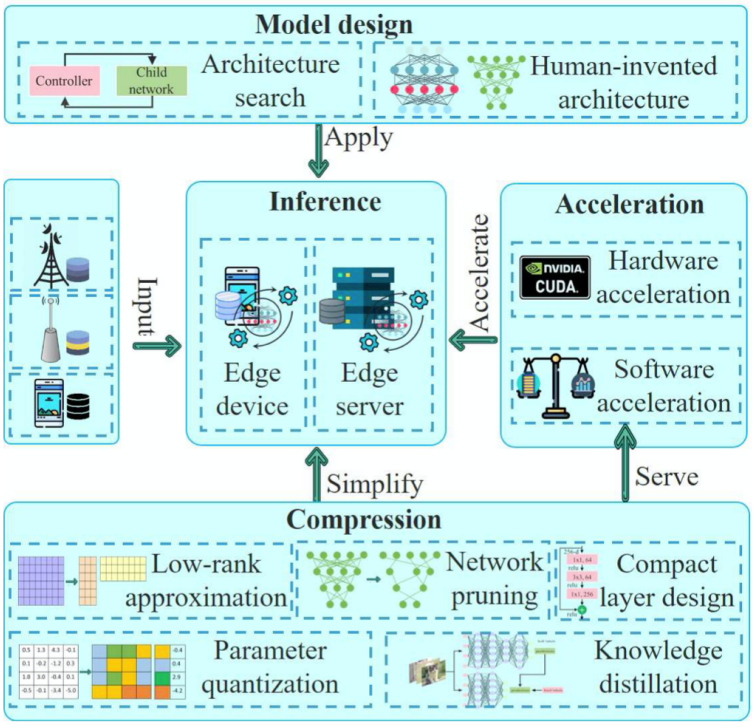


Figure 1.3: Edge inference

Edge inference is the final step of a *neural network model life cycle* where the trained model is used to infer new conclusion/prediction via a forward pass of unseen data through the neural network. This step take place on the edge device and it presents some challenges due to the limited amount of computational power and memory available on the device.

In order to work correctly and efficiently the model needs to go through a series of optimisations that decreases the power/memory consumption and latency whilst maintaining the accuracy at acceptable levels ideally minimizing or without incurring in any loss.

As summarized in figure 1.3 shows, there are different techniques that they can be used to optimise models for edge inference. These are:

- **Model design**: let the machines themselves design optimal models or human-invented architectures [33].
- **Acceleration**: hardware acceleration mainly focuses on parallel execution while software acceleration focuses on optimising resource management and compilers, based on compressed models [33].
- **Compression**: low-rank approximation, knowledge distillation, compact layer design, network pruning and parameters quantization are few techniques in order to achieve model compression [33].

With this introduction [33] we have set the background for this thesis where I focus on a specific technique of model pruning. In section 1.1 I give an overview of the entire flow of an intelligence application giving a brief explanation of every step of the flow.

In section 1.2 the main techniques of model optimisation for deployment, explaining are described and pros and cons discussed.

In chapter 2 the pruning technique is shown more in details and section section 3.1 illustrates the theory behind the *per-layer pruning configuration with heuristic*.

In chapter 3 I explain the per-layer pruning with heuristic and the architecture of an implementation in TensorFlow Model Optimization giving full explanation and a working example.

Finally in chapter 4 experimental results obtained experimenting the pruning technique on well-known neural networks are presented and the benefits of this approach are discussed.

Unless specified otherwise, **all the examples, code and documentation assume the use of TensorFlow (`https://www.tensorflow.org/`) and its ecosystem.**

1.1 From training to inference

The pillars of edge intelligence are **data, model and computation**. In general, a big amount of *data* is needed, in order to train a *model* that behaves as expected, while computational resources are needed in different amount throughout the whole process, from training to edge inference.

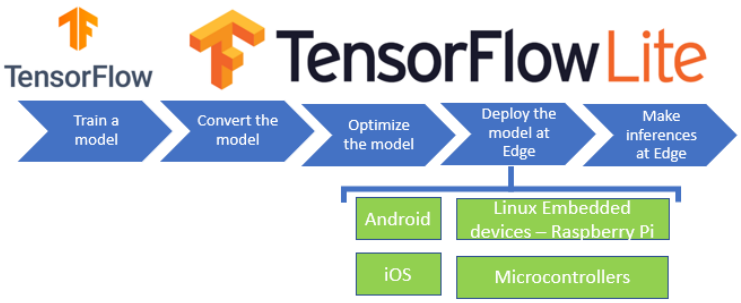


Figure 1.4: From training to edge inference

The figure Figure 1.4 shows a classic life cycle flow of an intelligent application. A brief explanation of each step:

- **Train a model**: once decided what the task is, the right model structure needs to be used. There are different options: create a custom model, use a pre-trained model or use Transfer Learning on a pre-trained model.
- **Convert the model**: once the model is trained, it needs to be converted in a special format (`.tflite`) that can be used on edge devices in efficient manner.

- **Optimise the model**: that's the key phase where the model is optimised to use less space/memory and to have a faster inference performance by decreasing the latency. More details will be presented in section section 1.2.
- **Deploy the model at Edge**: once the model has been optimised, it is ready to be deployed to the edge device.
- **Make inference at Edge**: this is the last step where finally the model is used to do inference on new data that it has never seen and hopefully providing expected results in a timely fashion.

The above flow shows the big picture of an *intelligent application life cycle* [19] and gives an idea of the complexity behind the creation of an edge intelligent application. In the rest of the chapter, I will explain what techniques are available in TensorFlow in order to **optimise the model**.

1.2 Model optimisations for deployment

To optimise TensorFlow models, the ecosystem offers *TensorFlow Model Optimization Toolkit* (abbrev. *TFMOT*) which minimizes the complexity of optimising machine learning inference.

Inference efficiency is a critical concern when deploying machine learning models because of latency, memory utilization, and in many cases power consumption. Particularly on edge devices, such as mobile and Internet of Things (IoT), resources are further constrained, and model size and efficiency of computation become a major concern.

Model optimisation is useful, among other things, for:

- Reducing latency and cost for inference for both cloud and edge devices (e.g., mobile, IoT). Latency in machine learning refers to the time taken by the model to process one unit of data when there is no parallelism i.e., time taken by MobileNet to classify an image.
- Deploying models on edge devices with restrictions on processing, memory and/or power-consumption.
- Reducing payload size for over-the-air model updates.
- Enabling execution on hardware restricted-to or optimised-for fixed-point operations.
- Optimising models for special purpose hardware accelerators.

The most common available techniques for model optimization are:

- **Weight Clustering**: Clustered models are those where the original model's parameters are replaced with a smaller number of unique values [9].
- **Quantization**: Quantized models are those where we represent the models with lower precision, such as 8-bit integers as opposed to 32-bit float. Lower precision is a requirement to leverage certain hardware [9].

- **Weight Pruning**: Sparse models are those where connections in between operators (i.e., neural network layers) have been pruned, introducing zeros to the parameter tensors [9].

When pre-optimised models and post-training tools do not satisfy your use case, the next step is to try the different training-time tools.

Training time tools piggyback on the model's loss function over the training data such that the model can "adapt" to the changes brought by the optimisation technique. [9]

1.2.1 Weight clustering

Clustering, or weight sharing, reduces the number of unique weight values in a model, leading to benefits for deployment. It first groups the weights of each layer into N clusters, then shares the cluster's centroid value for all the weights belonging to the cluster.

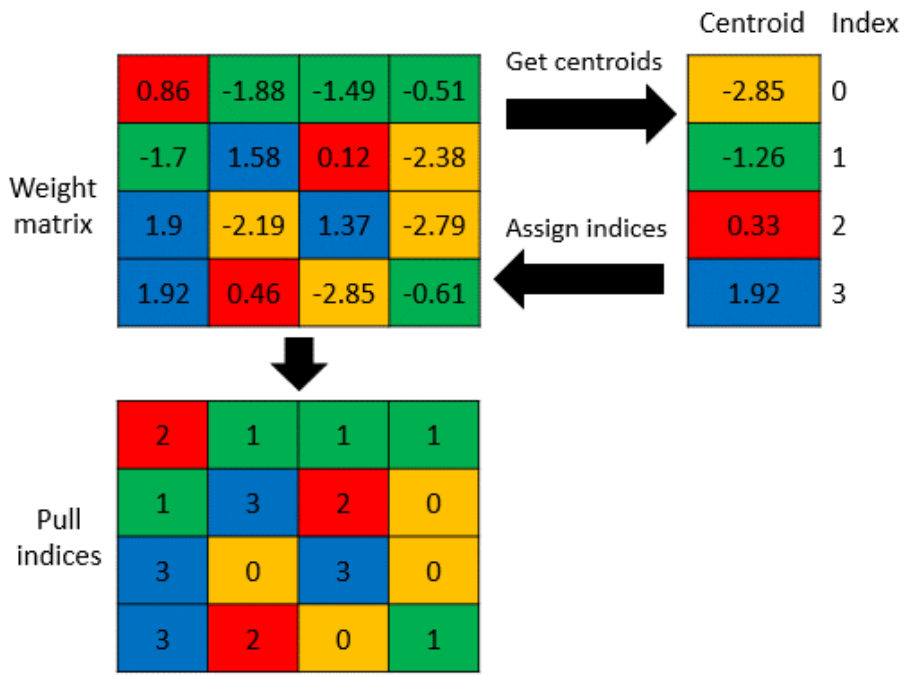


Figure 1.5: Weight clustering

A brief explanation of Figure 1.5. For example, a layer in the model contains a 4×4 matrix of weights (represented by the "weight matrix" in Figure 1.5). Each weight is stored using a float32 value. When the model is saved, 16 unique float32 values are stored.

Weight clustering reduces the size of the model by replacing similar weights in a layer with the same value. These values are found by running a clustering algorithm over the model's trained weights. The user can specify the number of clusters (in this case, 4). This step is shown in "Get centroids" in the diagram,

and the 4 centroid values are shown in the "Centroid" table. Each centroid value has an index (03).

Next, each weight in the weight matrix is replaced with its centroid's index. This step is shown in "Assign indices". Now, instead of storing the original weight matrix, the weight clustering algorithm can store the modified matrix shown in "Pull indices" (containing the index of the centroid values), and the centroid values themselves.

In this case, the size has been reduced from 16 unique floats, to 4 floats and 16 2-bit indices. The savings increase with larger matrix sizes.

Note that even if we still stored 16 floats, they now have just 4 distinct values. Common compression tools (like zip) can now take advantage of the redundancy in the data to achieve higher compression.

Weight clustering has an immediate advantage in reducing model storage and transfer size across serialization formats, as a model with shared parameters has a much higher compression rate than one without. This is like a sparse (pruned) model, except that the compression benefit is achieved through reducing the number of unique weights, while pruning achieves it through setting weights below a certain threshold to zero. Once a model is clustered, the benefit of the reduced size is available by passing it through any common compression tool.

To further unlock the improvements in memory usage and speed at inference time associated with clustering, specialized run-time or compiler software and dedicated machine learning hardware is required. [1]

This technique brings shows improvements via model compression. Future framework support can unlock memory footprint improvements that can make a crucial difference for deploying deep learning models on embedded systems with limited resources.

According to Google experiments [11], models can be compressed up to 5x with minimal loss of accuracy. Here below results on vision Figure 1.6 and speech models Figure 1.7.

Model	Original		Clustered			
	Top-1 accuracy (%)	Size of compressed .tflite (MB)	Configuration	# of clusters	Top-1 accuracy (%)	Size of compressed .tflite (MB)
MobileNetV1	71.02	14.96	Selective (last 3 Conv2D layers)	256, 256, 32	70.62	8.42
			Full (all Conv2D layers)	64	66.07	2.98
MobileNetV2	72.29	12.90	Selective (last 3 Conv2D layers)	256, 256, 32	72.31	7.00
			Full (all Conv2D layers)	32	69.33	2.60

Figure 1.6: Weight clustering on image classification

Size of compressed `.tflite` refers to the size of the zipped `.tflite` file obtained from the model from the following process:

1. Serialize the Keras model into `.h5` file
2. Convert the `.h5` file into `.tflite`
3. Compress the `.tflite` file into a zip

Model	Original		Clustered			
	Top-1 accuracy (%)	Size of compressed .tflite (MB)	Configuration	# of clusters	Top-1 accuracy (%)	Size of compressed .tflite (MB)
DS-CNN-L	95.03	1.5	Full	32	94.71	0.3

Figure 1.7: Weight clustering on keyword spotting

The weight clustering implementation is based on the *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding* [12] [11]

1.2.2 Quantization

There are two forms of quantization: **post-training quantization** and **quantization aware training (QAT)**. The former is easy to use whilst the latter often offers better model accuracy.

Quantization is lossy

Quantization is the process of transforming an ML model into an equivalent representation that uses parameters and computations at a lower precision. This improves the model's execution performance and efficiency

However, the process of going from higher to lower precision is intrinsically lossy in nature. As seen in Figure 1.8, quantization squeezes a small range of floating-point values into a fixed number of information buckets.

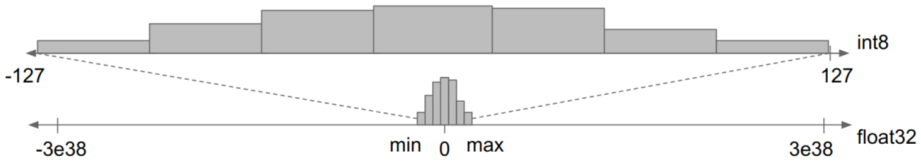


Figure 1.8: Quantization

This leads to information loss. The parameters (or weights) of a model can now only take a small set of values and the minute differences between them are lost. For example, all values in range [2.0, 2.3] may now be represented in one single bucket. This is like rounding errors when fractional values are represented as integers.

There are also other sources of loss. When these lossy numbers are used in several multiply-add computations, these losses accumulate. Further, int8 values, which accumulate into int32 integers, need to be rescaled back to int8 values for the next computation, thus introducing more computational error. [28]

Quantization aware training

Quantization aware training (QAT) emulates inference-time quantization, creating a model that downstream tools will use to produce actually quantized

models. The quantized models use lower-precision (e.g., 8-bit instead of 32-bit float), leading to benefits during deployment.

Quantization brings improvements via model compression and latency reduction. With default API, Google has observed that the model shrinks by 4x and 1.5 4x improvements in CPU latency. Eventually, latency improvements can be seen on compatible machine learning accelerators (e.g., EdgeTPU and NNAPI).

At the time of writing, TensorFlow Model Optimization implementation of QAT is still under development and has some limitations regarding to its functionality (distributed training, limited support for Subclassed Models, RNN/LSTM models, stable APIs, etc. . .)

Model	Non-quantized Top-1 Accuracy	8-bit Quantized Accuracy
MobilenetV1 224	71.03%	71.06%
Resnet v1 50	76.3%	76.1%
MobilenetV2 224	70.77%	70.01%

Figure 1.9: QAT on image classification

The models in Figure 1.9 were tested on ImageNet and evaluated in both TensorFlow and TFLite [8].

For a reference background, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference* paper [18] introduces some concepts that this tool uses. The TensorFlow Model Optimization implementation is not based on the same ideas, and there are additional concepts used in this tool (e.g., per-axis quantization) [8].

Post-training Quantization

Post-training quantization includes general techniques to reduce CPU and hardware accelerator latency, processing, power, and model size with little degradation in model accuracy. These techniques can be performed on an already trained float TensorFlow model and applied during TensorFlow Lite conversion. These techniques are enabled as options in the TensorFlow Lite converter.

Two types of post-quantization exist:

- **Quantizing weights**: Weights can be converted to types with reduced precision, such as 16-bit floats or 8-bit integers. Google generally recommends 16-bit floats for GPU acceleration and 8-bit integer for CPU execution. At inference, the most critically intensive parts are computed with 8-bits instead of floating point.
- **Full integer quantization of weights and activations**: Improve latency, processing, and power usage, and get access to integer-only hardware accelerators by making sure both weights and activations are quantized. This requires a small representative data set. The resulting model will still take float input and output for convenience.

There are several post-training quantization options to choose from. Figure 1.10 shows a summary table of the choices and the benefits they provide. [7]

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Figure 1.10: Post-quantization techniques

Compared to their float counterparts, quantized models are up to 2–4x faster on CPU and 4x smaller (See Figure 1.11).

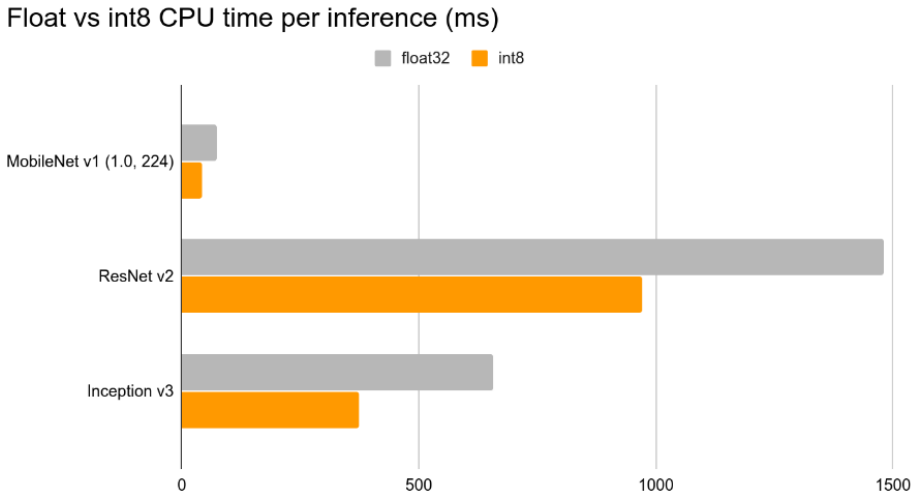


Figure 1.11: Post-quantization latency

With just 100 calibration images from ImageNet dataset, fully quantized integer models have comparable accuracy with their float versions (MobileNet v1 loses 1%) (See Figure 1.12).

Model	Float baseline	Quantization during training	Quantization after training
MobileNet v1 (1.0, 224)	70.95%	69.97%	69.568%
ResNet v2	76.8%	76.7%	76.652%
Inception v3	77.9%	77.5%	77.782%

Figure 1.12: Post-quantization accuracy

1.2.3 Weight pruning

Magnitude-based weight pruning gradually zeroes out model weights during the training process to achieve model sparsity. Sparse models are easier to compress, and we can skip the zeroes during inference for latency improvements.

This technique brings improvements via model compression. In the future, framework support for this technique will provide latency improvements. Google has seen up to 6x improvements in model compression with minimal loss of accuracy.

The technique is being evaluated in various speech applications, such as speech recognition and text-to-speech and has been experimented on across various vision (see Figure 1.13) and translation models (see Figure 1.14). [10]

Model	Non-sparse Top-1 Accuracy	Sparse Accuracy	Sparsity
InceptionV3	78.1%	78.0%	50%
		76.1%	75%
		74.6%	87.5%
MobilenetV1 224	71.04%	70.84%	50%

Figure 1.13: Weight pruning on image classification

Model	Non-sparse BLEU	Sparse BLEU	Sparsity
GNMT EN-DE	26.77	26.86	80%
		26.52	85%
		26.19	90%
GNMT DE-EN	29.47	29.50	80%
		29.24	85%
		28.81	90%

Figure 1.14: Weight pruning on translation

In chapter 2 I will go more in details of this technique.

1.2.4 Combine multiple optimisations

The optimisation techniques shown in section 1.2 bring excellent results in terms of model size and latency. The model can be optimised further combining those techniques. For instance, the model can be optimised using pruning, post-training quantization and then compress the result with any compression algorithm (e.g., zip, gzip) Similarly the model can be optimised using clustering, post-training quantization and compression. The optimisation can be pushed further including all the previous techniques as shown in Figure 1.15:

1. **Weight Pruning**: remove all near-zero weights
2. **Weight Clustering**: group all the remaining weights in different clusters
3. **Quantization**: the weights representing the clusters are quantized from floating point at 32-bit to integer at 16/8bit
4. **Compression**: the last step is to apply a compression algorithm to the model in order to improve deployment

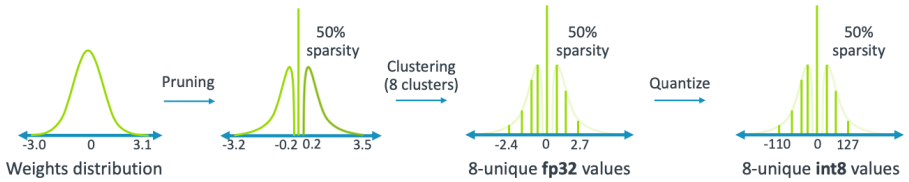


Figure 1.15: Collaborative optimisation

The paper *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding* [12] shows an optimisation pipeline (See Figure 1.16)

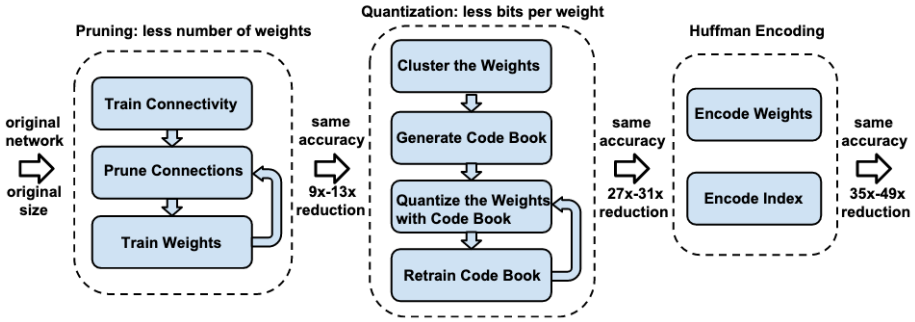


Figure 1.16: Combine multiple optimisations

Figure 1.16 shows the three-stage compression pipeline: pruning, quantization (with clustering) and Huffman coding. Pruning reduces the number of weights by 10×, while quantization/clustering further improves the compression rate: between 27× and 31×. Huffman coding gives more compression: between 35× and 49×. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

2 Pruning

In chapter 1 I gave a brief explanation of few techniques for doing model optimisation on a neural network. Pruning is one of these and in the first part of this chapter I give a more detailed explanation. The second part instead focuses on the pruning API in TensorFlow Model Optimization. This section gives the basis to fully understand the next chapter (chapter 3) where I will explain the architecture of the implementation of this technique in TensorFlow Model Optimization.

2.1 What's pruning?

Neural network pruning is the task of reducing the size of a network by removing parameters. This compression affects the size of the model, the latency, the amount of memory and the computational power needed to run the inference. These metrics need to be balanced with the accuracy of the model itself. I give a more detailed analysis about this trade-off in subsection 2.1.3

Pruning has been used since the late 1980s but has seen an explosion of interest in the past decade thanks to the rise of deep neural networks. It sets its roots with a couple of classic papers: *Optimal Brain Damage* [21] and *Optimal Brain Surgeon* [14]

In the last decade (2010–2020) a few dozen papers have been published in literature about pruning [3] and all of them have been showing that pruning is an effective technique that can be applied to a variety of neural network on different fields (image and speech recognition, text processing, etc. . .). Moreover, they highlight that pruning is a versatile technique as, I said earlier, it has a positive impact on multiple metrics, all important for a better edge deployment of the model.

How does pruning reduce the size of a model? The basic principle is to prune (remove) unnecessary neurones or weights (see Figure 2.1):

- **weights**: this is done by setting individual parameters to zero and making the network sparse. The effect will be to maintain the same architecture of the network but lowering down the number of parameters.

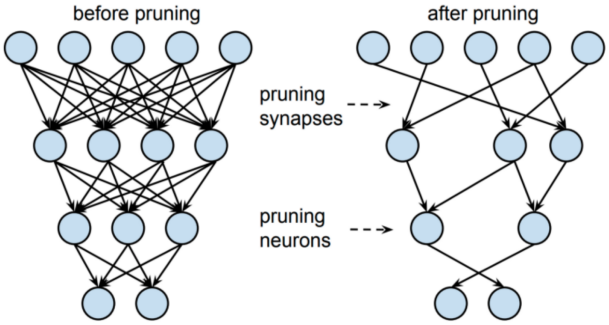


Figure 2.1: Pruning weights and neurons

- **neurons**: this is done by removing the entire node from the network with all its connections. This would make the network architecture smaller but with the target to keep the accuracy of the starting network.

2.1.1 Pruning techniques

The main problem in pruning is to understand what to prune. Of course, the goal is to remove nodes and/or weights that are less useful, i.e., whose removal does not greatly affect the model performance. There are different methods to understand what to prune with very little or no effect on accuracy. Below a brief description of different pruning techniques is given.

Magnitude Pruning

A neural network can be expressed in many ways and functions can be a very simple case of a neural network. Its coefficients can be changed in order to learn the input data points. There are coefficients that, despite changing their values, they won't change the behaviour of the function and these are referred as **non-significant**. In neural networks these coefficients are weights and biases: they are **trainable parameters**, and the same non-significant concept can be applied to them with a bit more complexity.

During the back propagation (gradient descent) some weights are updated with larger gradient magnitudes (both positive and negative) than the others. These weights are the **significant** ones and the weights receiving very small gradients can be considered as **non-significant** as their impact is minimal to the optimization of the loss function. After the training, the weight magnitude of every layer can be explored in order to check which weights are significant.

The weight magnitude can then be used as a criterion for pruning the neural network. In this technique a **threshold** is specified and all the weights below this threshold are considered non-significant. This is usually combined with a **sparsity target** the network should achieve. The **non-significant weights will be zeroed**, cancelling effectively their impact in the neural network. This can be applied to biases as well and, in general, to any trainable parameter, to be precise.

Once the pruning is done it's always advisable to retrain the network in order to compensate for any drop in performance. It's worth noticing that during the

retraining the pruned weights won't be updated. [27]

Magnitude pruning is the technique I will be using in chapter 3.

Channel Pruning

Channel pruning is a technique specifically for CNN (Convolutional Neural Network) as it relies on the architecture of this type of networks. The building blocks of a classical CNN are (see Figure 2.2):

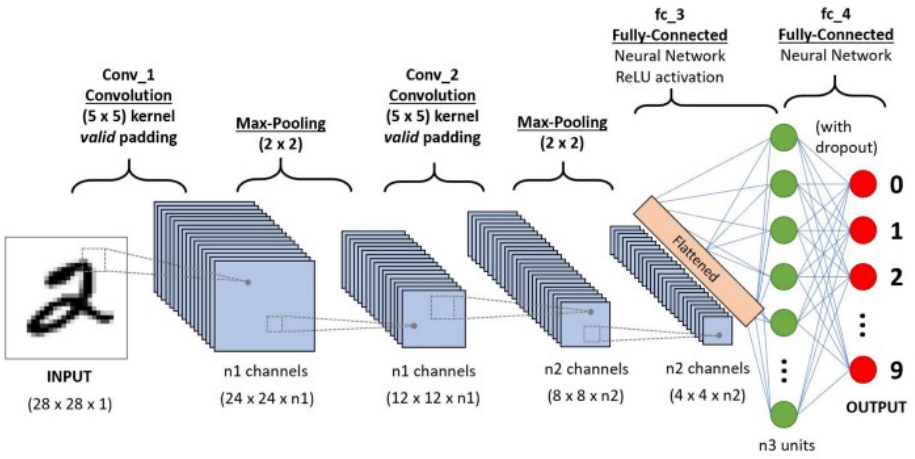


Figure 2.2: Convolutional neural network architecture

- **Convolutional layer**: it is the core building block of a CNN. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.
- **Pooling layer**: it provides an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.
- **ReLU layer**: ReLU stands for **Rectified Linear Unit** and it is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance (see Figure 2.3)
- **Fully connected layer**: after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in non-convolutional artificial neural networks.

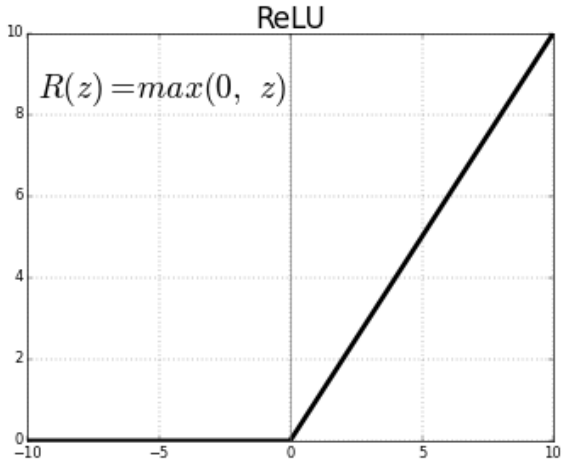


Figure 2.3: Rectified linear unit activation function

- **Loss layer**: it specifies how training penalizes the deviation between the predicted (output) and true labels and is normally the final layer of a neural network. Various loss functions appropriate for different tasks may be used. SoftMax loss is used for predicting a single class of K mutually exclusive classes. Sigmoid cross-entropy loss is used for predicting K independent probability values in [0, 1]. Euclidean loss is used for regressing to real-valued labels. [30]

Figure 2.4 shows the channel pruning algorithm for a single convolutional layer. The aim is to reduce the number of channels of feature map B, while maintaining outputs in feature map C. Once the channels are pruned, corresponding channels of the filters that take these channels as input can be removed. Moreover, filters that produce these channels can be removed as well. Channel pruning involves two key steps.

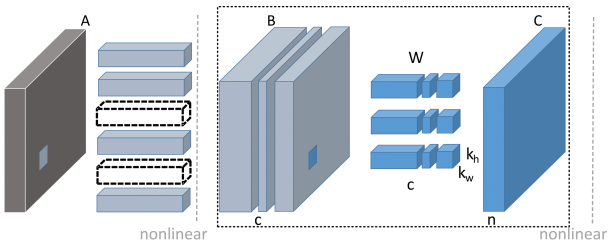


Figure 2.4: Channel pruning for accelerating a CNN

The first is the channel selection, since a proper channel combination to maintain as much information needs to be selected. The second is reconstruction. The following feature maps need to be reconstructed using the selected channels. Motivated by this, the process is an ***iterative two-step algorithm***.

In the first step, the aim is to *select most representative channels*. Since an exhaustive search is infeasible even for tiny networks, a LASSO regression-based

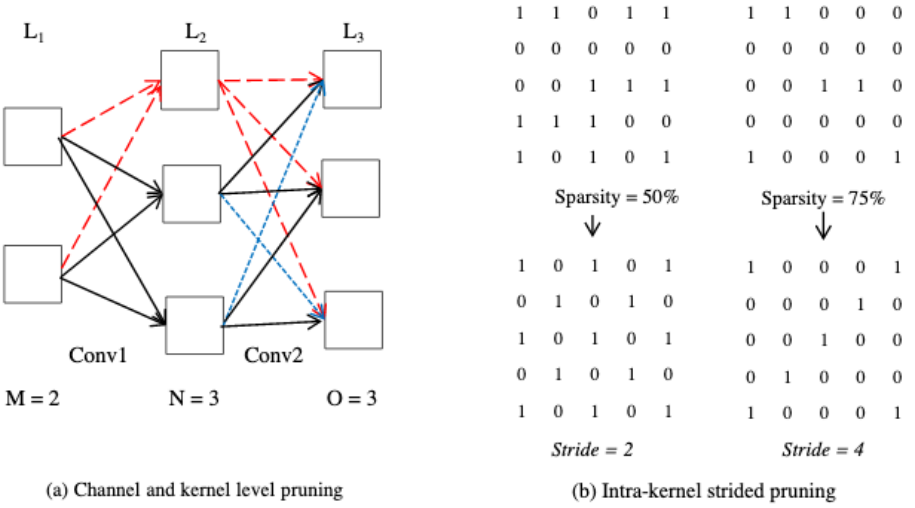
method needs to be performed to figure out representative channels and prune redundant ones.

In the second step, *outputs are reconstructed with remaining channels* with linear least squares.

The whole model can be pruned applying the approach layer by layer sequentially. For each layer, input volumes are obtained from the current input feature map, and output volumes from the output feature map of the un-pruned model. [16]

Structured Pruning

Pruning techniques can be broadly categorized as structured or unstructured. *Unstructured pruning* does not follow a specific geometry or constraint. In most cases, this technique needs extra information to represent sparse locations. It depends on sparse representation for computational benefits. On the other hand, *structured sparsity* places non-zero parameters at well-defined locations. This kind of constraint enables modern CPUs and graphics processing units (GPUs) to easily exploit computational savings.



(a) Channel and kernel level pruning

(b) Intra-kernel strided pruning

Figure 2.5: Structured pruning

Channel, kernel and intra-kernel sparsity could be a mean of *structured pruning*. In *channel level pruning*, all the incoming and outgoing weights to/from a feature map are pruned. Channel level pruning can directly produce a lightweight network. *Kernel level pruning* drops a full k × k kernel, whereas the intra-kernel sparsity prunes weights in a kernel. The intra kernel stride sparsity can significantly speed-up convolution layer processing. The kernel level pruning is a special case of *intra-kernel sparsity* with 100% pruning. These granularities can be applied in various combinations and different orders. [2]

2.1.2 Pruning pipeline

So far, I've been explaining what pruning is and what kind of techniques exist to pruning neural networks. Let's take a step back and see how these techniques fit the big picture of the pruning pipeline. There are different ways to apply pruning to neural networks:

Traditional network pruning pipeline

This is the traditional approach for pruning a neural network. It prunes redundant connections using a three-step method:

1. Train the network to learn which connections are important: unlike conventional training, the final values of the weights are not learnt, but rather which connections are important.
2. Prune the unimportant connections: all connections with weights below a threshold are removed from the network, converting a dense network into a sparse network.
3. Retrain the network to fine tune the weights of the remaining connections: that's the critical step because if the pruned network is used without retraining, accuracy is significantly impacted.

Learning the right connections is an **iterative process**. Pruning followed by a retraining is one iteration, after many such iterations the minimum number connections could be found.

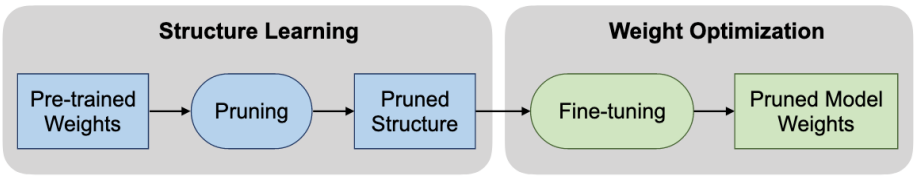


Figure 2.6: Traditional pruning pipeline

After pruning connections, neurons with zero input connections or zero output connections may be safely pruned. This pruning is furthered by removing all connections to or from a pruned neuron. The retraining phase automatically arrives at the result where dead neurons will have both zero input connections and zero output connections. This occurs due to gradient descent and regularization. A neuron that has zero input connections (or zero output connections) will have no contribution to the final loss, leading the gradient to be zero for its output connection (or input connection), respectively. Only the regularization term will push the weights to zero. Thus, the dead neurons will be automatically removed during retraining. [13]

In this thesis we will focus mainly on the traditional pipeline, but the described approach can be easily applied to other pruning pipelines.

Training from scratch

Another pruning pipeline is "training from scratch" where the network pruning is rethought. Generally, there are two common assumed beliefs behind this

pruning procedure. First, it is assumed that starting with training a large, over-parametrized network is important as it provides a high-performance model (due to stronger representation & optimization power) from which one can safely remove a set of redundant parameters without significantly affecting the accuracy. Therefore, this is usually believed, and reported to be superior to directly training a smaller network from scratch a commonly used baseline approach. Second, both the pruned architecture and its associated weights are assumed to be essential for obtaining the final efficient model. Thus, most existing pruning techniques choose to fine-tune a pruned model instead of training it from scratch. The preserved weights after pruning are usually considered to be critical.

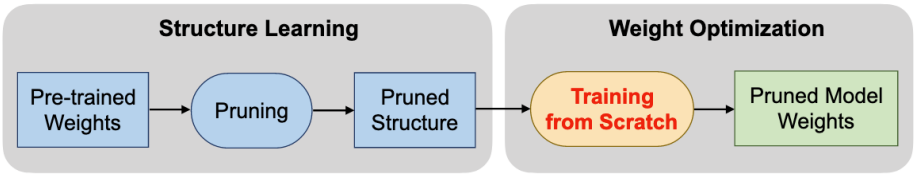


Figure 2.7: Pruning pipeline with training from scratch

With training from scratch, both beliefs mentioned above are not necessarily true for structured pruning methods, which prune at the levels of convolution channels or larger. For structured pruning methods with predefined target network architectures, directly training the small target model from random initialization can achieve the same, if not better, performance, as the model obtained from the three-stage pipeline. For structured pruning methods with auto-discovered target networks, training the pruned model from scratch can also achieve comparable or even better performance than fine-tuning. Interestingly, for unstructured pruning method that prunes individual parameters, training from scratch can mostly achieve comparable accuracy with pruning and fine-tuning on smaller scale datasets but fails to do so on the large-scale ImageNet benchmark. Note that in some cases, if a pre-trained large model is already available, pruning and fine-tuning from it can save the training time required to obtain the efficient model. [22]

Pruning from scratch

In this pipeline, pre-training an over-parametrized model is not necessary step for obtaining the target pruned structure. In fact, a fully trained over parametrized model will reduce the search space for the pruned structure.

Is it necessary for learning the pruned model structure from pre-trained weights? After some research it has been found that the answer is quite surprising: an effective pruned structure does not have to be learned from pre-trained weights [29].

It has been empirically shown that the pruned structures discovered from pre-trained weights tend to be homogeneous, which limits the possibility of searching for better structure.

In fact, more diverse and effective pruned structures can be discovered by directly pruning from randomly initialized weights, including potential models with better performance. Based on the above observations, it has been created a novel network pruning pipeline where a pruned network structure can be

directly learned from the randomly initialized weights (see Figure 2.8). This pruning pipeline allows **pruning from scratch.**

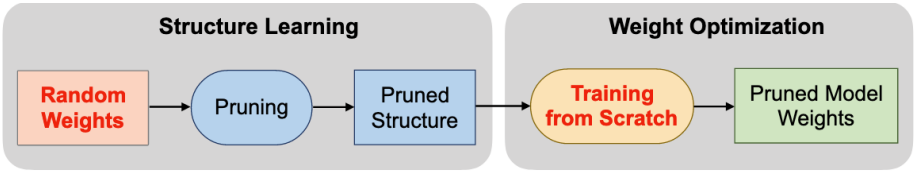


Figure 2.8: Pruning pipeline with pruning from scratch

Specifically, it is used a similar technique in Network Slimming to learn the channel importance by associating scalar gate values with each layer. The channel importance is optimized to improve the model performance under the sparsity regularization. What is different from previous works is that random weights are not updated during this process. After finishing the learning of channel importance, a simple binary search strategy is used to determine the channel number configurations of the pruned model given resource constraints (e.g., FLOPS). Since it is not needed to update the model weights during optimization, the pruned structure can be discovered at an extremely fast speed.

This approach not only greatly reduces the pre-training burden of traditional pruning methods, but also achieves similar or even higher accuracy under the same computation budget. [29]

2.1.3 Pruning-Accuracy trade-offs

When a neural network is pruned, its structure and architecture might change affecting various characteristic. Pruning affects mostly sparsity of the model and this influences model size, memory used for inference, number of parameters, speed and latency, compute power and energy needed for interference. Whilst pruning has mostly a positive impact on the above metrics, this optimization might have a negative impact on accuracy of the network model. Of course, the drop in accuracy needs to be limited or removed completely and there are techniques for doing so. In the section though, I will focus on the trade-offs which need to be considered when pruning a network model.

As the model sparsity increases, a well-designed pruning technique can efficiently reduce the parameter redundancy while maintaining the original accuracies. However, after the over-parametrization has been exploited, pruning or any other model optimization technique will start to introduce accuracy degradations which are traded for higher optimization strength and better inference performance. Three typical scenarios are illustrated in Figure 2.9

- **Accuracy-driven**: in this region, the optimizations produce minimal accuracy impact with no downside.
- **Quick deployment**: the trade-off between accuracy and performance is the key in this region.
- **Performance-driven**: this region represents the deployment on highly resource-constrained devices where the limit of optimizations is reached.

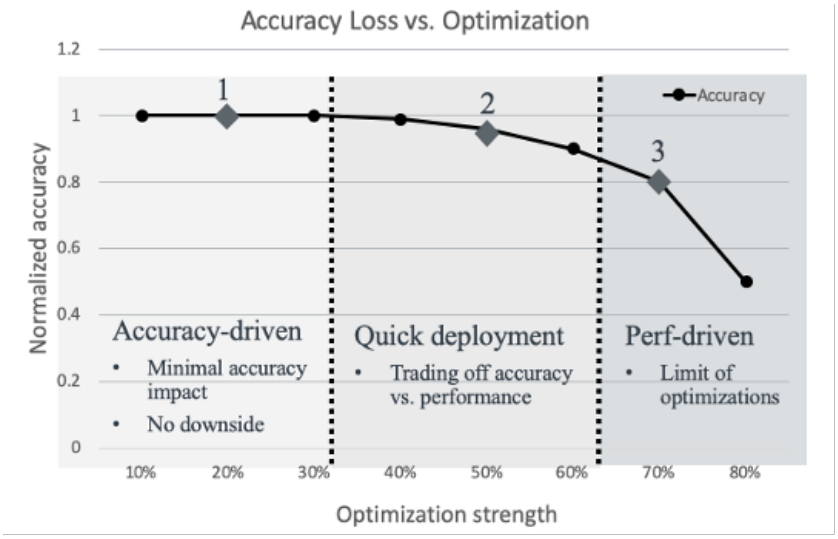


Figure 2.9: Trade-off between accuracy and optimization strength

Accuracy usually is measured with Top-1 and/or Top-5 terms. These are:

- **Top-1**: the answer given by the model must be exactly the expected one. In case of classification, it is the answer with highest probability. It is the conventional accuracy.
- **Top-5**: the expected answer is any of the first five answers with highest probability.

When calculating accuracy, Top-1/5 are used to compare against other metrics. Depending on where the model is needed for, a different level of optimizations is needed, and they might target different kinds of metrics. Pruning is one of those techniques which affect any aspect of the neural network: model size, memory used, latency etc

It can be more or less aggressively tuned depending what level of optimization the network needs to have. When pruning a neural network, due its overparametrized and redundant nature, a usual target sparsity could be between 50% and 80%. These values though depend on the model used and the result of the evaluation, hence they can be more than 80% as well.

2.2 An overview of pruning in TensorFlow Model Optimization

In this section I will explain briefly how pruning is implemented in TensorFlow Model Optimization (TFMOT).

TFMOT already supports pruning as technique to optimize neural network and **sparsity** is the module that deals with pruning.

```
import tensorflow_model_optimization as tfmot

model = build_your_model()
```

```
4
5  pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
6      initial_sparsity=0.0,
7      final_sparsity=0.5,
8      begin_step=2000,
9      end_step=4000)
10
11 model_for_pruning = tfmot.sparsity.keras.prune_low_magnitude(
12     model,
13     pruning_schedule=pruning_schedule)
14
15 ...
16
17 model_for_pruning.fit(...)
```

Listing 2.1: Pruning API in TFMOT

There are few things to explain in Listing 2.1. At the time of writing TFMOT supports two **pruning schedules**:

- `ConstantSparsity`: this schedule applies constant sparsity (%) throughout training.
- `PolynomialDecay`: this schedule applies pruning based on a polynomial function, so it is non-constant.

Both pruning schedules are taking the following parameters:

- `final/target_sparsity`: this is the sparsity the network should have once the training has ended.
- `begin_step`: step at which to begin pruning.
- `end_step`: step at which to end pruning (-1 means to prune till the end of training).
- `frequency`: only apply pruning every `frequency` step.

The `PolynomialDecay` accepts a couple of extra parameters:

- `initial_sparsity`: sparsity (%) at which pruning begins.
- `power`: exponent to be used in the sparsity function.

Which schedule to use? It seems there has not been much research on what pruning schedule to use depending on the condition. It is known that PolynomialDecay applies the pruning gradually: if the pruning is applied later in the training, the network has time to converge leading to more robust and stable results. The same can be applied with ConstantSparsity since the pruning starts, the sparsity will be applied straight away, and this might cause some instability in the results. The best way to choose the schedule is through experimentation.

2.2.1 Pruning API in TFMOT

Pruning API in TFMOT are very easy to use. Below a more detailed example that shows how to prune a model. I skip the details of building the model and focus more on the pruning. In section 3.3 I'll show details on how to create a model.

```
import tempfile
import os

import tensorflow as tf
import numpy as np

from tensorflow import keras
import tensorflow_model_optimization as tfmot

# Load MNIST dataset
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the input image so that each pixel value is between 0 to 1.
train_images = train_images / 255.0
test_images = test_images / 255.0

# Build and train the model
model = build_and_train_your_model()

prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

# Compute end step to finish pruning after 2 epochs
batch_size = 128
epochs = 2

# 10% of training set will be used for validation set
validation_split = 0.1

num_images = train_images.shape[0] * (1 - validation_split)
end_step = np.ceil(num_images / batch_size).astype(np.int32) * epochs

# Define pruning parameters
pruning_params = {
  "pruning_schedule": tfmot.sparsity.keras.PolynomialDecay(
    initial_sparsity=0.4,
    final_sparsity=0.8,
    begin_step=0,
    end_step=end_step)
}

model_for_pruning = prune_low_magnitude(model, **pruning_params)

# "prune_low_magnitude" requires a recompile
model_for_pruning.compile(
  optimizer='adam',
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=['accuracy']
)

logdir = tempfile.mkdtemp()

callbacks = [
  tfmot.sparsity.keras.UpdatePruningStep(),
  tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
]

# Fine tune with pruning for two epochs
model_for_pruning.fit(
  train_images,
  train_labels,
  batch_size=batch_size,
  epochs=epochs,
  validation_split=validation_split,
  callbacks=callbacks
)

# Evaluate the accuracy of the pruned model
_, model_for_pruning_accuracy = model_for_pruning.evaluate(
  test_images,
  test_labels,
  verbose=0
)
```

```
74 print('Pruned test accuracy:', model_for_pruning_accuracy)
```

Listing 2.2: Pruning example in TFMOT

The Listing 2.2 can be divided in the following three sections:

1. Define and train the model (lines 1–31)
2. Setup the model pruning (lines 33–49)
3. Prune, fine tuning weights and evaluate the model (lines 51–74)

Below more details of the three sections.

Define and train the model

Apart the usual imports at the top of the file, the MNIST[1] dataset is loaded generating the split between training and test images (line 12). 10% of the dataset is kept for validation purposes (line 28)

`build_and_train_your_model` is a custom function to define and train the model.

`batch_size = 128` defines the number of samples that will be propagated through the network. In this case 128 samples at the time are taken and propagated through the network. The main advantage is the memory usage: the memory footprint is the equivalent of loading 128 samples of the training data at any time.

`epochs = 2` means that the training will do a full pass twice over the full training set. Please note that in this case the epoch defines how many passes the pruning training will do.

Finally, the number if images are the full set minus 10% reserved to validation (line 30). The shape of `train_images` is the following

```
1 >>> train_images.shape
2 (60000, 28, 28)
```

Listing 2.3: Shape of train_images

The first element is the number if images present in the dataset.

With all data above, the `end_step` is then calculated.

Setup the model pruning

In this block, the model is prepared for the pruning activity. This is done by defining the pruning parameters. In this case only the pruning schedule has been defined (lines 34–40).

`prune_low_magnitude` accepts as parameters the trained model and the dictionary with the pruning settings: the function has the goal to augment the model layers with pruning information taken by the pruning parameters. The resulting model then is compiled again (lines 45–49) before calling the `fit` method.

[1] http://yann.lecun.com/exdb/mnist/

Prune, fine tuning weights and evaluate the model

After the compilation of the model, it is finally time to prune the model.

The `UpdatePruningStep` call-back is the one that updates pruning wrappers with the optimizer step. Not adding this call-back to the `fit` method will result in throwing an error.

Finally, the `fit` method fine-tunes the weights thanks to the call-back's definition above. After the pruning, the model is evaluated and return the accuracy of the new model (lines 69–74)

3 Objective: per-layer pruning with heuristic

This chapter regards the main objective of this thesis: the technique of per-layer pruning with heuristic. It is divided in three main sections: section 3.1 gives the theoretical background, while section section 3.2 and section 3.3 focus on the changes to be introduced in API in TensorFlow Model Optimization in order to use this technique.

Arm Ltd. The work illustrated in this thesis is part of the ML Tooling project within MLG (ML Group) at Arm Ltd. The vision of the project is to advance open-source tools for neural network training and optimisation tools to enable partners and ecosystem to take advantage of optimum performance and the full range of features available in Arm hardware and software.

In order to maximize the performance and power advantages, NPUs (Neural Process Unit [23, 24]) require the networks to be optimised and converted to int8/int16 format.

Improving the functionality, coverage and user experience of existing ecosystem tools will make it easy for developers to target ML for all Arm IP. This project ensures that it is easy to use open-source tools to prepare machine learning models for running efficiently on Arm IP. This includes approaches to quantizing and retraining networks to 8/16 bit, compressing weights, and reducing operations with pruning and clustering trained weights.

3.1 Per-layer pruning with heuristic

In chapter 2 various aspects are explained about pruning and how TFMOT implements it. This is a needed prerequisite to fully comprehend the content of this section.

3.1.1 Distribution of sparsity

TFMOT supports two pruning schedules: constant sparsity and polynomial decay. The schedule specifies when and what sparsity to apply to the network and this is applied equally across all layers of the network model.

This hints to the first distribution type a network model can have, **uniform distribution**. With this distribution basically the same pruning level is applied to all prunable layers, regardless of their operation types, sizes, and shapes. The uniform distribution is the only distribution implemented in TFMOT at the time of writing.

The distribution can be based on **reinforcement learning** or **optimization based**. The tuning of the sparsity distribution and other hyper-parameters can be viewed as a search problem whose solutions can be learned [15] by training a reinforcement learning agent or optimized by techniques such as simulated annealing. This approach penalizes accuracy loss and encourages model compression with single or multiple objectives. Real or estimated hardware performance statistics can be used by this approach to guide the tuning and produce target-specific optimized models.

Another option is to base the sparsity distribution on a **heuristic**. The heuristic is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. In short, heuristics can be considered as mental shortcuts that ease the cognitive load of deciding, without affecting performance in most practical cases. [31]

3.1.2 Heuristic details

After seeing examples of sparsity distribution, in this subsection I explain how the heuristic can affects the sparsity distribution.

The ***target pruning ratio*** (`pr`) can be distributed based on a simple heuristic: the more parameters a layer has, the more prunable it becomes due to the redundancy.

The function uses a heuristic to calculate the optimum sparsity per each individual layer while still maintaining the requested target pruning ratio.

The heuristic is based on the simple intuition that a layer with more weights has more redundancy and, therefore, can have a higher pruning ratio without affecting performance, while a lower number of prunable weights would be more sensitive to pruning and therefore be assigned a smaller target pruning ratio.

More specifically, the per-layer target pruning ratio can be defined as

$$pr_i = \alpha \cdot \log |layer_i| \tag{3.1}$$

where $|layer_i|$ denotes the **number of parameters in i^{th} layer**, and the **weighted coefficient** α as

$$\alpha = \frac{pr \cdot \sum |layer_i|}{\sum (|layer_i| \cdot \log |layer_i|)} \tag{3.2}$$

This heuristic has been verified with many networks and shown its effectiveness at distributing the overall target sparsity [32]. I'll be showing the effect of the heuristic in chapter 4.

Custom heuristic formula

The above heuristic formula is a fixed one and the user doesn't have any control of it. A useful extension would be to allow the user to specify his/her own heuristic formula to define the sparsity distribution across layers of the entire model.

In section 3.2 I will show how the user can use a custom formula to prune the entire model. For this to work, the object that implements the formula needs to have access to the whole model.

If the user uses the default formula, it's already proven the final sparsity is respected. This might not be the case when using custom formulas. A sanity check mechanism is needed in order to ensure the custom formula respects the final sparsity. If this is not the case, the user should be warned that the custom formula might return unexpected results as it might produce a sparsity different from the final one specified.

3.1.3 What should it be pruned?

Currently in TensorFlow it is possible to prune the following objects:

- **Model**: this could be either sequential or functional.
- **List of layers**: it is implemented but not so supported or documented.
- **Layer**: this is the basic object that can be pruned with TensorFlow pruning API.

In order to prune using the heuristic, the algorithm need knowledge of the whole model because it needs to apply different sparsity at layer level whilst maintaining a specific target/final sparsity set by the user.

For this reason, pruning with heuristic is **supported only on the whole model** hence the single layer is not supported at all and the framework should raise an error.

Pruning with heuristic to a list of layers doesn't have much sense and it doesn't seem to be a use case for that hence it is not supported, and the framework should raise an error.

3.1.4 Heuristic with PolynomialDecay scheduler

The PolynomialDecay scheduler requires some special consideration when used with heuristic.

```
...

pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
    initial_sparsity=0.3,
    final_sparsity=0.8,
    begin_step=2000,
    end_step=4000)

...
```

Listing 3.1: PolynomialDecay Scheduler in TFMOT

As it is shown in Listing 3.1, the scheduler has both initial and final sparsity. There is an issue when the **initial sparsity is bigger than the final**

sparsity calculated by the heuristic for a specific layer. For instance, if the heuristic calculates the sparsity of a layer to be 0.2, this will clash with the `initial_sparsity=0.3`. In order to address this issue, a couple of approaches can be taken:

1. When the initial sparsity is bigger than the layer's final sparsity calculated by the heuristic, the initial sparsity is ignored and set to zero. In this way, the final sparsity of the layer it will be always bigger than the initial sparsity. A warning is raised when this happens.

2. When the initial sparsity is bigger than the layer's final sparsity calculated by the heuristic, the initial sparsity is considered in proportion of the final sparsity. In Listing 3.1 the `initial_sparsity` is set to 0.3. If the heuristic calculates the final sparsity of a layer to be 0.2, then the initial sparsity for that layer will be set to $0.3 * 0.2 = 0.06$ A warning is raised when this happens.

The approach used in section 3.2 is the first one: when the initial sparsity is bigger than the sparsity that the heuristic has calculated, the initial sparsity will be ignored and set to zero. The reason why I implement the first one is to have some consistency in the pruning API: in the second approach the `initial_sparsity` has a different semantic meaning at run time depending on the value of the layer's final sparsity and this could be confusing for the user.

3.2 Architecture of the solution

The whole implementation is contained in the TensorFlow Model Optimization (TFMOT) repository[1] This because the pruning with heuristic is a training-time optimization and it is an improvement of the current pruning API.

At the time of writing the code is not public yet.

If the heuristic based pruning needs to be applied, few lines of the above example need to be changed.

```
...

from tensorflow_model_optimization.python.core.api.experimental import (
    sparsity_distribution,
)

...

pruning_params = {
  "pruning_schedule":
    tfmot.sparsity.keras.PolynomialDecay(
      initial_sparsity=0.4,
      final_sparsity=0.8,
      begin_step=0,
      end_step=end_step
    ),
  "sparsity_distribution":
    sparsity_distribution.HeuristicSparsityDistribution(),
}

...

model_for_pruning = \
  sparsity_distribution.prune_low_magnitude_custom_distribution(
```

[1] https://github.com/tensorflow/model-optimization

```
25      sequential_model, **pruning_params
26    )
27
28  ...
```

Listing 3.2: Pruning with Heuristic

The main changes compared to classical API are:

- Import the new experimental module (lines 3–5)
- Define the heuristic based sparsity distribution in the pruning parameters (lines 17–18)
- `prune_low_magnitude_custom_distribution` is called in favour of the classic method (`prune_low_magnitude`)

There are few key points to highlight in the implementation:

- `HeuristicSparsityDistribution` is a subclass of `SparsityDistribution`: the subclass needs to implement `sparsity_function(self, model, target_pruning_ratio)` method in order to define the custom distribution to apply to model's layers.

  Actual layer's information is stored in a data structure inside the instance of the class, and it will be exported using `get_sparsity_map` method.

  The class has also a mechanism that checks if the projected sparsity distribution honours the target pruning ration set by the user: in case it doesn't, it prints a warning message.

- `prune_low_magnitude_custom_distribution` is a wrapper around the classic `prune_low_magnitude` function. It has a similar signature and takes an additional parameter: `sparsity_distribution` which should be an instance of `SparsityDistribution`.

  The `sparsity_distribution` create the `sparsity_map` which will be used to wrap model's layer with sparsity distribution information.

  The function will revert back to the classic `prune_low_magnitude` in the following cases: `sparsity_distribution` is not an instance of `SparsityDistribution`, or it is not defined and the object to prune is not a sequential of functional model. In these cases, a warning will be printed saying the heuristic cannot be applied.

For a deeper understanding of the implementation, Appendix A shows the code of `HeuristicSparsityDistribution` and `SparsityDistribution`, whilst Appendix B shows `prune_low_magnitude_custom_distribution`

3.3 MNIST pipeline with heuristic pruning

The implementation code of the MNIST pipeline with heuristic pruning can be found in Appendix C.

At the very high level the pipeline does the following:

1. Build, train and evaluate the model

2. Save the model in a file
3. Prune and evaluate the new model
4. Save the pruned model in a file
5. Show the models' sizes to show the effect of pruning

The Listing 3.3 shows the output of the execution of the pipeline script. Just after the output I will be highlighting the key points to observe.

```
$ python full_heuristic_pruning_mnist.py
train_images shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3212288
_________________________________________________________________
dropout (Dropout)            (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                10250
=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
Epoch 1/4
1688/1688 [==============================] - 8s 5ms/step - loss: 0.1155 -
    accuracy: 0.9641 - val_loss: 0.0413 - val_accuracy: 0.9873
Epoch 2/4
1688/1688 [==============================] - 8s 5ms/step - loss: 0.0398 -
    accuracy: 0.9875 - val_loss: 0.0354 - val_accuracy: 0.9898
Epoch 3/4
1688/1688 [==============================] - 8s 5ms/step - loss: 0.0285 -
    accuracy: 0.9910 - val_loss: 0.0386 - val_accuracy: 0.9888
Epoch 4/4
1688/1688 [==============================] - 8s 5ms/step - loss: 0.0215 -
    accuracy: 0.9933 - val_loss: 0.0363 - val_accuracy: 0.9912
Test loss: 0.026622682809829712
Test accuracy: 0.991599977016449
Saved model to: /tmp/mnist.h5
Layer "conv2d": 800 weights of which about 287 will be zeroed. Sparsity will
    be 0.3589671368157816.
Layer "conv2d_1": 51200 weights of which about 29814 will be zeroed. Sparsity
     will be 0.5823013278812128.
Layer "dense": 3211264 weights of which about 2583624 will be zeroed.
    Sparsity will be 0.8045506230249138.
Layer "dense_1": 10240 weights of which about 5078 will be zeroed. Sparsity
    will be 0.49587367241725167.
Total weights for the model: 3273504
Projected zero weights for the model: 2618803
Projected sparsity: 0.8

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
prune_low_magnitude_conv2d ( (None, 28, 28, 32)        1634
_________________________________________________________________
```

```
prune_low_magnitude_max_pool (None, 14, 14, 32)        1
_________________________________________________________________
prune_low_magnitude_conv2d_1 (None, 14, 14, 64)        102466
_________________________________________________________________
prune_low_magnitude_max_pool (None, 7, 7, 64)          1
_________________________________________________________________
prune_low_magnitude_flatten  (None, 3136)              1
_________________________________________________________________
prune_low_magnitude_dense (P (None, 1024)              6423554
_________________________________________________________________
prune_low_magnitude_dropout  (None, 1024)              1
_________________________________________________________________
prune_low_magnitude_dense_1  (None, 10)                20492
=================================================================
Total params: 6,548,150
Trainable params: 3,274,634
Non-trainable params: 3,273,516
_________________________________________________________________
Epoch 1/2
211/211 [==============================] - 3s 15ms/step - loss: 0.0074 -
    accuracy: 0.9977 - val_loss: 0.0285 - val_accuracy: 0.9932
Epoch 2/2
211/211 [==============================] - 3s 15ms/step - loss: 0.0033 -
    accuracy: 0.9991 - val_loss: 0.0313 - val_accuracy: 0.9923
Per layer sparsity:
conv2d :  800  weights,  0.35875  sparsity
conv2d_1 :  51200  weights,  0.5823046875  sparsity
dense :  3211264  weights,  0.804550482302296  sparsity
dense_1 :  10240  weights,  0.49589843749999996  sparsity
Sparse weights: 2618803.0
All weights: 3273504
Overall model sparsity :  0.7999999389033892
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3212288
_________________________________________________________________
dropout (Dropout)            (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                10250
=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
Test loss: 0.019717451184988022
Test accuracy: 0.9940000176429749
Saved model to: /tmp/pruned_mnist.h5
Size of /tmp/mnist.h5 is 13121.672 KB
Size of zipped /tmp/mnist.h5 is 12177.406 KB
Size of /tmp/pruned_mnist.h5 is 13121.672 KB
Size of zipped /tmp/pruned_mnist.h5 is 3880.847 KB
```

Listing 3.3: MNIST pipeline output execution

The output can be divided in three main blocks. These are:

Lines 1–39

The table with the list of layers is produced by the `summary()` method. The main thing to notice is the amount of trainable and non-trainable parameters:

the latter are zero.

After the table there is the output produced by the `fit()` method follow by the loss and accuracy of the model. The accuracy is **0.9915999** which is pretty good.

Lines 40–109

The output in this block belongs to pruning. For every prunable layer, the projected sparsity is printed along with the number of weights that will be zeroed. The total projected sparsity is the reported. These figures are calculated by the `HeuristicSparsityDistribution` instance. At this point the model is not pruned yet.

The summary table is printed again and the thing to notice is that **Non-trainable** parameters are about the same amount of the trainable ones: this is the effect of the layer augmentation by the `prune_low_magnitude_custom_distribution` function. Every trainable parameter has a linked non-trainable parameter which represents the sparsity mask: before the actual pruning all values of the non-trainable parameters are set to 1.

After the pruning which runs for 2 epochs, the **actual sparsity** of the model is calculated: the sparsity matches the projected sparsity printed in the previous block. After the pruning, the model doesn't need to carry the extra information hence it is stripped: in fact, the table now shows the "right" number of parameters. The stripping of these parameters if critical if the model needs to be compressed.

The accuracy of the pruned model is **0.994** which is slightly better than the original model.

Lines 110–114

Finally, the last block of output shows information about sizes of the models saved to disk. There are four models saved:

- **mnist.h5** is 13121.672 KB
- **zipped mnist.h5** is 12177.406 KB
- **pruned_mnist.h5** is 13121.672 KB
- **zipped pruned_mnist.h5** is **3880.847 KB**

The non-zipped models have the same size: this is normal because the pruned model, although it has 80% of weights to zero, it stills needs variable allocation to host these zeroed weights.

In the zipped models instead, there is a big difference: the non-pruned model is about **12MB** whilst the pruned one is **3.8MB**: this because the pruned model has sparse weights and the one with zero can be easily compressed reducing the size.

Considering the accuracy as well, it is incredible to see how the pruning didn't impact it at all: a model with 80% sparsity results with a **slightly better accuracy** and about **30% the size of the original model.**

4 Experiments

In this chapter we present experiments held to evaluate how the pruning with heuristic affects MobileNet v1 [17] based models. The experiments have been run using both CIFAR-10 [20] and ImageNet 2012 [4]. An overview of MobileNet v1 architecture and a presentation the datasets used in the experiments is given, while experimental results are finally reported.

4.1 MobileNet v1

MobileNet is a class of efficient models for mobile and embedded vision applications (Figure 4.1) developed by Google. It is based on a streamlined architecture that uses depthwise separable convolutions to build light weight deep neural networks.

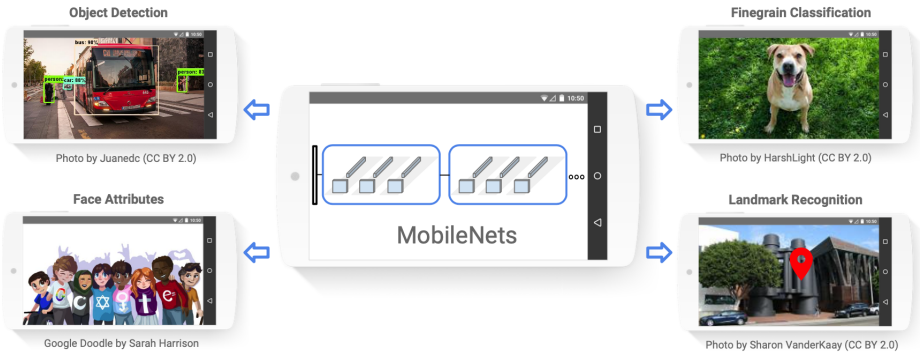


Figure 4.1: MobileNet applications

Convolutional neural networks have been more popular in the recent years and their accuracy increased thanks to more complex architecture increasing their size whilst impacting negatively in speed. In many real-world applications, these networks need to run on edge devices with limited resources and the inferences need to be carried out in a timely fashion.

The focus of MobileNet is to increase the efficiency of the network by decreasing the number of parameters by not compromising performance [26].

4.1.1 Depthwise Separable Convolution

Depthwise separable convolution is the core basis of MobileNet architecture. It is a **depthwise convolution followed by a pointwise convolution**.

A normal convolution is shown in Figure 4.2

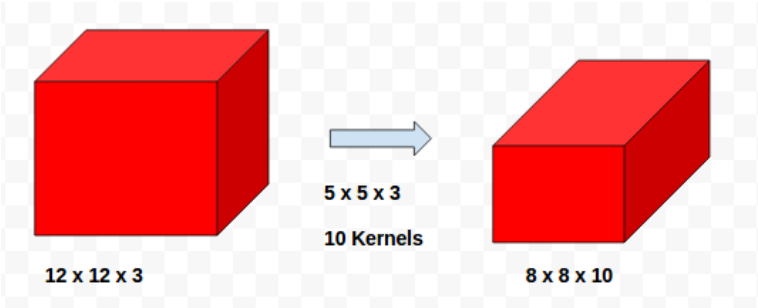


Figure 4.2: Convolution

In the figure there is an input image size of $12 \times 12 \times 3$ and the kernel (or filter) is $5 \times 5 \times 3$ with stride $= 1$. There are though 10 kernels to apply, and this gives an output image of $8 \times 8 \times 10$. The total computational cost is $\mathbf{12 \times 12 \times 5 \times 5 \times 3 \times 10 = 108000}$.

There are the following dimensions:

- Input image: $\boldsymbol{D_f \times D_f \times M}$
- Output image: $\boldsymbol{D_f \times D_f \times N}$
- Convolution kernel: $\boldsymbol{D_k \times D_k \times M \times N}$

So, in a normal convolution the total computational cost is $\boldsymbol{D_k \times D_k \times M \times N \times D_f \times D_f}$

The above convolution can be divided in 2 phases:

1. Depthwise convolution
2. Pointwise convolution

The first one is the **depthwise convolution** and it is shown in Figure 4.3.

In this case the input has 3 channels and there are 3 $5 \times 5 \times 1$ kernels. These 3 kernels are applied to the three channels respectively producing 3 $8 \times 8 \times 1$ output. When the 3 outputs are stacked the final output is $8 \times 8 \times 3$.

The second phase is the **pointwise convolution** as shown in Figure 4.4.

The output image of the previous step is the input image for this step. The convolution is done using a $1 \times 1 \times 3$ kernel on the input image producing a feature map. Repeating this using 10 different $1 \times 1 \times 3$ kernels will produce 10 different feature maps that will be stacked together.

The computation for each step is:

1. Depthwise convolution: $\mathbf{12 \times 12 \times 5 \times 5 \times 3 = 10800}$

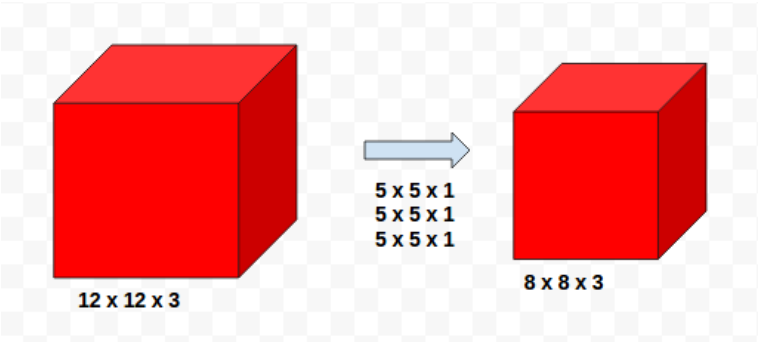


Figure 4.3: Depthwise convolution

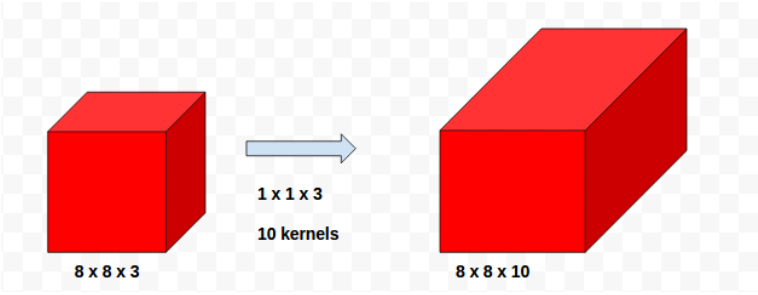


Figure 4.4: Pointwise convolution

2. Pointwise convolution: $\mathbf{8 \times 8 \times 3 \times 10 = 1920}$

Therefore, the total number of computations is $\mathbf{10800 + 1920 = 12720}$

More generically the computational cost is $\boldsymbol{D_k \times D_k \times M \times D_f \times D_f + M \times N \times D_f \times D_f}$

In this specific case using a kernel of 3×3 there is about 8 to 9 times less computational reduction: $\mathbf{108000/12720 \approx 8.45}$

4.1.2 MobileNet architecture

The network architecture is built on depthwise separable convolutions except for the first layer which is a full convolution

There are 28 convolutional layers (counting depthwise and pointwise layers) and 1 fully connected layer followed by a SoftMax layer (Figure 4.5).

All layers are followed by a batch normalization and ReLU non-linearity (Figure 4.6) except for the final fully connected layer which has no non-linearity and feeds into a SoftMax layer for classification. As seen earlier in the thesis, the SoftMax is used to predict a single class of K mutually exclusive classes. In the case of MobileNet, when used with ImageNet, it can classify up to 1000 classes.

4.1.3 Width Multiplier

MobileNet has been developed to be small and low latency but sometimes specific use cases or applications need the model to be faster and smaller. In

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 4.5: MobileNet architecture

order to construct these smaller and less computationally expensive models a very simple parameter $\boldsymbol{\alpha}$ called width multiplier has been introduced.

The role of the width multiplier α is to thin a network uniformly at each layer: the number of input channels M becomes αM and the number of output channels N becomes αN.

So, depthwise separable computational cost becomes $\boldsymbol{D_k \times D_k \times \alpha M \times D_f \times D_f + \alpha M \times \alpha N \times D_f \times D_f}$ where $\alpha \in (0, 1]$ with typical settings of 1, 0.75, 0.5, 0.25.

The Figure 4.7 shows the impact that α has on accuracy, numbers of operations and parameters.

In this thesis I consider $\boldsymbol{\alpha = 1}$ to be the baseline.

4.1.4 Resolution Multiplier

The second hyper-parameter to reduce the computational cost of a neural network is a resolution multiplier $\boldsymbol{\rho}$. This can be applied to the input image and the internal representation of every layer is subsequently reduced by the same multiplier. Including ρ, the computational cost becomes $\boldsymbol{D_k \times D_k \times \alpha M \times \rho D_f \times \rho D_f + \alpha M \times \alpha N \times \rho D_f \times \rho D_f}$ where $\rho \in (0, 1]$ which is typically set

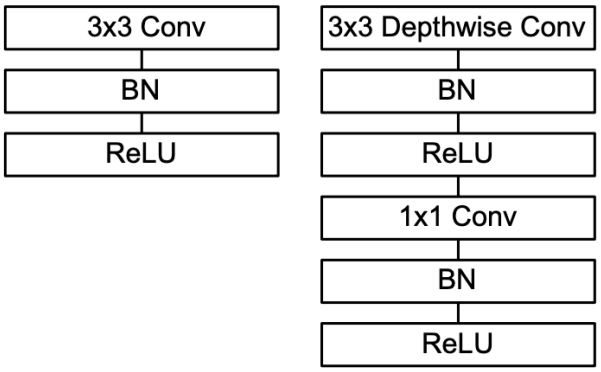


Figure 4.6: Normal convolution vs MobileNet convolution

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Figure 4.7: Width multiplier impact

implicitly so that the input resolution of the network is 224, 192, 160 or 128.

The Figure 4.8 shows the impact that ρ has on accuracy, numbers of operations and parameters.

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2

Figure 4.8: Resolution multiplier impact

In this thesis I consider $\boldsymbol{\rho = 1}$ to be the baseline.

4.2 Datasets: CIFAR-10 and ImageNet

In this section, I give a brief explanation of CIFAR-10 and ImageNet datasets.

4.2.1 CIFAR-10 dataset

The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train neural networks. It is one of the most widely used datasets for machine learning research.

It consists of **60000 32×32** colour images in **10 classes**, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks (Figure 4.9).

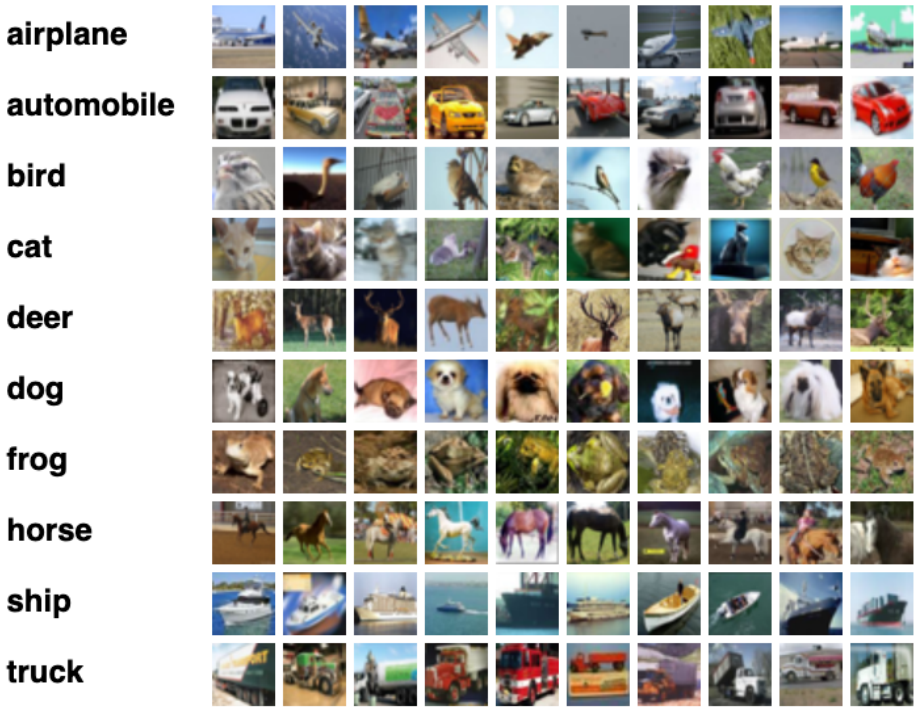


Figure 4.9: CIFAR-10 sample images

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pick-up trucks.

4.2.2 ImageNet 2012 dataset

ImageNet is an image dataset organized according to the WordNet hierarchy. Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset". There are more than 100000

synsets in WordNet, majority of them are nouns (80000+). In ImageNet, there are on average 1000 images to illustrate each synset. Images of each concept are quality-controlled and human-annotated. In its completion, ImageNet will offer tens of millions of cleanly sorted images for most of the concepts in the WordNet hierarchy (Figure 4.10)

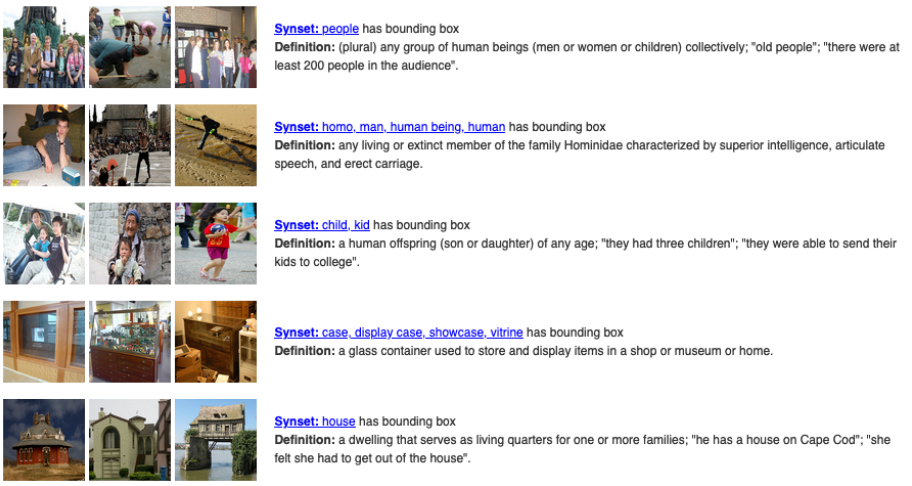


Figure 4.10: ImageNet sample images

The dataset I've used for the experiments has the following sets:

- `train` containing **1281167 images**
- `validation` containing **50000 images**

There supposed to be a `test` set, but it doesn't have labels because no labels have been publicly released.

The ILSVRC uses a "trimmed" list of only **1000 image categories** or "classes", including 90 of the 120 dog breeds classified by the full ImageNet schema.

4.3 Analysis of the experiment results

After a brief explanation of the model and the datasets used for the experiments, in this section I show what results I have in pruning MobileNet v1 with CIFAR-10 and ImageNet.

4.3.1 Environments

The environment used for running CIFAR-10 experiments has the following characteristics:

- **CPU and memory:** 2 x Intel Xeon Gold 5120T CPU @ 2.20GHz, 28 cores (56 threads) total, 64GB memory

- **Graphic:** NVIDIA TITAN Xp, 12GB GDDR5X, 3840 cores, CUDA driver version: 460.27.04, CUDA version: 11.2
- **Operating System:** Ubuntu 18.04.5 LTS, kernel 5.4.060-generic
- **Software Stack:** Python 3.8.5, TensorFlow 2.4.0, TFMOT 0.5.0.dev20210206 with my patch for heuristic distribution, and dependencies. Everything has been isolated using conda (https://docs.conda.io/en/latest/)

The environment used for running ImageNet experiments has the following characteristics:

- **CPU and memory:** AWS p3.2xlarge, 8 vCPU, 61GB memory
- **Graphic:** NVIDIA Tesla V100, 16GB, 5120 cores, CUDA Driver Version: 450.80.02, CUDA Version: 11.0
- **Operating System:** Ubuntu 18.04.5 LTS, kernel 5.4.01037-aws
- **Software Stack:** Python 3.7.6, TensorFlow 2.4.1, TFMOT 0.5.0.dev20210206 with my patch for heuristic distribution, and dependencies. Everything has been isolated using conda (https://docs.conda.io/en/latest/)

The reason I chose AWS for ImageNet is because of the time the training takes with this dataset: one epoch is about 1.5h and I needed to run 20 epochs. As I show later, the number of experiments is 20 and AWS gives me the flexibility to have multiple instances in parallel.

4.3.2 Pipeline details

In order to run experiments, I've developed a custom pipeline (not included in this thesis) which executes the following steps:

1. Load in memory the model architecture from a pre-trained checkpoint
2. Convert the model into float32, int8 and int16 tflite format
3. Prune the original model with given parameters
4. Check model sparsity and calculate loss, TOP-1 and TOP-5 accuracies
5. Convert the pruned model into float32, int8 and int16 tflite format
6. For every tflite generated, run:
   (a) Accuracy evaluation on the validation dataset (10000 samples)
   (b) Get the size and compressed size of the tflite file
   (c) Only for int8/int16 tflite files: run Ethos-U vela (subsection 4.3.3) to get an estimation of the inference speed on Ethos-U NPU and the tflite size of the vela generated tflite file.

The above pipeline has been run passing different sparsity level and distributions. I ran 10 pipelines for every dataset: (Heuristic, 0.5), (Uniform, 0.5), (Heuristic, 0.75), (Uniform, 0.75), (Heuristic, 0.8), (Uniform, 0.8), (Heuristic, 0.85), (Uniform, 0.85), (Heuristic, 0.9), (Uniform, 0.9).

The pruning scheduler chosen is `PolynomialDecay`: sparsity is introduced slowly, model weights can consider this impact and become robust to the effect of weights being dropped.

Every run of the pipeline gives the following metrics:

1. Keras model: loss, TOP-1 and TOP-5 accuracies
2. For every tflite file: sizes of the original model, compressed, and pruned compressed
3. For every int8/int16 tflite file: inference speed and vela compressed size

Later I'll show graph that compares the above metrics across sparsity levels and distributions.

4.3.3 Ethos-U Vela

Vela is a tool used to compile a TensorFlow Lite for Microcontroller (tflite) neural network model into an optimised version that can run on an embedded system containing an Arm Ethos-U NPU.

In order to be accelerated by the Ethos-U NPU the network operators must be quantised to either 8-bit (unsigned or signed) or 16-bit (signed).

The optimised model will contain TensorFlow Lite Custom operators for those parts of the model that can be accelerated by the Ethos-U NPU. Parts of the model that cannot be accelerated are left unchanged and will instead run on the Cortex-M series CPU using an appropriate kernel (such as the Arm optimised CMSIS-NN kernels).

After compilation the optimised model can only be run on an Ethos-U NPU embedded system.

The tool will also generate performance estimates for the compiled model.

For more information about vela, please refer to PyPi home page https://pypi.org/project/ethos-u-vela/

Vela memory optimization

The Vela compiler also performs various memory optimizations to reduce both the permanent (for example flash) and runtime (for example SRAM) memory requirements. One such technique for permanent storage is the compression of all the weights in the model.

Another technique is cascading, which addresses the runtime memory usage. Cascading reduces the maximum memory requirement by splitting the feature maps (FM) of a group of consecutively supported operators into stripes. A stripe can be either the full or partial width of the FM. And it can be the full or partial height of the FM. Each stripe in turn is then run through all the operators in the group.

The parts of the model that can be optimized and accelerated are grouped and converted into TensorFlow Lite custom operators. The operators are then compiled into a command stream that can be executed by the Ethos-U NPU.

Finally, the optimized model is written out as a $TFL\mu$ model and a Performance Estimation report is generated that provides statistics, such as memory usage and inference time.

The compiler includes numerous configuration options that allow you to specify various aspects of the embedded system configuration (for example the Ethos-U NPU configuration, memory types, and memory sizes). There are also options to control the types of optimization that are performed during the compilation process. [25]

4.3.4 Results for MobileNet v1 with CIFAR-10

For running this experiment, the classical MobileNet v1 architecture has been slightly modified on three layers.

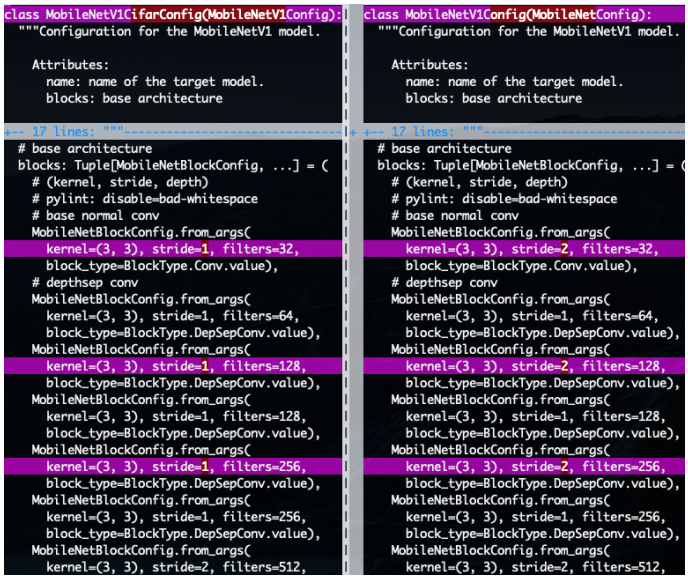


Figure 4.11: MobileNet v1 changes for CIFAR-10

The original MobileNet architecture is designed for ImageNet dataset where images' size is $224 \times 224 \times 3$. Because CIFAR-10 images have smaller size (32×32) I had to modify slightly the architecture.

The main change is that in the first layers the `stride=2` has been replaced with `stride=1`. Figure 4.11 shows what the difference is between MobileNet for CIFAR-10 and the classical architecture.

The fine-tuning of the model has been done with the following hyper parameters which have been found with Keras Tuner (https://keras-team.github.io/keras-tuner/):

- **Optimizer**: SGD (Stochastic Gradient Descend)
- **Optimizer momentum**: 0
- **Batch size**: 32
- **Learning rate**: 6.995e-06 (0.000006995)

- **Epochs**: 15
- **Sparsity scheduler**: Polynomial Decay

Accuracy and loss results

In this section I analyse data about accuracy and loss. The heuristic distribution will be always compared to the uniform distribution.

For reference, **the TOP-1 accuracy of a non-pruned MobileNet v1 with CIFAR-10 is 0.9487.**

Figure 4.12 shows how the two distributions impact the TOP-1/TOP-5 accuracies and the loss function.

The first thing to notice is that the heuristic distribution always outperforms the uniform distribution both for TOP-1 (blue line) and TOP-5 (grey line). The heuristic distribution seems more robust compared to the uniform and this is highlighted by the fact it loses less in accuracy when the sparsity is increasing. When the sparsity is set to 0.9, the heuristic distribution has an increase of **2.14% in the TOP-1** compared to the uniform distribution. A sweet spot though is at **sparsity 0.85**: the model doesn't lose too much accuracy compared to its non-pruned baseline and the sparsity is high enough to see benefits in terms of size, memory and latency of the inference. At this sparsity level, the heuristic distributions gains 1% in accuracy compared to the uniform distribution, giving an accuracy of **0.9239%**: this is **only 2.48% less** than the non-pruned accuracy.

The loss has a slightly different trend: up to a sparsity level of 0.8, there is no big difference between the two distributions with the uniform being slightly better. The trend completely flips at the latest two highest sparsity level: the loss of the uniform distribution increases much more compared to the heuristic one. This is another proof that the heuristic distribution seems more robust at higher sparsity levels.

Figure 4.13 shows accuracy across FP32, int8 and int16 tflite models both for uniform and heuristic distribution.

This graph confirms the trend seen in the previous ones: independently of the quantization scheme used, **the heuristic distribution performs better than the uniform one across all sparsity levels**. The top 3 lines (dark blue, amber and grey) represent the accuracies of the heuristic distribution whilst the bottom 3 lines (yellow, light blue and green) represent the accuracies of the uniform distribution. Moreover, the heuristic distribution seems to have a smoother trend across sparsity levels compared to the uniform distribution.

tflite size results

Figure 4.14 shows the sizes of tflite files across different sparsity levels for FP32, int8 and int16.

The sparsity distribution doesn't really influence the size of the tflite: the distribution controls how weights are distributed across layers while maintaining the final sparsity of the model. **The number of weights is exactly the same.** As example let's take the 0.85 sparsity level and see how the weights are distributed in both heuristic and uniform distribution.

```
Conv2d_0_0:            864 weights,     384 zero weights,
    0.4444444444444444 sparsity
```

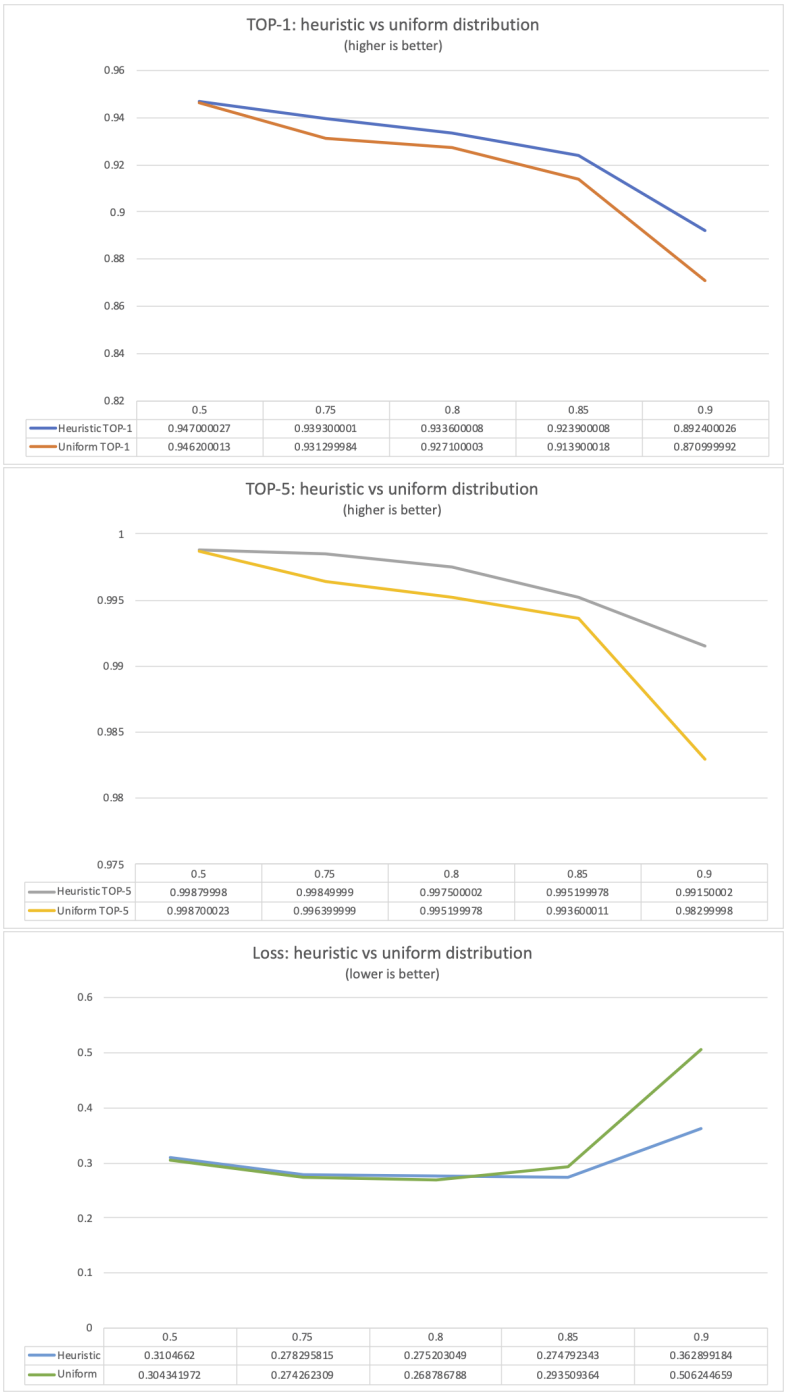
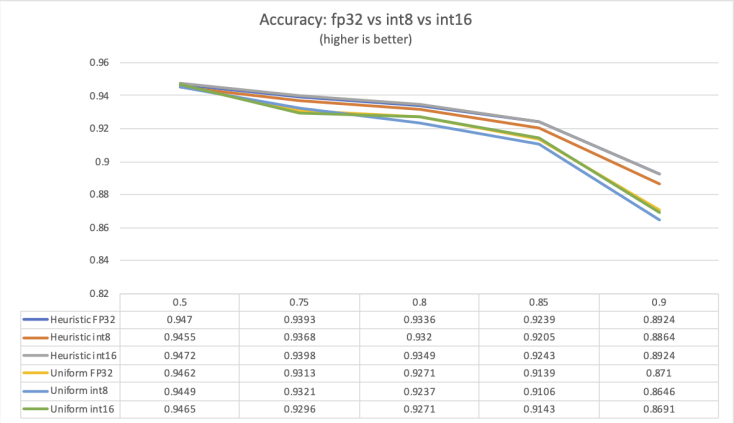


Figure 4.12: Heuristic vs Uniform: CIFAR-10 TOP-1/TOP-5 and Loss



	0.5	0.75	0.8	0.85	0.9
Heuristic FP32	0.947	0.9393	0.9336	0.9239	0.8924
Heuristic int8	0.9455	0.9368	0.932	0.9205	0.8864
Heuristic int16	0.9472	0.9398	0.9349	0.9243	0.8924
Uniform FP32	0.9462	0.9313	0.9271	0.9139	0.871
Uniform int8	0.9449	0.9321	0.9237	0.9106	0.8646
Uniform int16	0.9465	0.9296	0.9271	0.9143	0.8691

Figure 4.13: Heuristic vs Uniform: CIFAR-10 tflite accuracy

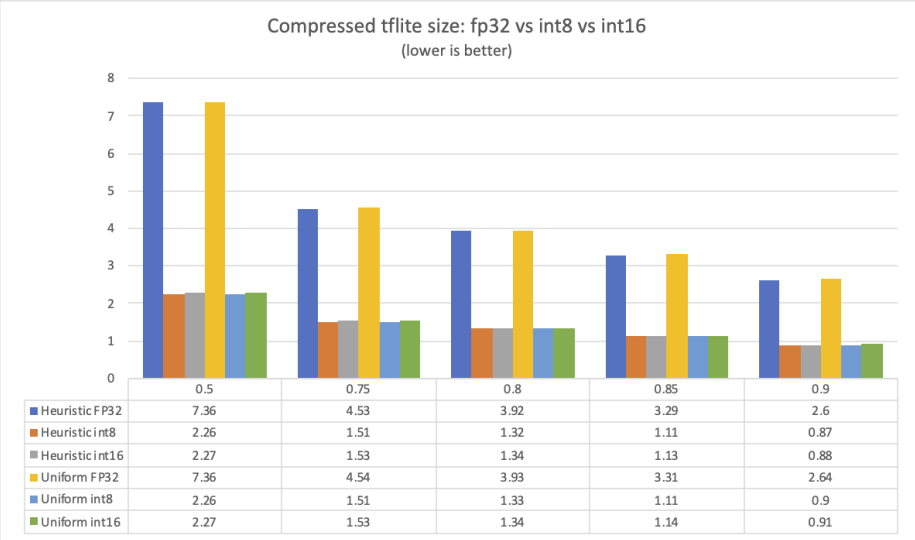
```
Conv2d_1/pointwise:    2048 weights,    1026 zero weights,   0.5009765625
    sparsity
Conv2d_2/pointwise:    8192 weights,    4851 zero weights,   0.5921630859375
    sparsity
Conv2d_3/pointwise:    16384 weights,   10448 zero weights,  0.6376953125
    sparsity
Conv2d_4/pointwise:    32768 weights,   22388 zero weights,  0.6832275390625
    sparsity
Conv2d_5/pointwise:    65536 weights,   47760 zero weights,  0.728759765625
    sparsity
Conv2d_6/pointwise:    131072 weights,  101490 zero weights,
    0.7743072509765625 sparsity
Conv2d_7/pointwise:    262144 weights,  214920 zero weights,
    0.819854736328125 sparsity
Conv2d_8/pointwise:    262144 weights,  214920 zero weights,
    0.819854736328125 sparsity
Conv2d_9/pointwise:    262144 weights,  214920 zero weights,
    0.819854736328125 sparsity
Conv2d_10/pointwise:   262144 weights,  214920 zero weights,
    0.819854736328125 sparsity
Conv2d_11/pointwise:   262144 weights,  214920 zero weights,
    0.819854736328125 sparsity
Conv2d_12/pointwise:   524288 weights,  453721 zero weights,
    0.8654041290283203 sparsity
Conv2d_13/pointwise:   1048576 weights, 955202 zero weights,
    0.9109516143798828 sparsity
top/Conv2d_1x1_output: 10250 weights,   6213 zero weights,
    0.6061463414634146 sparsity
Sparse weights: 2678083
All weights: 3150698
Overall model sparsity: 0.8499967309
```

Listing 4.1: MobileNet v1 and CIFAR-10: heuristic weights distributions

As shown in Listing 4.1, the heuristic distribution of the weights has set to zero about 91% of the weights in the layer `Conv2d_13/pointwise` while the first layers of the model have a sparsity below 50%.

For reference, Listing 4.2 shows the same sparsity level (0.85) but with a uniform distribution. All layers have 85% of the weights set to zero.

```
Conv2d_0_0:            864 weights,    734 zero weights,
    0.8495370370370371 sparsity
Conv2d_1/pointwise:    2048 weights,   1741 zero weights,   0.85009765625
    sparsity
Conv2d_2/pointwise:    8192 weights,   6963 zero weights,   0.8499755859375
    sparsity
```



	0.5	0.75	0.8	0.85	0.9
Heuristic FP32	7.36	4.53	3.92	3.29	2.6
Heuristic int8	2.26	1.51	1.32	1.11	0.87
Heuristic int16	2.27	1.53	1.34	1.13	0.88
Uniform FP32	7.36	4.54	3.93	3.31	2.64
Uniform int8	2.26	1.51	1.33	1.11	0.9
Uniform int16	2.27	1.53	1.34	1.14	0.91

Figure 4.14: Heuristic vs Uniform: CIFAR-10 FP32, int8, int16 tflite size

```
Conv2d_3/pointwise:    16384 weights,  13926 zero weights,  0.8499755859375
    sparsity
Conv2d_4/pointwise:    32768 weights,  27853 zero weights,  0.850006103515625
    sparsity
Conv2d_5/pointwise:    65536 weights,  55706 zero weights,  0.850006103515625
    sparsity
Conv2d_6/pointwise:    131072 weights, 111411 zero weights,
    0.8499984741210938 sparsity
Conv2d_7/pointwise:    262144 weights, 222822 zero weights,
    0.8499984741210938 sparsity
Conv2d_8/pointwise:    262144 weights, 222822 zero weights,
    0.8499984741210938 sparsity
Conv2d_9/pointwise:    262144 weights, 222822 zero weights,
    0.8499984741210938 sparsity
Conv2d_10/pointwise:   262144 weights, 222822 zero weights,
    0.8499984741210938 sparsity
Conv2d_11/pointwise:   262144 weights, 222822 zero weights,
    0.8499984741210938 sparsity
Conv2d_12/pointwise:   524288 weights, 445645 zero weights,
    0.8500003814697266 sparsity
Conv2d_13/pointwise:   1048576 weights,891290 zero weights,
    0.8500003814697266 sparsity
top/Conv2d_1x1_output: 10250 weights,  8704 zero weights,   0.849170731707317
    sparsity
Sparse weights: 2678083
All weights: 3150698
Overall model sparsity: 0.8499967309
```

Listing 4.2: MobileNet v1 and CIFAR-10: uniform weights distributions

What the Figure 4.14 highlights though is that quantized models have a much smaller size compared to FP32.

As reference, the uncompressed sizes of the tflite files are:

- **FP32**: 12.85Mb
- **int8**: 3.61Mb
- **int16**: 3.65Mb

At 0.85 sparsity level, there are the following gains in terms of sizes:

- **FP32**: 12.85Mb → 3.30Mb, a **decrease of 74.31%**
- **int8**: 3.61Mb → 1.11Mb, a **decrease of 69.25%**
- **int16**: 3.65Mb → 1.13, a **decrease of 69.04%**

As mentioned in section 4.3.3, vela compiler has a compression mechanism built it when parsing a tflite file. In fact, vela generates a new tflite file which has a better compression rate compared to gzip. Let's take the above sparsity level (0.85) and see how vela compiler affects the tflite size. Of course, this is true **only for quantized tflite models.** The non-pruned quantized models (both int8 and int16) have a **vela size of 3Mb.**. The pruned ones have a **vela size of 0.91Mb**, a **decrease of 69.67%**.

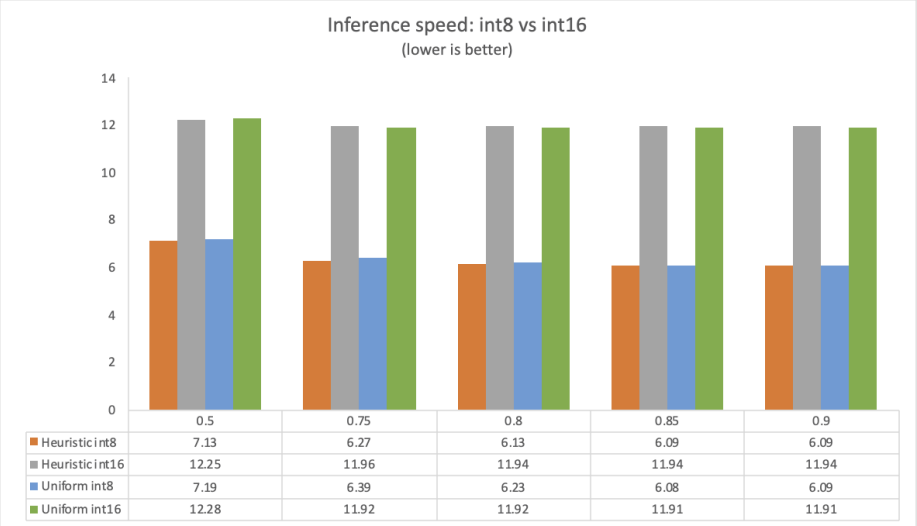
Let's take the 2 extreme points of the tflite sizes: FP32 non-pruned and int8/int16 pruned at 0.85. **The tflite size goes from 12.85 to 0.91Mb, a decrease of 92.92%**. The accuracy goes from 0.9487 to 0.9205 (int8) or 0.9243 (int16), **a decrease of 2.97% (int8) or 2.57% (int16)**

Inference speed results (int8 and int16 only)

The inference speed showed in Figure 4.15 are generated by vela while optimising the tflite file. Vela compiler can take multiple profiles depending for what NPU it needs to optimise the tflite for.

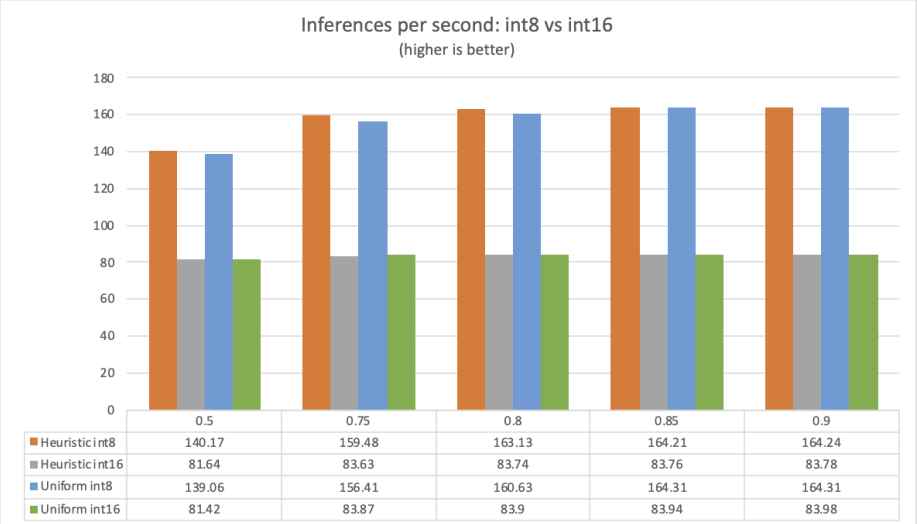
In this case it has been optimised for an Ethos-U55 high-end embedded NPU with the following characteristics: SRAM, 4 GB/s and flash, 0.5 GB/s. Listing 4.3 shows vela output with estimations. At the very end there is the inference speed (`Batch Inference time`)

```
Warning: Using internal-default values for memory mode
Info: MEAN model_2/top/GlobalPool/Mean is a CPU only op
Warning: Mean operation is unknown or unsupported, placing on CPU

Network summary for mobilenet-v1_cifar10-int8-pruned
Accelerator configuration               Ethos_U55_256
System configuration             Ethos_U55_High_End_Embedded
Memory mode                          internal-default
Accelerator clock                                500 MHz
Design peak SRAM bandwidth                      4.00 GB/s
Design peak Off-chip Flash bandwidth            0.50 GB/s

Total SRAM used                               415.64 KiB
Total Off-chip Flash used                     899.83 KiB (2.20 bits per
    element)

69 passes fused into 58
1/191 (52.4%) operations falling back to the CPU
Average SRAM bandwidth                          2.82 GB/s
Input   SRAM bandwidth                          7.71 MB/batch
Weight  SRAM bandwidth                          6.19 MB/batch
Output  SRAM bandwidth                          3.25 MB/batch
Total   SRAM bandwidth                         17.19 MB/batch
Total   SRAM bandwidth            per input     17.19 MB/inference (batch
    size 1)

Average Off-chip Flash bandwidth                0.14 GB/s
Input   Off-chip Flash bandwidth                0.11 MB/batch
Weight  Off-chip Flash bandwidth                0.77 MB/batch
Output  Off-chip Flash bandwidth                0.00 MB/batch
Total   Off-chip Flash bandwidth                0.88 MB/batch
Total   Off-chip Flash bandwidth  per input      0.88 MB/inference (batch
    size 1)

Neural network macs                          611565578 MACs/batch
```



	0.5	0.75	0.8	0.85	0.9
Heuristic int8	7.13	6.27	6.13	6.09	6.09
Heuristic int16	12.25	11.96	11.94	11.94	11.94
Uniform int8	7.19	6.39	6.23	6.08	6.09
Uniform int16	12.28	11.92	11.92	11.91	11.91

(a) Inference speed



	0.5	0.75	0.8	0.85	0.9
Heuristic int8	140.17	159.48	163.13	164.21	164.24
Heuristic int16	81.64	83.63	83.74	83.76	83.78
Uniform int8	139.06	156.41	160.63	164.31	164.31
Uniform int16	81.42	83.87	83.9	83.94	83.98

(b) Inferences per second

Figure 4.15: Heuristic vs Uniform: CIFAR-10 int8, int16 inference speed

```
Hardware macs                                  668484736 MACs/batch
Network Tops/s                                       0.20 Tops/s
Hardware Tops/s                                      0.22 Tops/s

NPU cycles                                       2967787 cycles/batch
SRAM Access cycles                               2148562 cycles/batch
DRAM Access cycles                                     0 cycles/batch
On-chip Flash Access cycles                            0 cycles/batch
Off-chip Flash Access cycles                      881088 cycles/batch
Total cycles                                     3044776 cycles/batch

Batch Inference time                 6.09 ms,   164.22 inferences/s (batch
    size 1)
```

Listing 4.3: MobileNet v1 and CIFAR-10: vela output on an int8 tflite file

Similarly, to what has been observed in section 4.3.4, the distribution doesn't really impact the inference speed. It looks like though that the heuristic distribution is slightly faster than the uniform one, but the increase is so small that can be ignored.

Figure 4.15 shows anyway that int16 is much slower than int8: this because the native MACs (multiply-accumulate) operations in Ethos-U55 are 8 (for weights) $\times$ 8 (for activations) bit. 8$\times$16-bit operations run at half the speed of 8$\times$8-bit operations as an 8$\times$16 MAC is twice as complex as an 8$\times$8 MAC.

Let's take the inference time for non-pruned tflite files and see what kind of speed up there is with a pruned model at 0.85.

- **int8**: 8.68ms $\rightarrow$ 6.09ms, a **decrease of 29.84%**
- **int16**: 13.93ms $\rightarrow$ 11.91ms, a **decrease of 14.50%**

Quantization has a huge benefit in terms of tflite size and latency of the inference and the benefits are even bigger when paired up with pruning.

4.3.5 Results MobileNet v1 with ImageNet 2012

After seen the results using CIFAR-10, in this section the dataset used is ILSVRC (ImageNet Large Scale Visual Recognition Challenge) 2012, commonly known as ImageNet.

There are no architectural changes to the model.

The fine-tuning of the model has been done with the following hyper parameters found via experimentations:

- **Optimizer**: SGD (Stochastic Gradient Descend)
- **Optimizer momentum**: 0.9
- **Optimizer decay rate**: 0.9
- **Learning rate decay rate**: 0.94
- **Learning rate decay epochs**: 2.5
- **Dropout rate**: 0.2
- **Standard weight decay**: 4e-05
- **Truncated normal standard deviation**: 0.09

- **Batch norm decay**: 0.9997
- **Batch size**: 64
- **Learning rate**: 0.001
- **Epochs**: 20
- **Sparsity scheduler**: Polynomial Decay

Accuracy and loss results

For ImageNet I focus mainly on the FP32 results as similar considerations to CIFAR-10 can be done for quantized models in terms of accuracy.

Like before, the heuristic distribution will be always compared to the uniform distribution.

For reference, **the TOP-1 accuracy of a non-pruned MobileNet v1 with ImageNet is 0.709.**

Figure 4.16 shows how the two distributions impact the TOP-1/TOP-5 accuracies and the loss function.

Like in the previous case, the heuristic distribution always outperforms the uniform distribution both for TOP-1 (blue line) and TOP-5 (grey line). The heuristic distribution seems more a little bit more robust compared to the uniform one as the sparsity increases the difference between the two sparsity increases slightly. When the sparsity is set to 0.9, the heuristic distribution has an increase of **1.02% in the TOP-1** compared to the uniform distribution. **Any sparsity bigger than 0.5 is going to have an impact in terms of accuracy**: the TOP-1 accuracy for a non-pruned model is 0.709 and for a sparsity of 0.5 there is a TOP-1 accuracy of 0.7021 for the heuristic distribution and 0.7011 for the uniform distribution. If the requirements of the application need to have a close accuracy of the original model, then 0.5 is a pretty good sparsity level. At this point choosing one of the other distribution changes very little in terms of accuracy: the heuristic has only **0.1041%** more accuracy than the uniform distribution. If the application requires a smaller model, it needs to compromise in the accuracy: from 0.75 to 0.85 sparsity level, there is a drop of about 6%. At 0.85 sparsity level, the heuristic distribution has an accuracy of **0.6018** whilst the uniform distribution falls in the range below, at **0.5927**. At 0.9 sparsity level there is a steeper drop in accuracy (around 0.52): with this accuracy it might not make any sense to deploy the model at all. If the sparsity needs to be between 0.5 and 0.85 then the heuristic distribution is the one of choice as it gives better accuracy.

The loss follows has a specular behaviour of the TOP-1 accuracy: at 0.5 sparsity level the loss function is the same for both distributions. From 0.75 to 0.85 they increase almost in a parallel fashion and to get to 0.9 were both distributions have a steeper increase. Anyway from 0.75 to 0.9 the heuristic distribution (blue line) has always a smaller loss function compared to the uniform. This is another proof that the heuristic distribution seems more robust at higher sparsity levels.

Figure 4.16: Heuristic vs Uniform: ImageNet TOP-1/TOP-5 and Loss

tflite size results

Like in the previous case, the sparsity distribution doesn't really influence the size of the tflite: the number of weights is the same between the uniform and the heuristic distributions.

This model though is bigger than the previous one: even the architecture is the same, it deals with different input size: CIFAR-10 images are 32×32 whilst ImageNet images are 224×224.

Let's take the 0.85 sparsity level and see how the weights are distributed following the heuristic distribution.

```
2021-02-19 15:26:23,484 Conv2d_0_0:            864 weights,     377 zero weights,    0.4363425925925926 sparsity
2021-02-19 15:26:23,488 Conv2d_1/pointwise:    2048 weights,    1008 zero weights,   0.4921875 sparsity
2021-02-19 15:26:23,492 Conv2d_2/pointwise:    8192 weights,    4765 zero weights,   0.5816650390625 sparsity
2021-02-19 15:26:23,496 Conv2d_3/pointwise:    16384 weights,   10264 zero weights,  0.62646484375 sparsity
2021-02-19 15:26:23,500 Conv2d_4/pointwise:    32768 weights,   21993 zero weights,  0.671173095703125 sparsity
2021-02-19 15:26:23,504 Conv2d_5/pointwise:    65536 weights,   46919 zero weights,  0.7159271240234375 sparsity
2021-02-19 15:26:23,508 Conv2d_6/pointwise:    131072 weights,  99704 zero weights,  0.76068115234375 sparsity
2021-02-19 15:26:23,512 Conv2d_7/pointwise:    262144 weights,  211138 zero weights, 0.8054275512695312 sparsity
2021-02-19 15:26:23,517 Conv2d_8/pointwise:    262144 weights,  211138 zero weights, 0.8054275512695312 sparsity
2021-02-19 15:26:23,522 Conv2d_9/pointwise:    262144 weights,  211138 zero weights, 0.8054275512695312 sparsity
2021-02-19 15:26:23,526 Conv2d_10/pointwise:   262144 weights,  211138 zero weights, 0.8054275512695312 sparsity
2021-02-19 15:26:23,531 Conv2d_11/pointwise:   262144 weights,  211138 zero weights, 0.8054275512695312 sparsity
2021-02-19 15:26:23,536 Conv2d_12/pointwise:   524288 weights,  445735 zero weights, 0.8501720428466797 sparsity
2021-02-19 15:26:23,543 Conv2d_13/pointwise:   1048576 weights, 938389 zero weights, 0.8949174880981445 sparsity
2021-02-19 15:26:23,547 top/Conv2d_1x1_output: 1026025 weights, 915809 zero weights, 0.8925796155064448 sparsity
2021-02-19 15:26:23,547 Sparse weights: 3540653
2021-02-19 15:26:23,547 All weights: 4166473
2021-02-19 15:26:23,547 Overall model sparsity: 0.849796218528237
```

Listing 4.4: MobileNet v1 and ImageNet: heuristic weights distributions

As shown in Listing 4.4, the heuristic distribution of the weights has set different sparsity levels for every layer depending on how many weights a layer has.

As reference, the uncompressed sizes of the tflite files are:

- **FP32**: 16.91Mb
- **int8**: 4.66Mb
- **int16**: 4.71Mb

At 0.85 sparsity level, there are the following gains in terms of sizes:

- **FP32**: 16.91Mb → 4.07Mb, a **decrease of 75.93%**
- **int8**: 4.66Mb → 1.43Mb, a **decrease of 69.31%**
- **int16**: 4.71Mb → 1.13, a **decrease of 76.01%**

The non-pruned quantized models (both int8 and int16) have a **vela size of 3.96Mb.**. The pruned ones have a **vela size of 1.16Mb**, a **decrease of 71.46%**.

Let's take the 2 extreme points of the tflite sizes: FP32 non-pruned and int8/int16 pruned at 0.85. **The tflite size goes from 16.91 to 1.16Mb, a decrease of 93.14%**.

Inference speed results (int8 and int16 only)

Like in the previous, I have run vela for every experiment and the same conclusions can be made: the distribution doesn't really impact the inference speed. It looks like though that the heuristic distribution is slightly faster than the uniform one, but the increase is so small that can be ignored.

Let's take the inference time for non-pruned tflite files and see what kind of speed up there is with a pruned model at 0.85 (as shown in Listing 4.5)

```
Info: MEAN model_2/top/GlobalPool/Mean is a CPU only op
Warning: Mean operation is unknown or unsupported, placing on CPU

Network summary for mobilenet-v1_imagenet2012-int8-pruned
Accelerator configuration               Ethos_U55_256
System configuration             Ethos_U55_High_End_Embedded
Memory mode                          internal-default
Accelerator clock                                500 MHz
Design peak SRAM bandwidth                      4.00 GB/s
Design peak Off-chip Flash bandwidth            0.50 GB/s

Total SRAM used                               599.80 KiB
Total Off-chip Flash used                    1130.38 KiB (2.11 bits per
    element)

69 passes fused into 63
1/187 (53.5%) operations falling back to the CPU
Average SRAM bandwidth                          2.69 GB/s
Input   SRAM bandwidth                          9.77 MB/batch
Weight  SRAM bandwidth                          6.40 MB/batch
Output  SRAM bandwidth                          5.06 MB/batch
Total   SRAM bandwidth                         21.24 MB/batch
Total   SRAM bandwidth            per input     21.24 MB/inference (batch
    size 1)

Average Off-chip Flash bandwidth                0.14 GB/s
Input   Off-chip Flash bandwidth                0.12 MB/batch
Weight  Off-chip Flash bandwidth                0.99 MB/batch
Output  Off-chip Flash bandwidth                0.00 MB/batch
Total   Off-chip Flash bandwidth                1.12 MB/batch
Total   Off-chip Flash bandwidth  per input      1.12 MB/inference (batch
    size 1)

Neural network macs                         568749545 MACs/batch
Hardware macs                               677312576 MACs/batch
Network Tops/s                                   0.14 Tops/s
Hardware Tops/s                                  0.17 Tops/s

NPU cycles                                    3812208 cycles/batch
SRAM Access cycles                            2655593 cycles/batch
DRAM Access cycles                                  0 cycles/batch
On-chip Flash Access cycles                         0 cycles/batch
Off-chip Flash Access cycles                  1115488 cycles/batch
Total cycles                                  3946734 cycles/batch

Batch Inference time                 7.89 ms,  126.69 inferences/s (batch
    size 1)
```

Listing 4.5: MobileNet v1 and ImageNet: vela output on an int8 tflite file

- **int8**: 11.84ms → 7.89ms, a **decrease of 33.36%**

- **int16**: 17.71ms → 14.73ms, a **decrease of 16.83%**

These measurements are anyway bigger than the previous case because the model generated is bigger hence it needs more time to be executed by the NPU.

5 Conclusions

Pruning is an excellent technique of model optimization: it allows to dramatically decrease the size of a model with no or low impact in accuracy.

In this thesis I have demonstrated the benefits of the heuristic distribution of weights while pruning a model: layers with mode weights can be pruned more compared to layers with less weights while maintaining the final sparsity specified by the user.

To further optimise the model, quantization can be applied: the non-uniform distribution of the weights though doesn't affect the quantization process.

The heuristic distribution results more robust compared to the uniform distribution of weights: in MobileNet v1, for both CIFAR-10 and ImageNet the heuristic distribution of weights always outperforms the uniform distribution. This is true from a sparsity level of 0.5 up to 0.9: especially in the CIFAR-10 case, this difference increases with higher sparsity levels giving a difference in accuracy of 2.14%.

The work in this thesis represents a solid base for enabling non-uniform distributions of weights in TensorFlow Model Optimization.

As next steps, I have identified the following:

- **Upstream code to GitHub**: using this thesis as starting point, create an RFC proposal and engage with Google to receive feedback. Once details have been agreed, a pull request in GitHub can be raised in order to be merged in TFMOT. As part of the pull request, user documentation, testing (unit tests and end-to-end tests) need to be written.
- **Provide more off-the-shelf heuristics**: the architecture of the code has been designed in a way to be as generic as possible: the user can implement his/her own distributions of weights and pass it to the pruning parameters. As extra step, more off-the-shelf heuristic distributions can be offered to the user: Erdős-Rényi and Erdős-Rényi-Kernel (ERK) [5]
- **Expand tests on new architecture/datasets**: this thesis focussed only on MobileNet v1 architecture with CIFAR-10 and ImageNet datasets. In order to further prove the benefits of the heuristic distribution of weights, tests on more comprehensive set of architectures and datasets need to be performed.

Appendices

A

Content of `distribution.py`

```
1 # The implementation code is not under public domain yet.
```

Listing A.1: Content of distribution.py

B

Content of `prune.py`

```
1 # The implementation code is not under public domain yet.
```

Listing B.1: Content of prune.py

C

MNIST pipeline with heuristic pruning

```
1 # The implementation code is not under public domain yet.
```

Listing C.1: Content of full_heuristic_pruning_mnist.py

Bibliography

[1] Mohamed Nour Abouelseoud and Anton Kachatkou. Weight clustering api. URL: https://blog.tensorflow.org/2020/08/tensorflow-model-optimization-toolkit-weight-clustering-api.html.

[2] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems*, 13(3):1–18, May 2017. URL: http://dx.doi.org/10.1145/3005348, doi:10.1145/3005348.

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020. arXiv:2003.03033.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *Proceedings of Machine Learning and Systems 2020*, pages 471–481. -, 2020.

[6] Google. Filter out disruptive noise in google meet. URL: https://workspaceupdates.googleblog.com/2020/06/remove-background-noise-google-meet.html.

[7] Google. Post-training quantization. URL: https://www.tensorflow.org/model_optimization/guide/quantization/post_training.

[8] Google. Quantization aware training. URL: https://www.tensorflow.org/model_optimization/guide/quantization/training.

[9] Google. Tensorflow model optimization. URL: https://www.tensorflow.org/model_optimization/guide.

[10] Google. Trim insignificant weights. URL: https://www.tensorflow.org/model_optimization/guide/pruning.

[11] Google. Weight clustering. URL: https://www.tensorflow.org/model_optimization/guide/clustering.

[12] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015. arXiv:1510.00149.

[13] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015. arXiv:1506.02626.

[14] B. Hassibi, D. G. Stork, and G. J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pages 293–299 vol.1, 1993. doi:10.1109/ICNN.1993.298572.

[15] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. *Lecture Notes in Computer Science*, page 815–832, 2018. URL: http://dx.doi.org/10.1007/978-3-030-01234-2_48, doi:10.1007/978-3-030-01234-2_48.

[16] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017. URL: http://dx.doi.org/10.1109/ICCV.2017.155, doi:10.1109/iccv.2017.155.

[17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. arXiv:1704.04861.

[18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun 2018. URL: http://dx.doi.org/10.1109/CVPR.2018.00286, doi:10.1109/cvpr.2018.00286.

[19] Renu Khandelwal. A basic introduction to tensorflow lite. URL: https://towardsdatascience.com/a-basic-introduction-to-tensorflow-lite-59e480c57292.

[20] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[21] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605. Morgan-Kaufmann, 1990. URL: https://proceedings.neurips.cc/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

[22] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning, 2018. arXiv:1810.05270.

[23] Arm Ltd. Ethos-n78. URL: https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-n78.

[24] Arm Ltd. Ethos-u65. URL: https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u65.

[25] Arm Ltd. The vela compiler. URL: https://developer.arm.com/documentation/101888/0500/NPU-software-overview/NPU-software-tooling/The-Vela-compiler?lang=en.

[26] Arun Mohan. Review on mobilenet v1. URL: https://medium.com/datadriveninvestor/review-on-mobilenet-v1-abec7888f438.

[27] Sayak Paul. Plunging into model pruning in deep learning. URL: https://wandb.ai/authors/pruning/reports/Plunging-Into-Model-Pruning-in-Deep-Learning--VmlldzoxMzcyMDg.

[28] TensorFlow Model Optimization team. Quantization aware training with tensorflow model optimization toolkit - performance with accuracy. URL: https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html.

[29] Yulong Wang, Xiaolu Zhang, Lingxi Xie, Jun Zhou, Hang Su, Bo Zhang, and Xiaolin Hu. Pruning from scratch. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):12273–12280, Apr 2020. URL: http://dx.doi.org/10.1609/AAAI.V34I07.6910, doi:10.1609/aaai.v34i07.6910.

[30] Wikipedia. Convolutional neural network. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network#Building_blocks.

[31] Wikipedia. Heuristic. URL: https://en.wikipedia.org/wiki/Heuristic.

[32] Jiaxiang Wu, Yao Zhang, Haoli Bai, Huasong Zhong, Jinlong Hou, Wei Liu, and Junzhou Huang. Pocketflow: An automated framework for compressing and accelerating deep neural networks. In *Advances in Neural Information Processing Systems (NIPS), Workshop on Compact Deep Neural Networks with Industrial Applications*. 2018.

[33] Dianlei Xu, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. Edge intelligence: Architectures, challenges, and applications, 2020. arXiv:2003.12172.

About this document

Proudly written in LaTeX and Vim.

Source code of this document can be found https://github.com/diegorusso/master-degree-thesis