Inference at the edge: Tuning Compression Parameters for Performance

Dissertation

BSc Computer Science: Artificial Intelligence

Sam Fay-Hunt — sf52@hw.ac.uk

Supervisor: Rob Stewart — R.Stewart@hw.ac.uk

April 21, 2021

DECLARATION

I, Sam Fay-Hunt confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: .....Sam Fay-Hunt.....

Date: .....21/04/2020.....

Abstract: *Abstract here*

Contents

Chapter 1

Introduction

1.1 Motivation

With the continued revolution of AI technologies a desire to utilise the power of neural networks beyond the traditional datacentre bound systems is becoming ever more prevalent. Using neural networks locally as opposed to in a datacentre (in other words at the *edge*) is already prevalent in modern life, to name a few; mobile phones, voice assistants and even our televisions are making use of this technology. Training neural networks is a costly, often slow, process, usually completely invisible to users of a network, so it makes sense to keep training in the datacentre where resources are abundant, however applying the knowledge from a trained neural network (referred to as *inference*) on the other hand has a myriad of advantages when it is performed at the edge.

Keeping data used for inference local to the device that produces this data means we can rely on a consistent inference latency, in particular computer vision applications such as self-driving cars or autonomous manufacturing machines tend to require a consistent low latency and as such cannot rely on cloud services to perform inference. Additional benefits include privacy and security; reducing data sent to datacentres helps mitigate the potential for a major data leak since no data needs to leave the device.

Computing at the edge can be limiting in terms of available memory space or performance capabilities, networks relying on real time data tend to require lower response times, but there is often a locality vs performance trade-off unless expensive custom solutions are used. GPUs are the most widely used piece of hardware to operate a neural network, however they are often physically large

and can consume a considerable amount of power, an alternative to the GPU is an AI accelerator such as the Intel Neural Compute Stick, or the Nvidia Jetson, these accelerators are purpose built for neural networks, usually have a much smaller form factor and consume considerably less power.

Pruning is a compression technique that involves removing weights from a network with the goal of making it smaller, intuitively we might think that a network with a smaller memory footprint would naturally have lower inference latency, but it is often not this simple. Utilising neural network compression requires expert level knowledge of not only the network structure but the consequences of compression because compression techniques such as pruning can have cascading effects throughout a neural network. This alone can make compression a daunting task, even for experienced machine learning practitioners, it gets worse however, these compression algorithms often feature complex parameters with implications that may not be revealed until a substantial amount of time has been invested in retraining a compressed model.

The most apparent reasons to prune a neural network is either to reduce the size of the neural network or its latency, this dissertation will focus on reducing the latency aspect, starting from a readily available off-the-shelf configuration for a neural network we will attempt to automate the process of optimising compression parameters with the goal of reducing latency, ideally with a minimal loss of accuracy.

1.2 Hypothesis

Using optimisation algorithm we can automate compression parameter selection and improve inference latency in a typical edge computing environment.

1.3 Research Aims

Aim 1 — This dissertation will research a methodology for reducing inference latency using off-the-shelf compression techniques as a starting point.

Aim 2 — We will investigate to what degree different parts of the network structure impact inference latency.

1.4 Objectives

- **O0:** Verify the functionality of distiller, check that the pruning methods available function as described in the literature.
- **O1:** Develop a pipeline to compress and benchmark a neural network model.
- **O2:** Select at least 1 neural network model to use for testing.
- **O3:** Select an appropriate compression algorithm for the goal of reducing latency.
- **O4:** Identify an appropriate off-the-shelf set of hyperparameters for the selected model.
- **05:** Establish baseline measurements in terms of accuracy and latency for both the unpruned and off-the-shelf (pruned) models.
- **O6:** Automate compression parameter optimisation by leveraging the pipeline from O1 to test combinations of parameters and empirically evaluate their impact on a target metric (such as latency or accuracy).
- **O7:** Evaluate how well the automated system optimizes the compression parameters, by comparing with the baseline and off the shelf measurements.

Chapter 2

Background

- *Adapt from D1*
- *rewrite with more of a focus on the concrete channel and pruning methodology used*
- *Would be good to include wandb bayes hyperparam optimisation details*
- *Discuss filters and channels in more depth*

This Chapter is split into 4 sections:

Section **??** — **Deep Learning**: An overview of the basic components of a deep neural network and the CNN model.

Section **??** — **Neural Network Compression**: Discusses neural network compression techniques and on how they change the underlying representations of DNNs.

Section **??** — **AI accelerators** Covers a few popular AI accelerators architectures, their strengths, weaknesses and specialisms.

Section **??** — **Memory factors for Deep Neural Networks**: Describes how DNNs interact with memory, and discusses some of the implications of this.

2.1 Deep Neural Networks

2.1.1 Neural Networks & Deep Learning

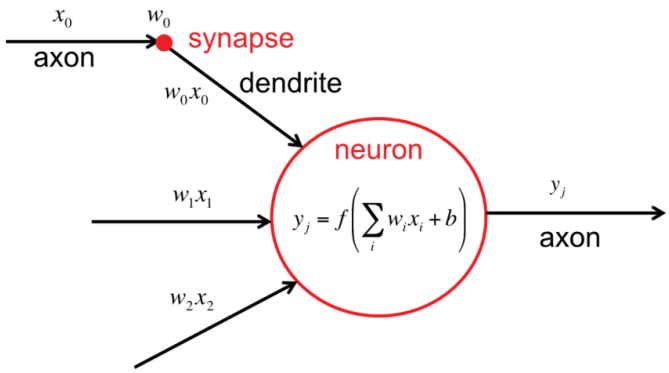

Figure 2.1: Neuron with corresponding biologically inspired labels.
(Adopted figure from [szeEfficientProcessingDeep2017])

Deep learning is a subcategory of machine learning techniques where a hierarchy of layers perform some manner of information processsing with the goal of computing high level abstractions of the data by utilising low level abstractions identified in the early layers [**dengTutorialSurveyArchitectures2014**].

Neural networks fundamental purpose is to transform an input vector commonly referred to as X into an output vector $\hat{Y}$. The output vector $\hat{Y}$ is some form of classification such as a binary classification or a probability distribution over multiple classes [**thierry-miegHowFundamentalConcepts**]. Between the input layer (X) and the output layer ($\hat{Y}$) there exists some number of interior layers that are referred to as hidden layers, the hidden and output layers are composed of neurons that pass signals derived from weights through the network, this model of computing was inspired by connectionism and our understanding of the human brain, see Fig. **??** for labels of the analagous biological components. Weights in a neural network effectively correspond to the synapses in the brain and the output of the neruon is modelled as the axon. All neruons in a Neural network have weights corresponding to their inputs, these weights are are intended to mirror the value scaling effect of a synapse by performing a weighted sum operation [**szeEfficientProcessingDeep2017**].

Neural networks and deep neural networks are often reffered to interchangably, they are primarily distinguished by the number of layers, there is no hard rule indicating when a neural network is considered deep but generally a network with more than 3 hidden layers is considered a

deep neural network, the rest of this dissertaion will refer to DNNs for consistency. Each neuron in a DNN applies an non-linear activation function to the result of its weighted sum of inputs and randomly initialised weights, without which a DNN would just be a linear algebra operation [**szeEfficientProcessingDeep2017**], the cumulative effect of the activations in each layer results in elabourate causal chains of transormations that influence the aggregate activation of the network.

2.1.2 Inference and Training

Training or learning in the context of DNNs is the process of finding the optimal parameters (value for the weights and bias) in the network. Upon completion of training *inference* can be performed, this is where new input data is fed into the network, a series of operations is performed using the trained parameters, and some meaninful output is obtained such as a classification, regression, or function approximation. Many techniques can be used to search for optimal parameters, one example known as supervised learning is as follows: Begin by passing some training data through the network, next the gap between the known ideal output (labels) and the computed outputs from the current weights is calculated using a loss function. Finally the weights are updated using an optimization process such as gradient descent coupled with some form of backward pass, backpropagation is a popular choice for this.

2.1.3 Convolutional Neural Networks

Much like traditional nerual networks the CNN architecture was inspired by human and animal brains, the concept of processing the input with local receptive fields is conceptually similar some functionality of the cat's visual cortex [**hubelReceptiveFieldsBinocular1962**, **lecunConvolutionalNetworksI**
pouyanfarSurveyDeepLearning2019]. The influential paper by Hubel & Weisel [**hubelReceptiveFieldsBino**
ultimately had a significant influence on the design of CNNs via the Neocognitron, as proposed by Fukushima in [**fukushimaNeocognitronSelforganizingNeural1980**] and again evaluated in [**fukushimaNeocognitronHierarchicalNeural1988**], these papers paved the way for the modern CNN.

A critical aspect of image recognition is robustness to input shift and distortion, this robustness was indicated as one of the primary achivements of the Neocognitron in Fukushima's pa-

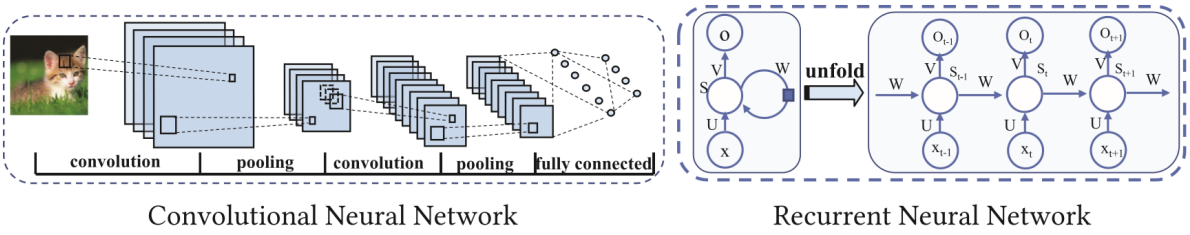

Figure 2.2: A typical example of a CNN (left) and RNN (right)
(Adopted figure from [chenDeepLearningMobile2020])

per [**fukushimaNeocognitronSelforganizingNeural1980**]. LeCunn and Bengio provide comprehensive explainations of how traditional DNNs are so inefficient for these tasks

The local receptive fields enable neurons to extract low level features such as edges, corners, and end-points with respect to their orientation. CNNs are robust to input shift or distortion by using receptive fields to identify these low level features across the entire input space, performing local averaging and downsampling in the layers following convolution layers means the absolute location of the features is less important than the position of the features relative to the position of other identified features [**lecunConvolutionalNetworksImages**]. Each layer produces higher degrees of abstraction from the input layer, in doing so these abstractions retain important information about the input, these abstractions are referred to as feature maps. The layers performing downsampling are known as pooling layers, they reduce the resolution or dimensions of the feature map which reduces overfitting and speeds up training by reducing the number of parameters in the network [**pouyanfarSurveyDeepLearning2019**].

CNNs have been found to be effective in many different AI domains, popular applications include: computer vision, NLP, and speech processing. However they are notorious for needing careful tuning of various hyperparameters, it is often computationally intensive to exhaustively search for optimal CNN hyperparameters, Snoek et al. [**snoekPracticalBayesianOptimization2012**] sucessfully applied a bayesian optimisation algorithm to efficiently search for higher quality hyperparameters.

2.1.4 Filters and Feature Maps

2.2 Neural Network Compression

Neural network compression is necessary due to storage related issues that often arise on resource contrained systems due to the high number of parameters that modern DNNs tend to use, state-of-the-art CNNs can have upwards of hundreds of millions of parameters. Different compression methods can result in variuous underlaying representations of the weight matrices, particularly with respect to its sparsity. Compression techniques that preserve the density of the weight matrix tend to result in inference acceleration on general-purpose processors[**lebedevSpeedingupConvolutionalNeural2015**, **zhangAcceleratingVeryDeep2016**], not all techniques preserve this density and can result in weight matrices with various degrees of sparsity which in turn have varing degrees of regularity. These techniques, the resulting representations of parameters, and their consequences will be discussed in this section.

2.2.1 Pruning

Network pruning is the process of removing unimportant connections, leaving only the most informative connections. Typically pruning is performed by iterating over the following 3 steps: begin by evaluating the importance of parameters, next the least important parameters are pruned, and finally some fine tuning must be performed to recover accuracy. There has been a substantial amount of research into how pruning can be used to reduce overfitting and network complexity [**hansonComparingBiasesMinimal**, **hassibiSecondOrderDerivatives**, **lecunOptimalBrainDamage**, **stromPhonemeProbabilityEstimation1997**], but more recent research shows that some pruning methodologies can produce pruned networks with no loss of accuracy [**hanLearningBothWeights2015**].

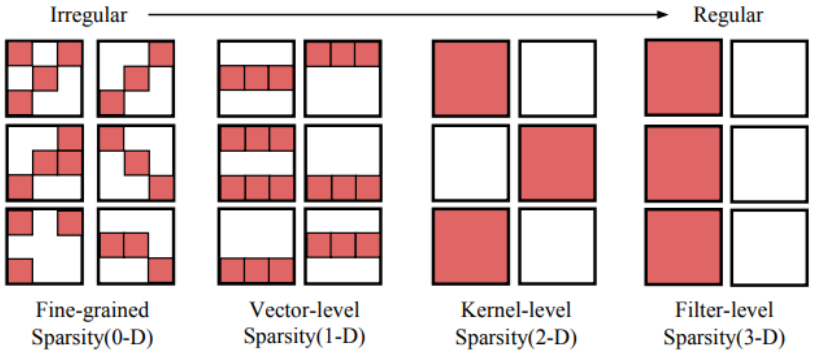

Figure 2.3: Sparse structures in a 4-dimensional weight tensor. Regular sparsity makes hardware acceleration easier.
(Adopted figure from [maoExploringRegularitySparse2017])

This process of pruning the weight matrix within a DNN results in a sparse matrix representation of weights, where the degree of sparsity is determined by the pruning algorithm being used and hyperparameters that can be tuned for the situation, such as how much accuracy loss is considered acceptable, and to what degree the neural network needs to be compressed. The pattern of sparsity in a weight matrix is a fundamental factor when considering how to accelerate a pruned neural network [**maoExploringRegularitySparse2017**], this is known as the *granularity of sparsity*. Figure **??** provides a visual representation of granularity of sparsity, the spectrum of granularity usually falls between either *fine-grained (unstructured)* or *coarse-grained (structured)*, pruning techniques are also categorised by the aformentioned granularities.

The influential paper Optimal Brain Damage by LeCun et al [**lecunOptimalBrainDamage**] was the first to propose a very fine-grained pruning technique by identifying and zeroing individual weights within a network. Fine-grained pruning results in a network that can be challenging to accelerate without custom hardware such as proposed in [**hanEIEEfficientInference2016**, **parasharSCNNAcceleratorCompressedsparse2017**], a software solution has been theorized by Han et al [**hanDeepCompressionCompressing2016**] that would involve developing a customized GPU kernel that supports indirect matrix entry lookup and a relative matrix indexing format, see Section **??** for further details on the necessary steps for this technique.

Coarse-grained pruning techniques such as channel and filter pruning preserve the density of the network by altering the dimensionality of the input/output vectors, channel pruning involves

removing an entire channel in a feature map, filter level pruning likewise removes an entire convolutional filter.

This style of pruning however can have a significant impact on the accuracy of the network, but as demonstrated by Wen et al [**wenLearningStructuredSparsity2016**] accelerating networks with very coarse-grained pruning is straightforward because the model smaller but still dense, so libraries such as BLAS are able to take full advantage of the structure.

2.2.2 Quantisation

Most off-the-shelf DNNs utilise floating-point-quantisation for their parameters, providing arbitrary precision, the cost of this precision can be quite high in terms of arithmetic operation latency, high resource use and higher power consumption. However this arbitrary precision is often unnecesssary, extensive research [**jacobQuantizationTrainingNeural2018**, **maOptimizingLoopOperation2017**] has shown reducing the precision of parameters can have an extremely small impact on the accuracy. Quantisation can be broadly categorised into two groups: non-linear quantisation and fixed-point (linear) quantisation.

Fixed-point quantisation is the process of limiting the floating point precision of each parameter (and potentially each activation) within a network to a fixed point.

In the extreme fixed-point quantisation can represent each parameter with only 1 bit (also known as binary quantisation) with up to a theoretical 32x compression rate (in practice this is often closer to 10.3x) [**chenDeepLearningMobile2020**], Umuroglu et al. [**umurogluFINNFrameworkFast2017**] used binary quantisation with an FPGA and achieved startling classification latencies (0.31μs on the MINIST dataset) while maintaining 95.8% accuracy, this is largely because the entire model can be stored in on-chip memory this is discussed further in Section **??**.

Method	Para.	Speed-up	Top-1 Err. ↑		Top-5 Err. ↑	
			No FT	FT	No FT	FT
CPD	-	3.19×	-	-	0.94%	0.44%
	-	4.52×	-	-	3.20%	1.22%
	-	6.51×	-	-	69.06%	18.63%
GBD	-	3.33×	12.43%	0.11%	-	-
	-	5.00×	21.93%	0.43%	-	-
	-	10.00×	48.33%	1.13%	-	-
Q-CNN	4/64	3.70×	10.55%	1.63%	8.97%	1.37%
	6/64	5.36×	15.93%	2.90%	14.71%	2.27%
	6/128	4.84×	10.62%	1.57%	9.10%	1.28%
	8/128	6.06×	18.84%	2.91%	18.05%	2.66%
Q-CNN (EC)	4/64	3.70×	0.35%	0.20%	0.27%	0.17%
	6/64	5.36×	0.64%	0.39%	0.50%	0.40%
	6/128	4.84×	0.27%	0.11%	0.34%	0.21%
	8/128	6.06×	0.55%	0.33%	0.50%	0.31%

Figure 2.4: Comparison of the speed-up when quantising a convolutional layer in Alexnet, 3 different methods.
(Adopted figure from [wuQuantizedConvolutionalNeural2016])

Non-linear Quantisation is a technique where the weights are split into groups and then assigned a single weight, this grouping can be accomplished in a number of ways, Gong et al. [**gongCompressingDee** used vector quantisation with k-means clustering and achieved compression rates of up to 24x while keeping the difference of top-five accuracy within 1%. Wu et al. [**wuQuantizedConvolutionalNeural2016**] quantised both FC and convolutional layers in Alexnet using their Q-CNN framework

The paper Deep Compression by Han et al [**hanDeepCompressionCompressing2016**] quantisation and weight sharing is taken a step further. First the weights are pruned and quantized, next clustering is employed to gather the quantized weights into bins (whose value is denoted by the centroid of that bin) finally an index is assigned to each weight that points to the weights corresponding bin, the bins value is the centroid of that cluster, which is further fine-tuned by subtracting the sum of the gradients for each weight in the bin their respective centroid see Fig. **??**.

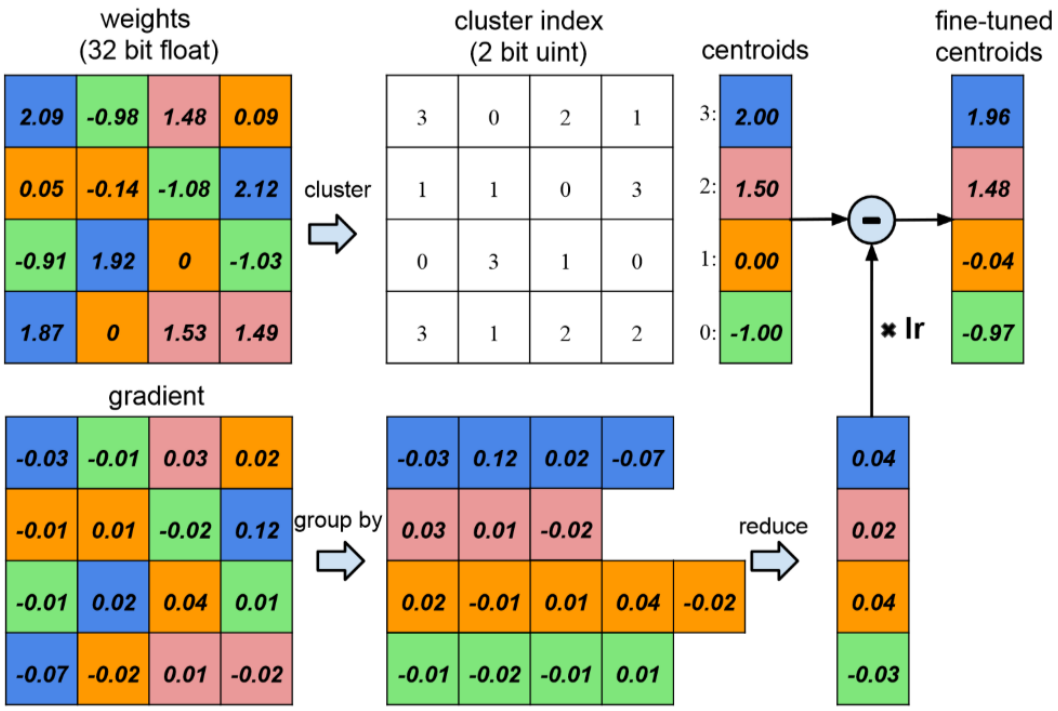

Figure 2.5: Weight sharing by quantisation with centroid fine-tuning using gradients **(Adopted figure from [hanDeepCompressionCompressing2016])**

2.3 AI accelerators

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

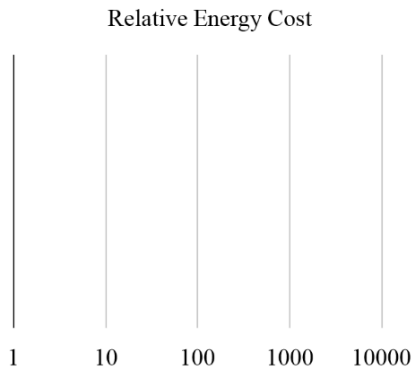

Figure 2.6: Energy table for 45nm CMOS process
(Adopted figure from [hanLearningBothWeights2015])

The increasing popularity of DNNs for classification tasks such as computer vision, speech recognition and natural language processing has promted work to accelerate execution using specialised hardware. AI accelerators tend to prioritise improving the performance of networks from two perspectives; increasing compuational throughput, and decreasing energy consumption. Energy consumption is critical to the feasibility of performing inference on mobile devices, the dominant factor in this area is memory access, figure. **??** shows the energy consumption for a 32 bit floating point add operation and a 32 bit DRAM memory access on a 45nm CMOS chip, they note that DRAM memory access is 3 orders of magnitude of an add operation. Hardware is commonly referred to as an AI accelerator, these can be built to accelerate both the *training* and *inference* stages of execution, this section will specifically focus on the *inference* phase, however many modern accelerators are capable of both.

2.3.1 VPU

One commercial hardware accelerator using a VPU architecture is the Intel Movidius Neural Compute Stick. It is a specialised SoC for computer vision applications, with a peak floating-point computational throughput of 1 TOPS, because of reasons described in Section **??** this peak throughput will be hard to achieve in any real world scenario.

- 16 VLIW (very long instruction word) SHAVE (streaming hybrid architecture vector engine)

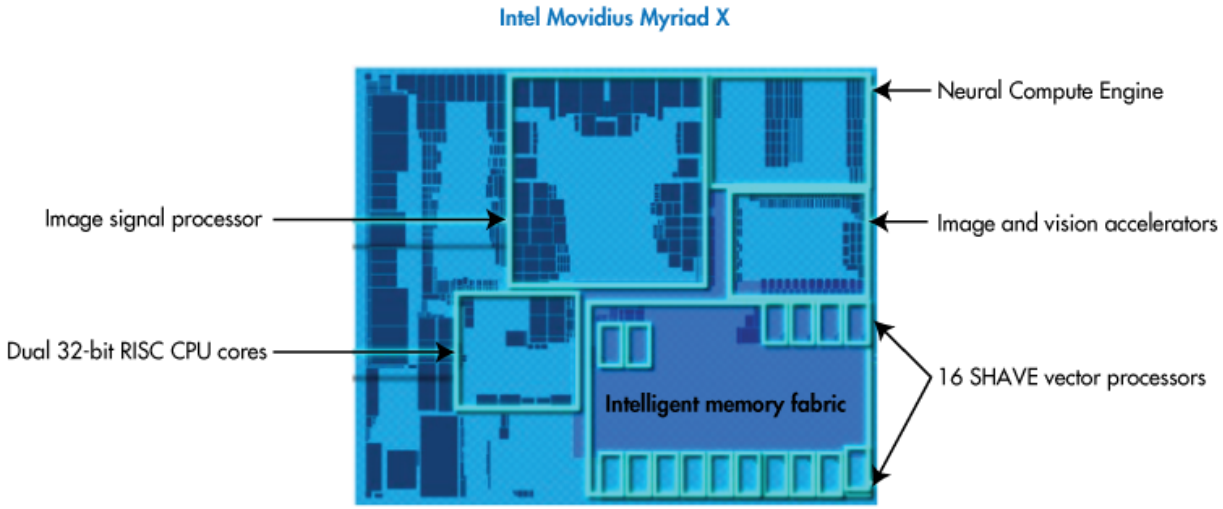

Figure 2.7: High level view of the Intel Movidius Myriad X VPU

processors, optimized for machine vision and able to run parts of a neural network in parallel.

- 2.5 MB On-Chip memory allowing for up to 400GB/s of internal bandwidth.
- 4Gb LPDDR4 DRAM

A key advantage of using hardware like the VPU is a customised computation pipeline that is optimised for high parallelism during inference. This however comes with the caveat that the OpenVINO framework is required to perform inference[**antoniniResourceCharacterisationPersonalScale2019**].

2.3.2 TPU

The TPU is a custom ASIC developed by google, designed specifically for TensorFlow, conventional access to these chips is via a cloud computing service. Google claims [**GoogleWinsMLPerf**] the latest 4th generation TPUv4 is capable of more than double the matrix multiplication TFLOPs of TPUv3 (Wang et al. [**wangBenchmarkingTPUGPU2019**] describes a peak of 420 TFLOPs for the TPUv3). The TPU implements data parallelism in a manner prioritising batch size, one batch of training data is split evenly and sent to each core of the TPU, so total on-board memory determins the maximum data batch size. Each TPU core has a complete copy of the model in memory, so the maximum size of the model is determined by the amount of memory avaliable to each core [**wangBenchmarkingTPUGPU2019**].

2.4 Memory factors for Deep Neural Networks

2.4.1 Memory Allocation

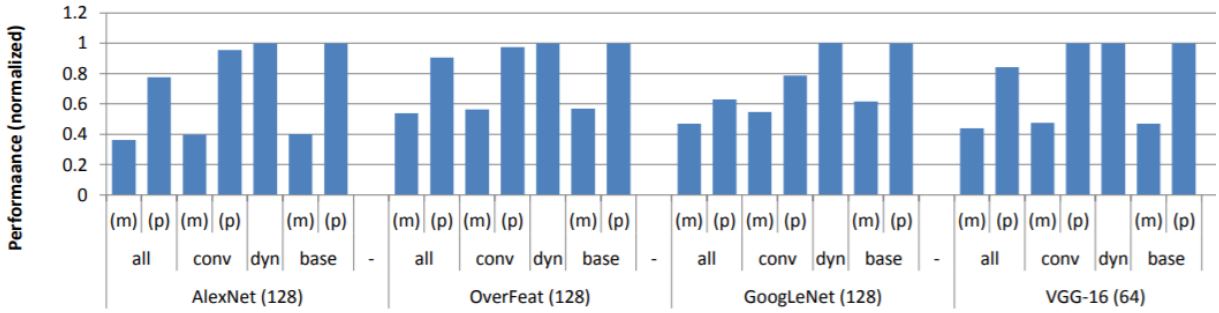

Figure 2.8: vDNN performance, showing the throughput using various memory allocation strategies. **(Adopted figure from [rhuVDNNVirtualizedDeep2016])**

While designed specifically for training networks that would otherwise be to large for a GPU, the memory manager vDNN proposed by Rhu et al [**rhuVDNNVirtualizedDeep2016**] does provide some insight into the importance of memory locality to neural network throughput. Fig. **??** summarizes the performance of neural networks using vDNN to manage memory compared to a baseline memory management policy (*base*). The vDNN policies include: static policies (denoted as *all* and *conv*) and a dynamic policy (*dyn*). *base* simply loads the full model into the GPU memory, consequently providing optimal memory locality. *all* refers to a policy of moving all Xs out of GPU memory, and *conv* only offloads Xs from convolutional layers, Xs are the input matrices to each layer, denoted by the red arrows in Fig. **??**. Each of *base*, *conv* and *all* are evaluated using two distinct convolutional algorithms - memory-optimal (m) and performance-optimal (p). Finally the *dyn* allocation policy chooses (m) and (p) dynamically at runtime.

Observing the results in Fig. **??** where performance is characterized by latency during feature extraction layers; a significant performance loss is evident in the *all* policy compared to baseline, this loss is caused because no effort is made to optimise the location of network parameters in memory. In this example the memory allocations are being measured between memory in the GPU (VRAM) and host memory (DRAM) accessed via the PCI lanes. This does show how important the latency in memory access can be crucial for model throughput.

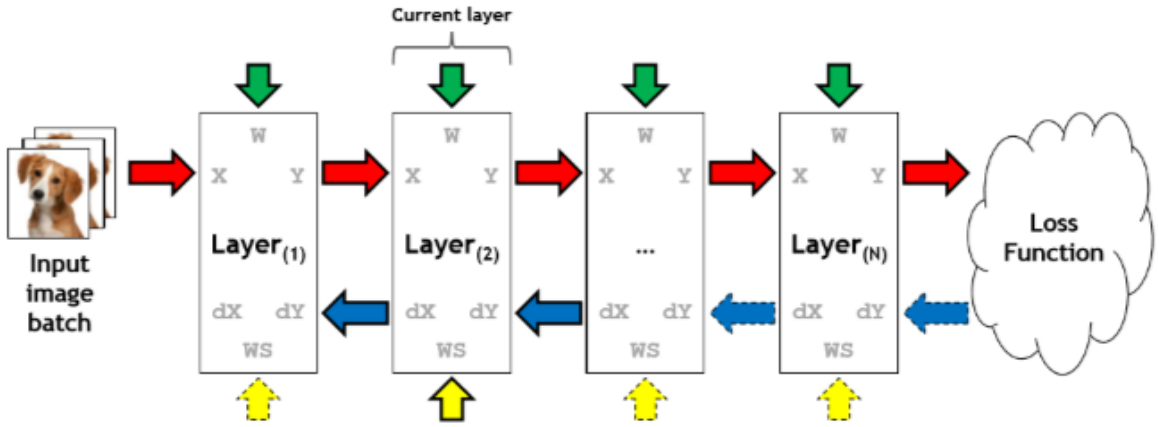

Figure 2.9: Memory allocations required for linear networks. All green (W) and red (X) arrows are allocated during inference, the blue and yellow arrows are allocated during training.
(Adopted figure from [rhuVDNNVirtualizedDeep2016])

2.4.2 Memory Access

A significant portion of DNN computation is matrix-vector multiplication, ideally weight reuse techniques can speed up these operations. However some DNNs feature FC layers with more than a hundred million weights (Fig. **??**), memory bandwidth here can be an issue since loading these weights can be a significant bottleneck [**qiuGoingDeeperEmbedded2016**]. As observed in Section **??** this indicates that compression (Section **??**) techniques could help alliviate this bottleneck by making parameters avaliable for cache reuse.

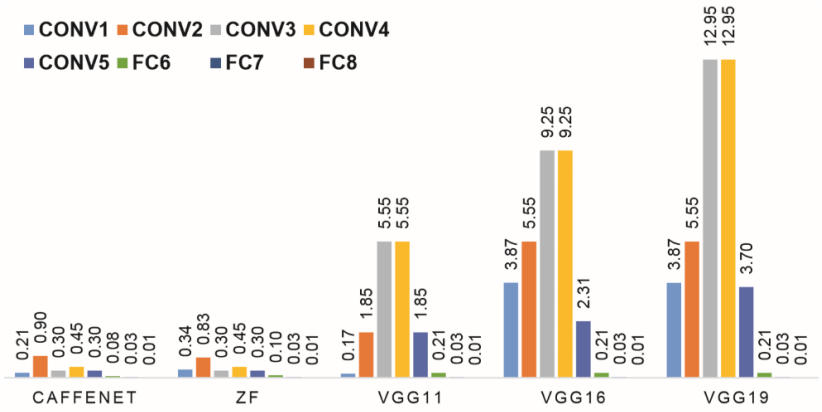

Figure 2.10: Operations demanded in different layers (GOP)
(Adopted figure from [qiuGoingDeeperEmbedded2016])

Often modern networks are so large and complex there can still be an insufficient cache capacity for the full network parameters even when using modern compression techniques such as described in [**hanDeepCompressionCompressing2016**], in a follow up paper Han et al. [**hanEIEEfficientInference2016** discuss this case where memory accesses occur for every operation because the codebook (from a pruned and then quantised network) cannot be reused properly. This paper proposes EIE (an FPGA inference engine for compressed networks) also shows that while compression does reduce the total number of operations, and a tangible speedup can be observed in the FC layers see Fig. **??**, this technique when applied to convolutional layers has some issues.

Han et al [**hanEIEEfficientInference2016**] provide an elegant description of a technique for exploiting the sparity of activations by storing an encoded sparse weight matrix in a variant of compressed sparse column format [**vuducAutomaticPerformanceTuning**], however implementing this is problematic (particularly in convolutional layers) due to the irregular memory access patterns, lack of library and kernel level support for this style of sparse matrix (as discussed in Section **??**). It should also be noted that Fig. **??** is comparing general purpose compute hardware with a custom built FPGA, so the speedup while impressive would be more appropriate compared to other purpose built FPGAs, however the most pertinent part of this Figure is the single batch size FC layer comparison between dense and sparse matrices.

Platform	Batch Size	Matrix Type	AlexNet			VGG16			NT-		
			FC6	FC7	FC8	FC6	FC7	FC8	We	Wd	LSTM
CPU (Core i7-5930k)	1	dense	7516.2	6187.1	1134.9	35022.8	5372.8	774.2	605.0	1361.4	470.5
		sparse	3066.5	1282.1	890.5	3774.3	545.1	777.3	261.2	437.4	260.0
	64	dense	318.4	188.9	45.8	1056.0	188.3	45.7	28.7	69.0	28.8
		sparse	1417.6	682.1	407.7	1780.3	274.9	363.1	117.7	176.4	107.4
GPU (Titan X)	1	dense	541.5	243.0	80.5	1467.8	243.0	80.5	65	90.1	51.9
		sparse	134.8	65.8	54.6	167.0	39.8	48.0	17.7	41.1	18.5
	64	dense	19.8	8.9	5.9	53.6	8.9	5.9	3.2	2.3	2.5
		sparse	94.6	51.5	23.2	121.5	24.4	22.0	10.9	11.0	9.0
mGPU (Tegra K1)	1	dense	12437.2	5765.0	2252.1	35427.0	5544.3	2243.1	1316	2565.5	956.9
		sparse	2879.3	1256.5	837.0	4377.2	626.3	745.1	240.6	570.6	315
	64	dense	1663.6	2056.8	298.0	2001.4	2050.7	483.9	87.8	956.3	95.2
		sparse	4003.9	1372.8	576.7	8024.8	660.2	544.1	236.3	187.7	186.5
EIE	**Theoretical Time**		**28.1**	**11.7**	**8.9**	**28.1**	**7.9**	**7.3**	**5.2**	**13.0**	**6.5**
	Actual Time		**30.3**	**12.2**	**9.9**	**34.4**	**8.7**	**8.4**	**8.0**	**13.9**	**7.5**

Figure 2.11: Wall clock time (μ) comparison for sparse and dense matrices in FC layers between CPU, GPU, mGPU and EIE (an FPGA custom accelerator)
(Adopted figure from [hanEIEEfficientInference2016])

Chapter 3

Methodology

This Chapter will discuss the methodology used to automate the search for lower latency models by tweaking pruning parameters. Section **??** explains how the pipeline was implemented, the discrete parts and how they all fit together. Section **??** discusses the specifics of how we tested the system, the network model used and pruning algorithms selected.

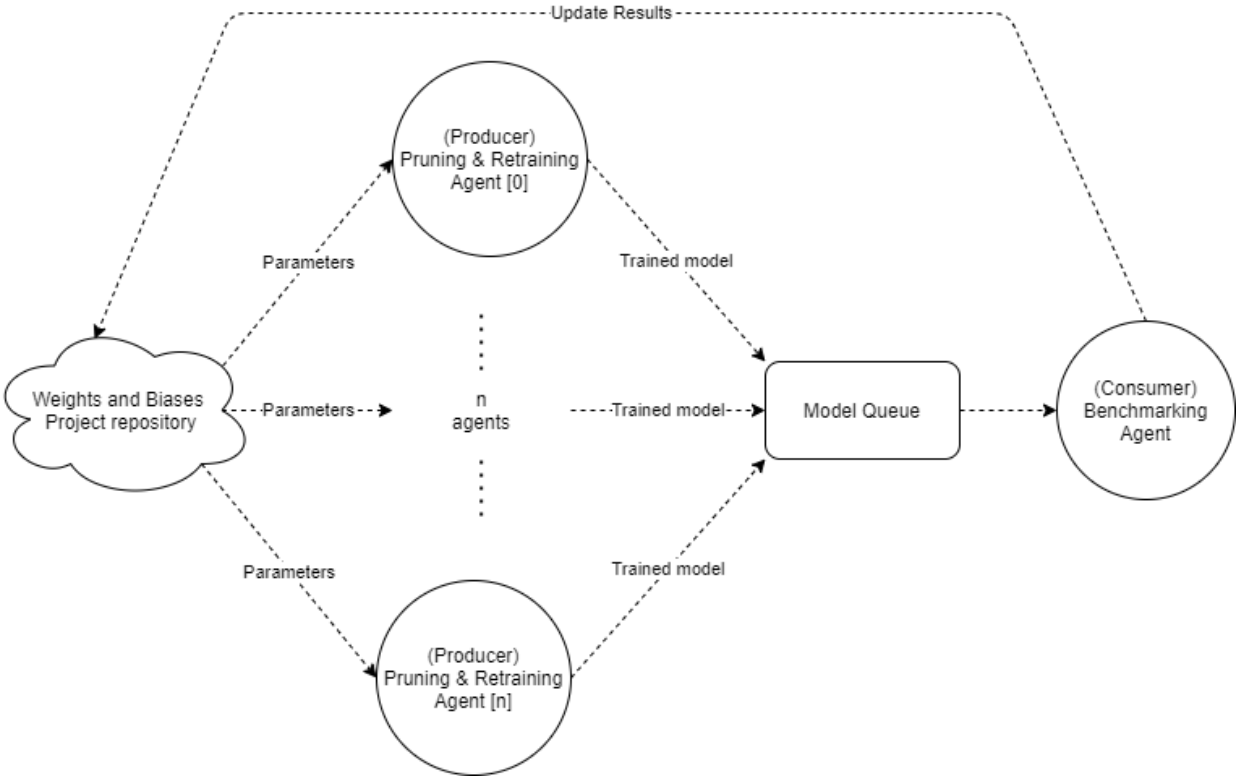

Figure 3.1: Diagram showing communication between discrete parts of the system.

3.1 Automated pruning pipeline

We constructed a pipeline to prune, retrain, benchmark, and record the data from each model, this pipeline consists of 4 separate elements; the systems to prune and retrain, a message queue (for this we used Redis), a benchmarking system, and finally the cloud service used to store the data. Figure **??** shows how each system interacts in the pipeline, pruning is handled by the agent/s marked 'Producer', benchmarking is handled by the 'Consumer' agent, and the Weights and Biases (WandB) system stores and provides the data used to compute each set of sweep parameters passed to the 'Producer' agents on request.

When pruning begins, the producer agent uses initially random pruning parameters, the producer then applies the pruning algorithm, and begins retraining the model. Upon completion of retraining the model is exported into ONNX format and added to a queue for the consumer (the benchmarking agent) to benchmark and record the results, these results are then logged to WandB. The parameter importance and correlation with the target metric is re-computed on each iteration of the pipeline using results that are logged to WandB each time a benchmark is performed.

The runtime of a benchmark for a single model on the NCS is usually at most 5 seconds, retraining the network however can take between 20 - 120 mins depending on the network size and number of epochs, the one-shot pruning method utilised in this experiment usually takes less than 5 seconds. Because the training process is so slow we separated the benchmarking system (consumer) from the pruning and retraining systems (producer), this made it easy to add additional pruning and retraining agents to a single experiment or run multiple experiments in parallel. To make use of this pipeline 2 files must be provided by the user, a distiller schedule with a definition for which weights will be pruned, and a WandB configuration file which defines the type and ranges of the parameters it will seek to optimise.

3.1.1 Defining parameters to prune

```
1          pruners:
2              layer_1_conv_pruner:
3                  class: 'L1RankedStructureParameterPruner'
4                  group_type: Filters
5                  desired_sparsity: 0.9
6                  weights: [
7                      module.layer1.0.conv1.weight,
8                      module.layer1.1.conv1.weight
9                  ]
10         lr_schedulers:
11             exp_finetuning_lr:
12                 class: ExponentialLR
13             gamma: 0.95
14
15         policies:
16             - pruner:
17                 instance_name: layer_1_conv_pruner
18                 epochs: [0]
19
20             - lr_scheduler:
21                     instance_name: exp_finetuning_lr
22                 starting_epoch: 10
23                 ending_epoch: 300
24                 frequency: 1
```

Figure 3.2: Example distiller schedule file, showing the pruning algorithm selected, and that algorithms parameters

Distiller uses a 'compression schedule' file to define the behaviour of the compression algorithms used, Figure **??** shows a simple example compression schedule, with a definition for a single 'pruner' instance (line 2 - ''`layer_1_conv_pruner`'), a single '`lr_scheduler`' instance (line 11 - '`exp_fine_tuning_lr`'), and their respective policies (explained below).

The pruning schedule is composed of lists of sections that describe 'pruners', 'lr-schedulers', and 'policies'. A 'pruner' defines a pruning algorithm and the layers on which that pruning algorithm will be applied, 'LR-schedulers' define the **learning-rate decay(Definition required)** algorithm. Finally each policy references an instance of a pruner or LR-scheduler, and controls when the respective algorithm will be applied, such as the start and end epoch, and the frequency

of application.

The example compression schedule shown in Figure **??** provides instructions to Distiller to use the 'L1RankedStructureParameterPruner' algorithm (as described in Section **??**) to prune the weights in each of the convolutions described by the 'weights' array, in this case '`group_type`' specifies filter pruning and '`desired_sparsity`' indicates how many tensors it will aim to remove (0.9 indicates the algorithm will attempt to remove 90% of the tensors), desired sparsity should not be confused with an actual change in sparsity — note that filter and channel pruning will always result in a dense layer with an actual sparsity of 0 because this is a form of coarse-grained pruning (see section **??**).

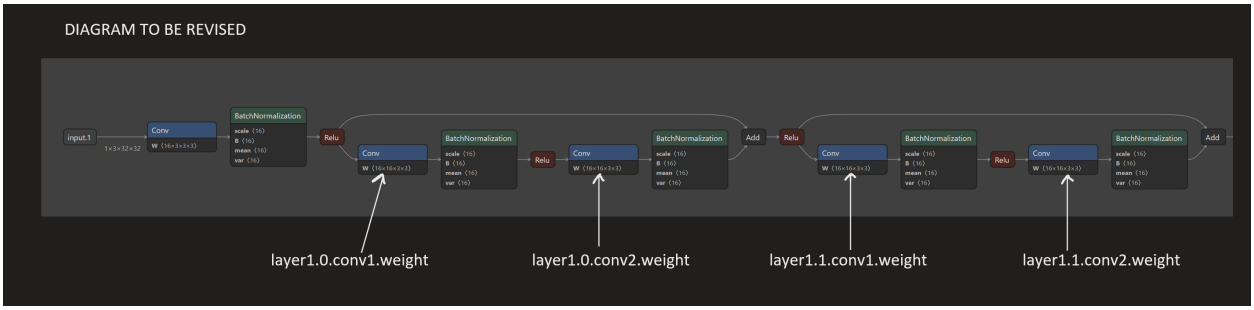

Figure 3.3: Resnet56 fragment showing first 2 residual block with the corresponding weights matrices labelled. **(TODO: rescale and redraw to highlight pertinent information)**

Each grouping of weights in the network has labels (see figure **??**), distiller uses these labels to identify which weight matrices are being referenced by the compression schedule. Lines 7 and 8 in the schedule in Figure **??** reference the weights we wish to prune from the model in Figure **??**.

3.1.2 WandB API

```yaml
program: pipeline.py
method: bayes
metric:
    goal: minimize
    name: Latency
parameters:
    layer_1_conv_pruner_desired_sparsity:
        min: 0.01
        max: 0.99
    layer_1_conv_pruner_group_type:
        values: [Channels, Filters]
```

Figure 3.4: WandB sweep configuration file

To explore the space of pruning parameter values the hyperparameter optimisation framework exposed by called 'Sweeps' was leveraged. This involves writing a python script that can run the entire pipeline (pruning, training & benchmarking) and record the results, to accomplish this each sweep needs a configuration file (see Figure **??**), table **??** shows a description of each key in the configuration file with a summary of appropriate arguments.

Key	Description	Value
program	Script to be run	Path to script
method	Search strategy	grid, random, or bayse
metric	The metric to optimise	Name and direction of metric to optimise
parameters	The parameter bounds to search	Name and min/max or array of fixed values

Table 3.1: Configuration setting keys, descriptions and values

The sweep configuration file tells WandB the names of the parameters to pass as arguments to the pipeline script with their expected value ranges, such as a list of strings or a min and max number. The pipeline script that receives the arguments from WandB contains a mapping from the WandB arguments to a corresponding value in a distiller compression schedule. This is accomplished by parsing a base schedule file and identifying which values will be changed, then a

new schedule is written with the parameters from WandB, this new schedule is then provided to Distiller as the compression schedule to use.

3.1.3 Benchmarking

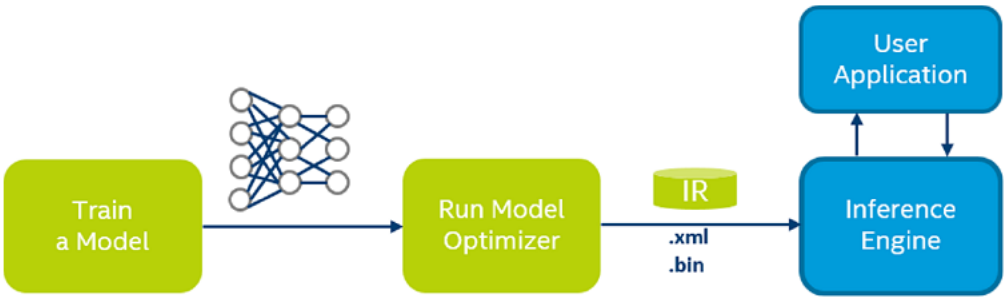

Figure 3.5: Workflow for deploying trained model onto NCS [**ModelOptimizerDeveloper**]

To pass the pruned and trained model to the Neural Compute stick OpenVino was used, it is a toolkit providing a high level **inference engine(Definition needed)** API, this facilitates the process of optimising the model for specialised hardware (in this case the NCS), and loading the optimised model into the hardware. OpenVino itself has a benchmarking tool that we leveraged to access detailed latency and throughput metrics; from end to end latency all the way down to the latency of each instruction used for inference on the VPU **link to example table of HW operations and latency in appendix**. Before starting the benchmark we convert the ONNX model into an Intermediate Representation (IR) format by running it through the model optimiser, the IR is then read by the Inference Engine and loaded into VPU memory. Once the model is ready we load the images that will be used for benchmarking into the VPU memory. We observe three measurements for every model, the end-to-end latency (from loading an image into the model until getting a result), the sum of latency for each instruction executed by the VPU once the image is loaded into memory, and finally we also measure the throughput (the number of images (frames) that can be processed per second or FPS).

3.2 Experiment Design

3.2.1 Filter Pruning algorithm

We selected the one-shot pruning algorithm dubbed 'L1RankedStructureParameterPruner' by Distiller, this is based on the algorithm described by Li et al. in Pruning Filters for Efficient Convnets [**liPruningFiltersEfficient2017**]. We prune the filters that are expected to have the smallest impact on the accuracy of the network, this is determined by computing the sum of the absolute weights in each filter $\Sigma\ |\mathcal{F}_{i,j}|$, sorting them, and pruning the filters starting with the smallest sum values. Each filter that gets removed causes the corresponding feature map to be removed, along with its corresponding kernel in the next convolutional layer, see Figure **??**.

Li et al [**liPruningFiltersEfficient2017**] defines the procedure for pruning m filters from the ith convolutional layer as follows:
Let n_i denote the number of input channels.

1. For each filter $\mathcal{F}_{i,j}$, calculate the sum of its absolute kernel weights $s_j = \Sigma_{l=1}^{n_i}\Sigma\ |\mathcal{K}_l|$.
2. Sort the filters by s_j.
3. Prune m filters with the smallest sum values and their corresponding feature maps. The kernels in the next convolutional layer corresponding to the pruned feature maps are also removed.
4. A new kernel matrix is created for both the ith and $i+1$th layers, and the remaining kernel weights are copied to the new model.

Upon completion of pruning the filters we now retrain the network to regain lost accuracy, **in general pruning the more resilient layers once and retraining can result in much of the lost accuracy to be regained.** Once pruning is completed we compensate for the performance degradation by retraining the network,

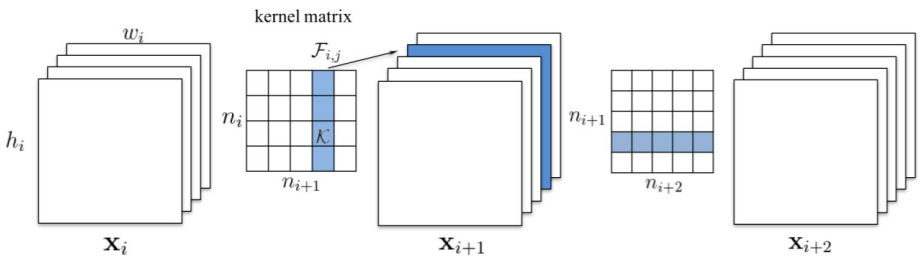

Figure 3.6: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer. [**liPruningFiltersEfficient2017**]**Include annotations for the feature map and kernel**

3.2.2 Model Selection

Pruning CNNs like AlexNet, or VGGNet is fairly straightforward, we can prune filters in any layer without worrying about damaging the fundamental structure of the network, however this is not the case with ResNets (short for Residual Networks), a very popular type of CNN that makes use of what is known as a 'residual block' (Figure **??** shows a residual block) which, from the perspective of pruning, adds additional interdependencies between layers.

We selected ResNet56 as the target network because it is one of the few networks with prebuilt 'off-the-shelf' schedules that also uses residual blocks. Performing this experiment on networks using residual blocks is important because the necessity of using compression techniques such as pruning increases as networks get larger, these residual blocks are very common in very large networks today.

The pre-tuned pruning schedule publicly available from Distiller has been hand built by an expert in the field, providing a solid baseline for comparison. It is not a trivial task to improve on the pre-existing hand built schedules manually without extensive understanding of layer sensitivities.

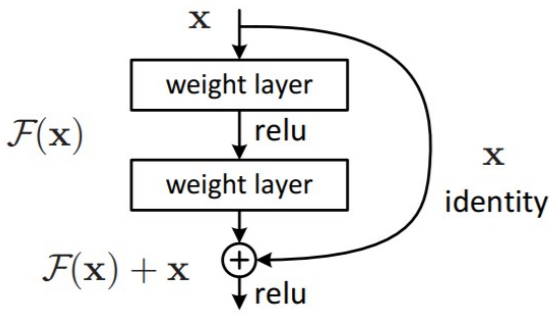

Figure 3.7: A residual block, note the identity feature map skips the weight layers, this is also known as a 'skip connection'.

Discuss further how residual blocks effect pruning. ResNets were originally proposed to help address a training accuracy degradation problem that can occur in very deep networks, degradation (or accuracy saturation) occurs when adding more layers to a suitably deep model leads to higher training error [**heDeepResidualLearning2016**]. The accuracy degradation problem with very deep CNNs is common enough that many new deep networks in research and production make use of them today.

3.2.3 Optimisation Process

In general machine learning training is computationally expensive, execution times can commonly be hours or even days, when the cost of retraining is so high it is easy to see why we would want to exchange some extra work up front for finding a good next search point rather than simply opting for the more commonly used hill climbing algorithms that rely more on the local gradient. One of the key ideas here is to use all information available from all previous evaluations of the search space, allowing us to model the plausibility of future search points, Bayesian optimisation processes are designed to construct this model by evaluating this prior knowledge about known properties, optimisation processes are considered to be some of the most effective optimisation methods in terms of the number of evaluations required [**brochuTutorialBayesianOptimization2010**].

The specific flavour of Bayesian optimisation process used in this dissertation is based on the method described by Snoek et al. [**snoekPracticalBayesianOptimization2012**] where statistical models are used to find search points heuristically using a Monte Carlo estimate of the expected improvement.

Due to the stochastic nature of the pruning methods we have utilised there is a considerable volume of noise in our data because when we perform optimisation we tune all pruners in parallel, each pruner will have a different degree of impact on the target metric. This results in a very noisy search surface (See Figure **??**), this noise can be challenging to overcome and is the primary reason we selected this optimisation approach.

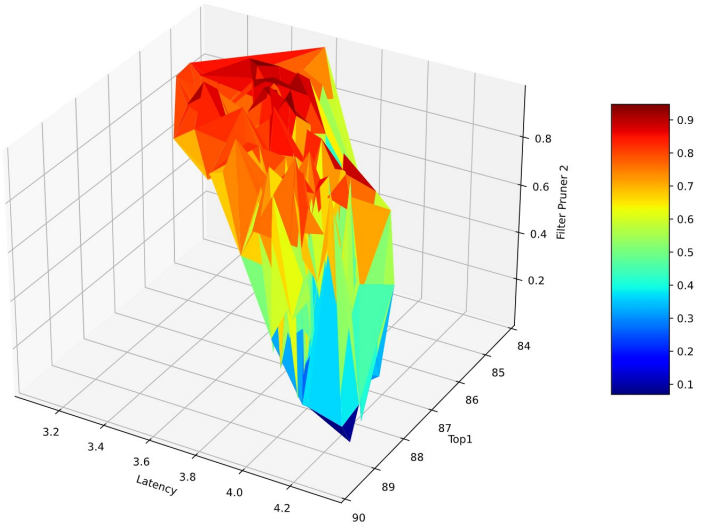

Figure 3.8: A single pruner search surface showing Latency and Top1. The very spikes would be challenging for many hill climbing algorithms, due to the depth of local maxima.

3.2.4 Experiment metrics and parameters

We conducted three experiments using a Resnet56 model with pretrained weights for the CIFAR10 dataset. This section describes the parameters and metrics that will be either set or observed during the experiments.

Parameters:

- **Desired Sparsity** — Specified on a per-pruner basis, this lets us specify to the pruning algorithm what proportion of the weights to try and prune (See Section **??** for more details).
- **Epochs** — The number of epochs used for retraining, one epoch refers to a single cycle of training through the entire training dataset.
- **Learning Rate** — Used to scale how much the retraining process will adjust the weights during each weight update.

Observed metrics:

- **Latency** — Computed by calculating the sum of CPU time for hardware operations inside the NCS after the model and images have been loaded into memory.
- **Total_Latency** — Measures the full latency to perform inference on an image once a model is optimised and loaded into the NCS, including loading the image into the stick memory, this is more indicative of real world requirements.
- **Throughput** — Shows the number of images per second that can be processed by the NCS (Frames Per Second).
- **Top1** — The % accuracy of the first class predicted by the model.
- **Top5** — The % accuracy of the first 5 classes the model predicted for a given image.

3.2.5 Schedule

Table **??** shows how the weights are grouped and labelled for pruning in the selected Resnet56 model, the 4 labelled pruners and their corresponding weights were used in all resnet56 experiments. Layers with a similar degree of sensitivity to pruning are grouped together, layers that are omitted from the table have a much higher sensitivity to pruning and are not pruned at all, pruning more sensitive layers can result in a significantly higher probability that pruned neural networks that lose all predictive ability (in other words the network will predict a single class 100% of the time). Grouping layers in this way helps us avoid having to use 56 pruning parameters (one for each layer per residual block) and significantly reduces the complexity of the parameter search. Note that only the first convolution in each residual block is being pruned (denoted by 'conv1' inside the weight name), because the convolutions following this will also have the kernels removed following the removed feature maps (See Section **??**).

Label	Weights
filter_pruner_layer_1	• module.layer1.0.conv1.weight • module.layer1.1.conv1.weight • module.layer1.2.conv1.weight • module.layer1.3.conv1.weight • module.layer1.4.conv1.weight • module.layer1.5.conv1.weight • module.layer1.6.conv1.weight • module.layer1.7.conv1.weight • module.layer1.8.conv1.weight
filter_pruner_layer_2	• module.layer2.1.conv1.weight • module.layer2.2.conv1.weight • module.layer2.3.conv1.weight • module.layer2.4.conv1.weight • module.layer2.6.conv1.weight • module.layer2.7.conv1.weight
filter_pruner_layer_3.1	• module.layer3.1.conv1.weight
filter_pruner_layer_3.2	• module.layer3.2.conv1.weight • module.layer3.3.conv1.weight • module.layer3.5.conv1.weight • module.layer3.6.conv1.weight • module.layer3.7.conv1.weight • module.layer3.8.conv1.weight

Table 3.2: Mapping of pruners labels to resnet56 weights

3.2.6 Passing pruned/trained networks to benchmark

Discuss reading/writing yaml files, outputting .onnx files, redis to pass messages between agents

3.2.7 Baseline data

For the purposes of all experiments we compare our results to two baseline sets of data, first the basic ResNets56 network with pretrained weights for CIFAR10 no pruning, and second an 'off-the-shelf' version of ResNet56 with pruning parameters hand-picked by the researcher responsible for development of Distiller (thus it is highly likely to be used as a starting point for new users to distiller) [**liPruningFiltersEfficient2017**].

Model	Top1	Top5	Throughput (FPS)	Latency (ms)	Total Latency (ms)
Baseline - no pruning	92.58	99.78	294.08	4.375	13.19
Off the shelf - no retraining	11.19	51.02	303.98	3.947	12.89
Off the shelf - retrained	87.72	99.47	305.27	3.88	12.95

3.2.8 Experiment 1: Rapid pruning, no retraining

Targeting the weights described in table **??** (the full schedule is listed in appendix **??**) we repeatedly pruned networks without performing any training to regain accuracy, we set the target metric to minimize Latency, the number of epochs to 0, and the learning rate to 0.1. The goal at this stage was to observe any reduction in latency, with the added benefit of allowing us to gather a large volume of data very quickly. Figure **??** shows the WandB configuration file used for this part of the experiment, this configuration seeks to optimise the desired sparsity settings for each of the 4 pruners.

```yaml
program: pipeline.py
method: bayes
metric:
  goal: minimize
  name: Latency
parameters:
  filter_pruner_layer_1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_2:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.2:
    min: 0.0
    max: 0.99
```

Figure 3.9: Targeting Latency sweep config

3.2.9 Experiment 2: Target latency, with retraining

Using the same WandB configuration file as in the first experiment (Figure **??**), we again target minimizing latency but this time retrain for a fixed 70 epochs, and set the learning rate to 0.1. The purpose was to observe how well the network would recover accuracy when we focus purely on Latency.

3.2.10 Experiment 3: Target Top1, with retraining

During the third experimental stage we tweaked the configuration file (see Figure **??**) to specify a new target metric: maximise Top1 . Similarly to Experiment 2 we kept the number of epochs fixed at 70 and the learning rate at 0.1. During this experiment we were interested in observing how the optimisation processes effected Top1, and if we could improve on our best Top1 score from the second experiment.

```yaml
program: pipeline.py
method: bayes
metric:
  goal: maximize
  name: Top1
parameters:
  filter_pruner_layer_1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_2:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.1:
    min: 0.0
    max: 0.99
  filter_pruner_layer_3.2:
    min: 0.0
    max: 0.99
```

Figure 3.10: Targeting Latency sweep config

3.2.11 Experiment 4: Improving on the fastest latency settings

Chapter 4

Evaluation

4.1 Evaluation of experimental design

- **clear result presentation**
- **explain problems and difficulties**
- **demonstrate understanding of results**
- **discuss further work**

4.1.1 Pipeline Development challenges

In order to accomplish Objective O0 before building the pipeline we manually ran each discrete part of the system (Distiller, and OpenVino). This immediately highlighted a critical issue; Distiller by default only zeros out the weights when pruning, it doesn't actually remove them from the model so unless a special flag is used with Distiller called 'thinnify' there is no measurable benefit to pruning, this issue is exacerbated by the fact that not all pruning algorithms have been implemented to work with 'thinnify' and there is little to no documentation on which are compatible. A trial and error approach was taken to find pruning algorithms with a working implementation of 'thinnify', we systematically tested each of the pruning algorithms that used coarse grained pruning and eventually identified the algorithm described in Section **??** as fully functional with 'thinnify'.

Once we had a verifiable improvement in latency from the baseline metrics described in Section **??**, we proceeded to develop the pipeline according to Objective O2. Setting up the benchmarking and model queue was very straight forward, for the queue we simply pulled down a docker image of Redis and ran it. The OpenVino benchmarking tool was very easy to integrate into the pipeline, we simply cleared and then wrote the relevant arguments to `sys.argv` (pythons array of command line arguments), called the main method of the benchmarking tool inside OpenVino, and captured the output.

Development of the Pruning & retraining part of the pipeline was considerably more challenging, we encountered a couple of issues that we resolved on our own fork of Distiller; the namespace of the python standard library `parser` module is shadowed by distiller which caused issues with our implementation, we also resolved an issue where Distiller would not correctly construct the path to model checkpoints once they were thinnified.

4.1.2 Hardware and Software

Both the pruning and benchmarking software runs on Ubuntu 20.04, we used various hardware configurations for the pruning and retraining agents. The benchmarking system was fixed for the duration of the project; a KVM virtual machine, 4 cores/8 threads from a Ryzen 3960X, 8Gb RAM, and the Intel Neural Compute Stick was plugged into a USB controller that was passed through to the VM. Originally the benchmarking system was combined with the pruning system, however due to library and python version compatibility issues we had to run OpenVino and Distiller independently this naturally led us to split the system across multiple machines with Redis as a message queue.

Pruning & retraining software — library specification

- Python 3.7.10
- Distiller — A python package for neural network compression research
- wandb — Weights and biases logging library
- PyYaml — Tools to read and write .yaml files
- Watchdog — Monitoring of file system events

- Redis-py — A python interface to a Redis server

Benchmarking software — library specification

- Python 3.8.5
- OpenVino — Deep neural network deployment toolkit
- Pandas — Data analysis tool
- Redis-py — A python interface to a Redis server

4.2 Evaluation of results

4.2.1 Experiment 1: Rapid pruning

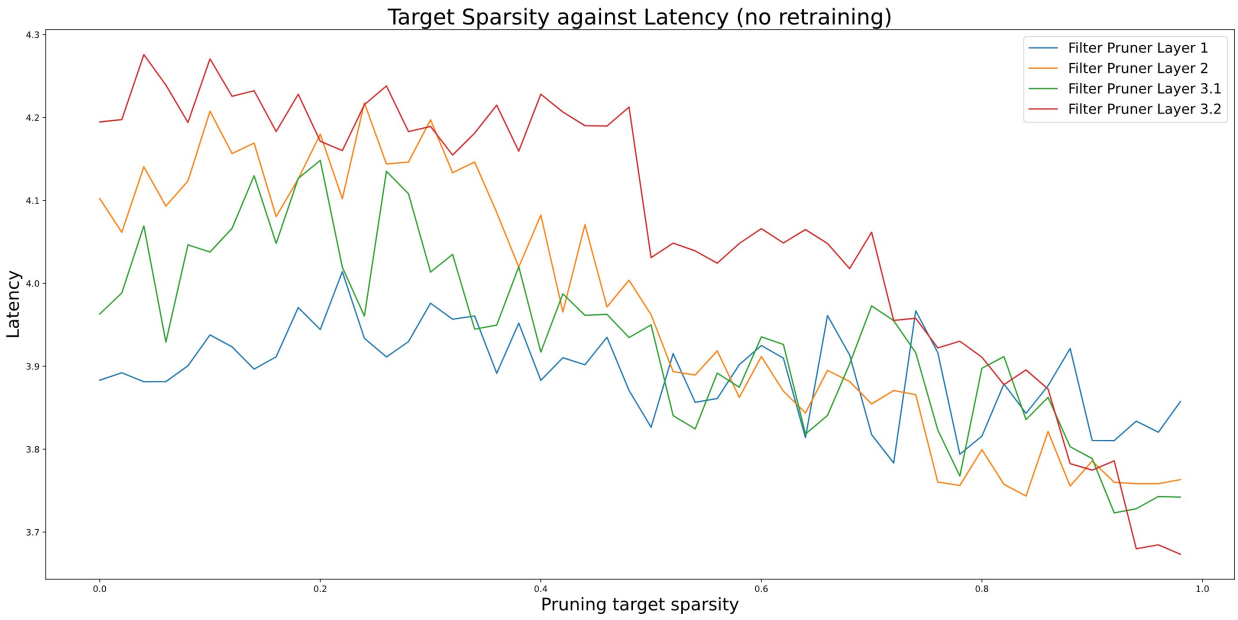

Figure 4.1: Each pruner target sparsity plotted against mean Latency per bin.

As discussed in section **??** for this experiment we set the training epochs to 0 and set the target metric to minimize latency. During this phase of the experiment we gathered data to observe how pruning would affect latency, this was useful as an initial proof of concept. This phase of the experiment was very time efficient, we were able to perform 1631 runs with around 18 hours of compute time; each run usually lasted between 24-55 seconds. Figure **??** shows the mean Latency

computed by using equal width binning, where each bin represents parameter values inside each discretized 0.02 range between 0.0 and 1.0. This chart does obfuscate any relationship between the parameters, however we can see how the filter pruner on Layer 3.2 (red) plots a more dramatic change in latency than the Pruner on Layer 1 (blue), this is also supported by computing the correlation between these values, see Table **??**, where 'Filter Pruner Layer 3.2' has a strong negative correlation with Latency indicating that increasing the desired sparsity results in an increased tendency to observe a lower latency than 'Filter Pruner Layer 1'.

Metric	**Filter Pruner Layer 1**	**Filter Pruner Layer 2**	**Filter Pruner Layer 3.1**	**Filter Pruner Layer 3.2**
Latency	−0.11259	−0.552583	−0.40775	−0.80726
Top1	0.004462	−0.071923	−0.104505	−0.152767

Table 4.1: Correlations between each target sparsity parameter and the metric being measured.

We found that the degree to which we prune was not at all indicative of the resulting accuracy of the network before retraining, for example we observed networks with low desired sparsity across the board that had a much lower Top1 accuracy, than networks that were pruned with much higher targets (See models 523, 1007). It is interesting to note how weakly the pruning targets correlate with Top1 accuracy, this indicates that the relationship between accuracy and pruning is more complex than the naïve perception that pruning less has a smaller accuracy impact. Observation of this weak correlation with Top1 prompted us to begin logging this initial set of metrics in addition to the metrics we logged as described in the Experimental Design Section **??** for experiments following this, we watched this data in the event that some pattern that might be indicative of how well a pruned network can recover accuracy before retraining has begun will emerge.

We observed a case where pruned networks that start with a Top1 accuracy of precisely $100/n$ where n is the number of classes would (according to our data) never recover any accuracy during retraining. Due to the stochastic nature of retraining and the pruning algorithm we selected, coupled with the fact that our methodology necessitated changing the pruning parameters each run we were unable to identify any other pattern or relationship between initial pruning metrics and the metrics of successful or high quality retrained networks from the data we gathered. Future investigation in this area should benchmark after each epoch with the intention of gaining some

insight into how these metrics change over the retraining process, ideally we would aim to project how well a network might recover during the early stages of retraining, allowing us to make an informed decision on whether to abandon the model and try again or not.

4.2.2 Experiment 2: Target Latency

4.2.3 Experiment 3: Target Top1

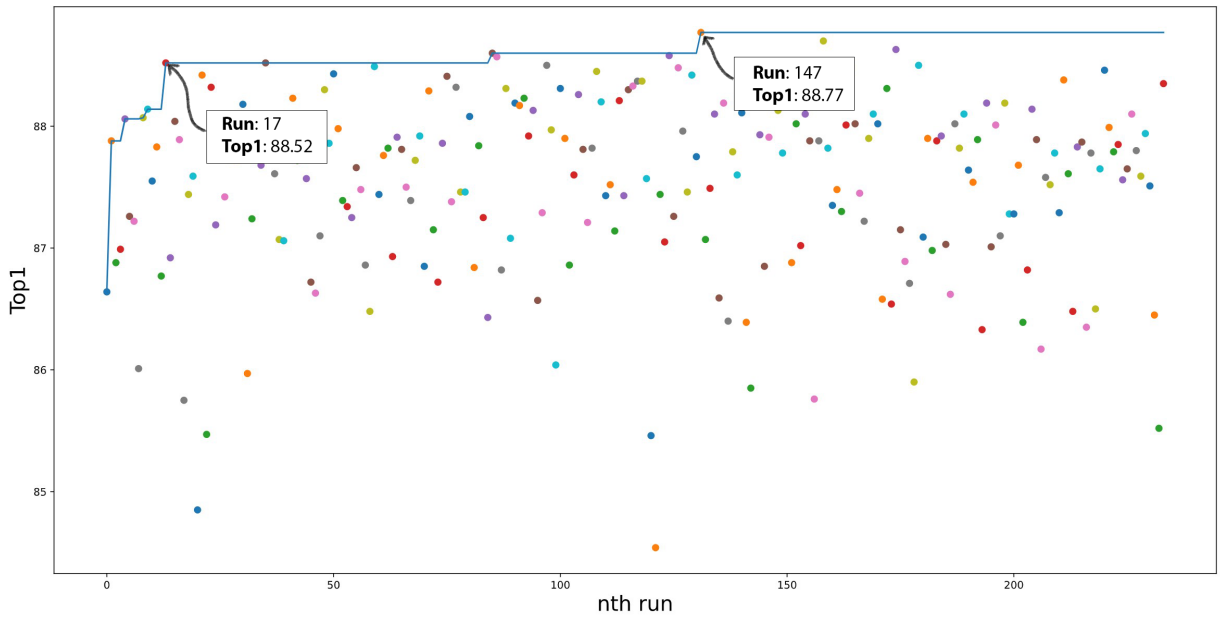

Figure 4.2: Number of runs to achieve best Top1 accuracy found

Interesting observations

- The models that lost all predictive power due to overpruning were not the fastest, even when targeting only latency.
- The relationship between more pruning and lower latency is not as simple as you get a faster model with fewer tensors
- When targeting accuracy we found models with as low latency when targeting latency directly.
- When targeting latency we found models with as high accuracy as when targeting accuracy directly.

- Surprising to see that retraining reduces the latency also

4.3 Further work

- *Suggested improvements for methodology*
- *Next steps*
- More datasets need to be tested
- More models should be used
- Layer selection should be automated
- how well does this system generalise?
- Further investigation should look at a relationship between higher accuracy models after only pruning and how they recover vs low accuracy models after pruning
- Why are there so few completely ruined networks before retraining compared to after retraining?
- Using this data to train a reinforcement learning dnn

Further investigation to identify which untrained pruned models will respond well to retraining (particularly with one-shot pruning methods) would be valuable because retraining is expensive and one-shot pruning is (comparatively) cheap. This could help inform researchers or users of pruning systems weather they should try and prune again or

Chapter 5

Conclusion

5.1 Discussion

- *Discuss results*

What were the actual latency improvements over baseline?

Appendix A

Back matter

A.1 Model Listing

The following section is a listing of cherry-picked models, with a hyperlink to further details for each model, and a description of the Pruning parameters and observed metrics.

All networks in this section have links to the original data as it was gathered, the pruning schedule labels used in this data are different to those described throughout the dissertation. The label names were changed to improve the readability of this dissertation, Table **??** shows how the labels in this dissertation map to the labels in the original data.

Dissertation Label		**Label in Dataset**
Filter Pruner Layer 1	→	filter_pruner_70
Filter Pruner Layer 2	→	filter_pruner_60
Filter Pruner Layer 3.1	→	filter_pruner_20
Filter Pruner Layer 3.2	→	filter_pruner_40

Table A.1: Mapping of labels from dissertation to dataset

Golden-sweep-523

This model was created during the 'no-retraining' experiment, very light pruning parameters with a low Top1, and a latency similar to no pruning. We can see how just a small amount of pruning has a dramatic effect on the accuracy.

Referenced by: [??]

Parameter Name	Value
filter_pruner_20	0.153
filter_pruner_40	0.2536
filter_pruner_60	0.09001
filter_pruner_70	0.09955

(a) Pruning parameters

Metric	Value
Latency(ms)	4.397
Loss	1.924
Throughput(FPS)	302.36
Top1	27.02
Top5	83.26
Total_latency(ms)	13.06

(b) Recorded Metrics

comfy-sweep-1007

This model was created during the 'no-retraining' experiment, high desired pruning parameters with a high (untrained) Top1, and fairly low Latency.

Referenced by: [??]

Parameter Name	Value
filter_pruner_20	0.9403
filter_pruner_40	0.9687
filter_pruner_60	0.3814
filter_pruner_70	0.9707

(c) Pruning parameters

Metric	Value
Latency(ms)	3.347
Loss	1.608
Throughput(FPS)	299.94
Top1	39.32
Top5	89.32
Total_latency(ms)	12.85

(d) Recorded Metrics

A.2 Schedule

The following is a listing of the 'off the shelf' schedule from Distiller. This is a hand written schedule discussed in Section **TBD**.

```
version: 1
pruners:
  filter_pruner_70:
    class: 'L1RankedStructureParameterPruner'
    group_type: Filters
    desired_sparsity: 0.7
    weights: [
      module.layer1.0.conv1.weight,
      module.layer1.1.conv1.weight,
      module.layer1.2.conv1.weight,
      module.layer1.3.conv1.weight,
      module.layer1.4.conv1.weight,
      module.layer1.5.conv1.weight,
      module.layer1.6.conv1.weight,
      module.layer1.7.conv1.weight,
      module.layer1.8.conv1.weight]

  filter_pruner_60:
    class: 'L1RankedStructureParameterPruner'
    group_type: Filters
    desired_sparsity: 0.6
    weights: [
      module.layer2.1.conv1.weight,
      module.layer2.2.conv1.weight,
      module.layer2.3.conv1.weight,
      module.layer2.4.conv1.weight,
      module.layer2.6.conv1.weight,
      module.layer2.7.conv1.weight]

  filter_pruner_20:
    class: 'L1RankedStructureParameterPruner'
    group_type: Filters
    desired_sparsity: 0.2
    weights: [module.layer3.1.conv1.weight]

  filter_pruner_40:
    class: 'L1RankedStructureParameterPruner'
    group_type: Filters
    desired_sparsity: 0.4
    weights: [
      module.layer3.2.conv1.weight,
      module.layer3.3.conv1.weight,
      module.layer3.5.conv1.weight,
```

```
      module.layer3.6.conv1.weight,
      module.layer3.7.conv1.weight,
      module.layer3.8.conv1.weight]


extensions:
  net_thinner:
      class: 'FilterRemover'
      thinning_func_str: remove_filters
      arch: 'resnet56_cifar'
      dataset: 'cifar10'

lr_schedulers:
   exp_finetuning_lr:
     class: ExponentialLR
     gamma: 0.95


policies:
  - pruner:
      instance_name: filter_pruner_70
    epochs: [0]

  - pruner:
      instance_name: filter_pruner_60
    epochs: [0]

  - pruner:
      instance_name: filter_pruner_40
    epochs: [0]

  - pruner:
      instance_name: filter_pruner_20
    epochs: [0]

  - extension:
      instance_name: net_thinner
    epochs: [0]

  - lr_scheduler:
      instance_name: exp_finetuning_lr
    starting_epoch: 10
    ending_epoch: 300
    frequency: 1
```