Inference at the edge: tuning compression parameters for performance

Deliverable 1: Final year Dissertation

BSc Computer Science: Artificial Intelligence

Sam Fay-Hunt — sf52@hw.ac.uk

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

April 9, 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

Acronyms

ASIC application specific integrated circuit.

BLAS basic linear algebra subprograms.

CNN convolutional neural network.

DNN deep neural network.

FC fully connected.

FPGA field programmable gate array.

NLP natural language processing.

RNN recurrent neural network.

SoC system on a chip.

TOPS trillion operations per second.

TPU tensor processing unit.

1 Introduction

1.1 Motivation

With the continued revolution of AI technologies a desire to utilise the power of neural networks at the edge is becoming ever more prevalent. The argument for localising inference is only becoming stronger with the ever-increasing avaliablilty of computation resources alongside new and constantly evolving AI applications, performing inference at the edge can provide better privacy and latency than the remote datacenter alternatives.

talk about how the cost of running cloud based inference services is econmically impractical for companies that might want to leveage this tech in iot devices, unless a subscription model is used. We want to perform inference on the consumers local machine, so companies dont have to charge subscriptions for cloud subscription serivces such as an ai coffee machine. 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 the latency aspect.

Neural network compression is one avenue to aid bringing inference to the edge, intuitively we might think that a network with a smaller memory footprint would naturally have lower inference latency, but this is often not the case. Utilising neural network compression effectively 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 througout a neural network. This alone can make compression a daunting task, even for experienced machine learning practicioners, 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.

1.2 Terminology

Introduce common terminology

- Edge
- Inference
- Epochs

- Channels
- Filters

1.3 Hypothesis

Using a systematic compression method selection process combined with a bayesian optimisation algorithm we can partially automate compression parameter selection and imporve inference latency based on an accuracy threshold in a typical edge computing environment.

1.4 Research Aims

Aim 1 - This dissertation will research methodologies for reducing inference latency using a collection of off-the-shelf compression techniques, we will investigate which compression techniques have a positive effect on inference latency, and consider the context of this improvement with respect to the layer structure of the neural network.

Aim 2 - We will use this contextual information to select approriate compression methodologies and reduce the search space down to a single pruning algorithm.

Aim 3 - Maintain a valid testing environment by using an edge based ai accelerator to perform inference, while training and compression will be performed on a GPU.

Aim 4 - Develop a platform to optimise compression parameters according to a metric representing the union of accuracy and latency.

Objectives

- **O0:** Develop a methodology to verify that the compression methods are actually being applied to the model being represented.
- **O1:** Select at least 1 neural network model to use for testing.
- **O2:** Select 2 suitable datasets for testing with a significant distinction between the cardinality of categories.
- **O3:** Evaluate a pool of compression algorithms with respect to end-to-end latency.
- **O4:** Measure latency for individual layers during inference.

- **O5:** Investigate the effect of composing select algorithms from different compression categories.
- **O6:** Select compression parameters to optimise.
- **O7:** Develop a interface to parameterise select compression methods.
- **O8:** Evaluate a model using a bayesian optimisation approach on compression parameters.

- *Introduce terminology - Inference, neural network model, pruning, layers, channels, filters*
- *Introduce models to be used - high level conceptual representation of the models*
- *Introduce hypothesis*
- *Describe research aims*
- *Define project objectives*
- *Describe how this work contributes to further research*

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 bayse hyperparam optimisation details*

This Section will be split into 4 subsections:

Section 2.1 - **Deep Learning**: An overview of the basic components of a deep neural network and the CNN model.

Section 2.2 - **Neural Network Compression**: Discusses neural network compression techniques and on how they change the underlying representations of DNNs.

Section 2.3 - **AI accelerators** Covers a few popular AI accelerators architectures, their strengths, weaknesses and specialisms.

Section 2.4 - **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

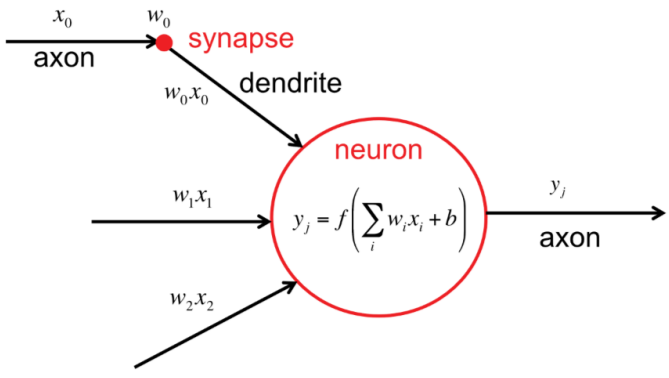

Figure 1: Neuron with corresponding biologically inspired labels.
(Adopted figure from [1])

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 [2].

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 [3]. 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. 1 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 [1].

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 [1], 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 [4]–[6]. The influential paper by Hubel & Weisel [4] ultimately had a significant influence on the design of CNNs via the Neocognitron, as proposed by Fukushima in [7] and again evaluated in [8], 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 paper [7]. 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 [5]. 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 [6].

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. [9] sucessfully applied a bayesian optimisation algorithm to efficiently search for higher quality hyperparameters.

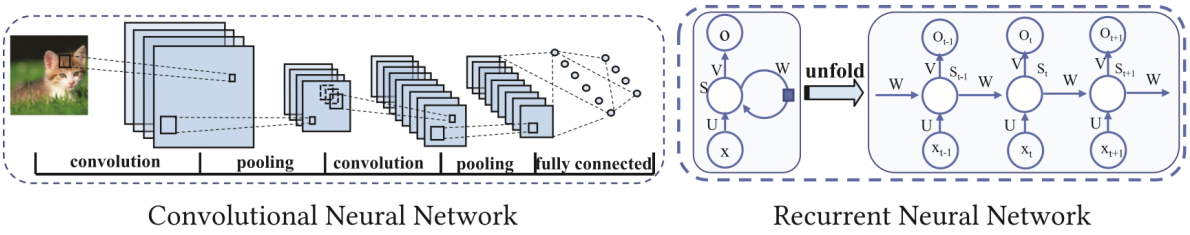

Figure 2: A typical example of a CNN (left) and RNN (right)
(Adopted figure from [10])

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[11], [12], 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 [13]–[16], but more recent research shows that some pruning methodologies can produce pruned networks with no loss of accuracy [17].

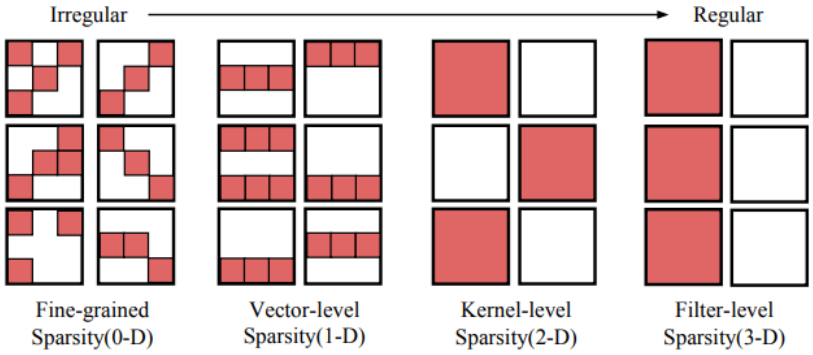

Figure 3: Sparse structures in a 4-dimensional weight tensor. Regular sparsity makes hardware acceleration easier.
(Adopted figure from [18])

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 [18], this is known as the *granularity of sparsity*. Figure 3 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 [15] 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 [19], [20], a software solution has been theorized by Han et al [21] that would involve developing a customized GPU kernel that supports indirect matrix entry lookup and a relative matrix indexing format, see Section 2.2.2 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 [22] 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 [23], [24] 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) [10], Umuroglu et al. [25] 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 2.4.1.

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 4: Comparison of the speed-up when quantising a convolutional layer in Alexnet, 3 different methods.
(Adopted figure from [26])

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. [27] 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. [26] quantised both FC and convolutional layers in Alexnet using their Q-CNN framework

The paper Deep Compression by Han et al [21] 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. 5.

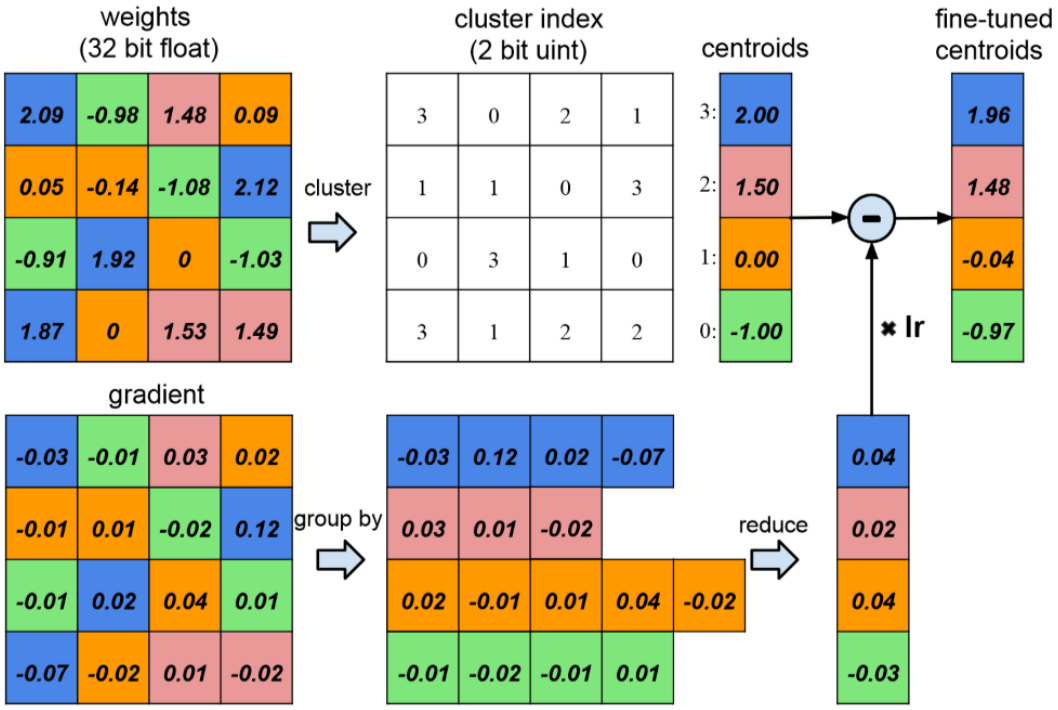

Figure 5: Weight sharing by quantisation with centroid fine-tuning using gradients
(Adopted figure from [21])

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**

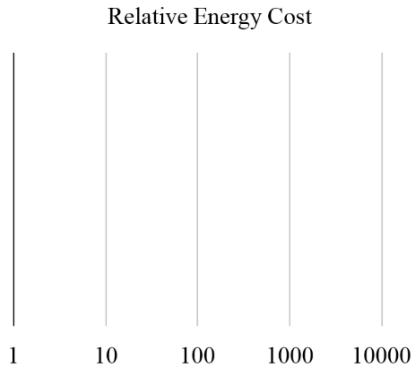

Figure 6: Energy table for 45nm CMOS process
(Adopted figure from [17])

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. 6 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 2.4.1 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)

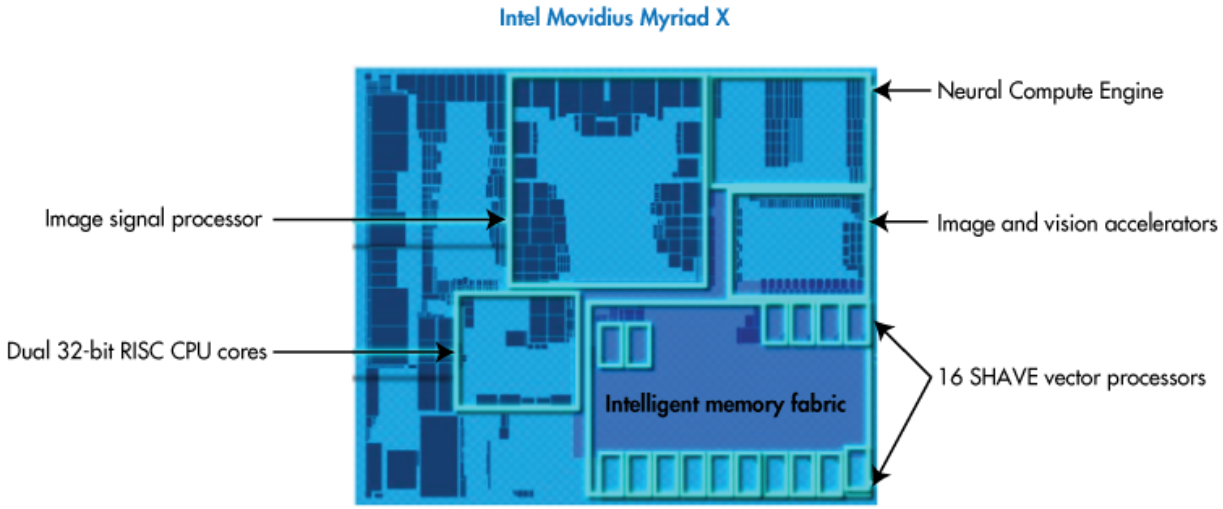

Figure 7: High level view of the Intel Movidius Myriad X VPU

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[28].

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 [29] the latest 4th generation TPUv4 is capable of more than double the matrix multiplication TFLOPs of TPUv3 (Wang et al. [30] 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 [30].

2.4 Memory factors for Deep Neural Networks

2.4.1 Memory Allocation

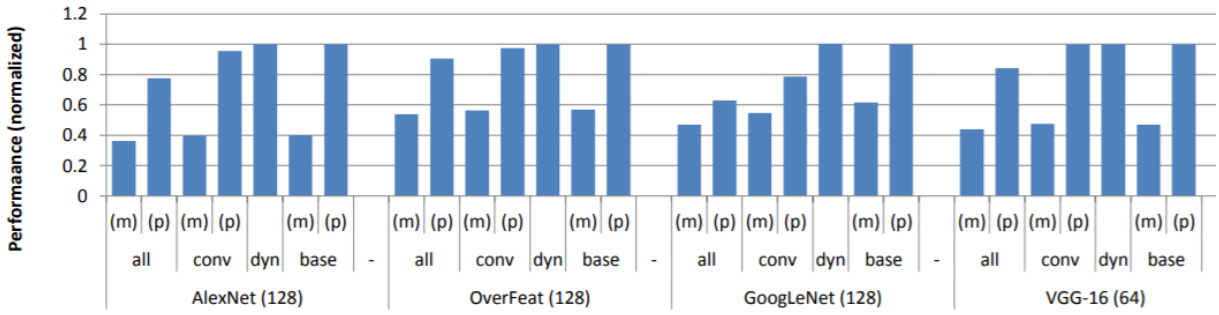

Figure 8: vDNN performance, showing the throughput using various memory allocation strategies. **(Adopted figure from [31])**

While designed specifically for training networks that would otherwise be to large for a GPU, the memory manager vDNN proposed by Rhu et al [31] does provide some insight into the importance of memory locality to neural network throughput. Fig. 8 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. 9. 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. 8 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.

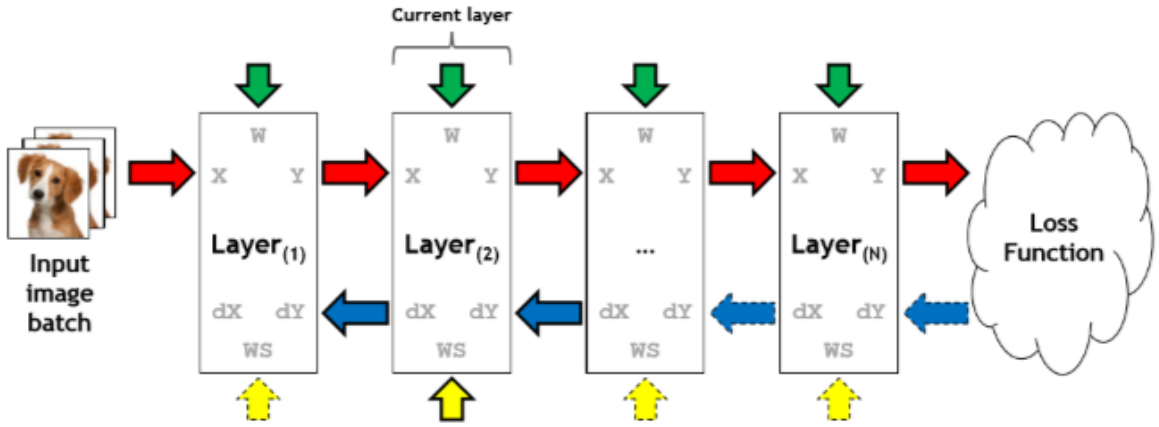

Figure 9: Memory allocations required for linear networks. All green (W) and red (X) arrows are allocated during inference, the blue and yellow arrows are allocated during training.
(Adopted figure from [31])

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. 10), memory bandwidth here can be an issue since loading these weights can be a significant bottleneck [32]. As observed in Section 2.4.1 this indicates that compression (Section 2.2) techniques could help alliviate this bottleneck by making parameters avaliable for cache reuse.

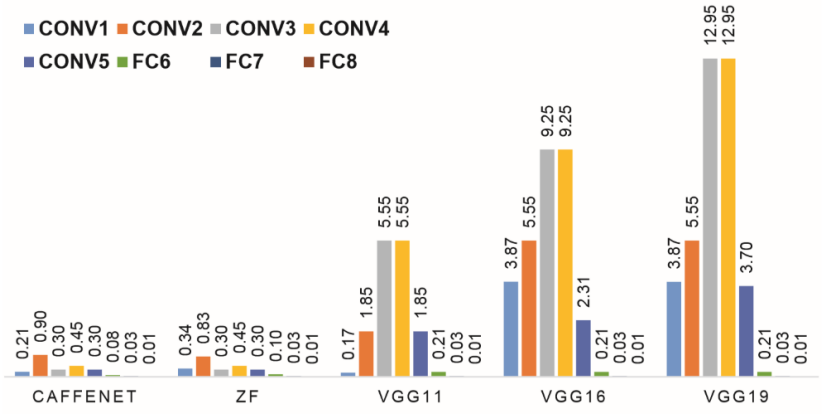

Figure 10: Operations demanded in different layers (GOP)
(Adopted figure from [32])

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 [21], in a follow up paper Han et al. [19] 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. 11, this technique when applied to convolutional layers has some issues.

Han et al [19] 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 [33], 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 2.2.1). It should also be noted that Fig. 11 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 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 [19])

3 Methodology

3.1 Overview

- *Questions to be addressed*
- *Metrics to be measured - why*

This section will discusss the methodology used to search for lower latency models by tweaking pruning parameters.

3.2 Conceptual Process

- *Sensitivity analysis - filter/channel selection and layer interdependencies*
- *Filter pruning implementation - Theory*
- *Channel pruning implementation - Theory*
- *Retraining pruned model*

3.2.1 Sensitivity Analysis

3.2.2 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 [34]. 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 results in the corresponding feature map to be removed, along with its corresponding kernel in the next convolutional layer, see Figure 12.

Li et al [34] 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 degredation by retraining the network,

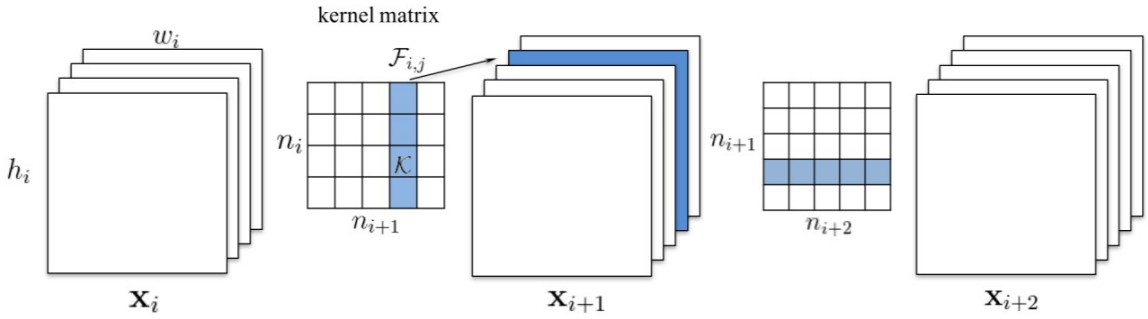

Figure 12: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer. [34]

3.3 Model

Link back to selected model - concrete examples of process described in previous section

- *Filter selection (visual representation of filters)*
- *Channel selection (visual representation of channels)*
- *Discussion of pruning consequences (and recovery) -¿ top1/top5 before retraining and after*

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' which, from the perspective of pruning, adds additional interdependencies between layers.

ResNets were originally proposed to help address a training accuracy degredation problem that can occur in very deep networks, degredation (accuracy saturation) occurs when adding more layers to a suitably deep model leads to higher training error [35]. Today ResNets are commonly used for computer vision tasks, we selected a resnet neural network for this experiment because networks that are shallow enough not to benefit from residual blocks will have far fewer pruning paramaters to manage, the practicality of automating these parameters increases with the size of a network.

The accuracy degredation problem with very deep CNNs is common enough that many new networks in research and production make use of what are known as residual blocks today.

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 experiemnt on networks using residual blocks is important because in general the necessity of using compression techniques such as pruning increases as networks get larger, residual blocks are important for larger networks for the reasons described above. The pre-tuned pruning schedule publicly avaliable from Distiller has been hand built by an expert in the field, so it provides a good point of comparison. It is not a trivial task to improve on the preexisting hand built schedules manually without extensive understanding of layer sensitivities.

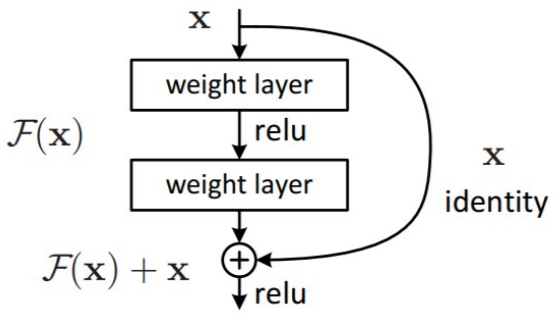

Figure 13: A residual block, note the identity feature map skips the weight layers, these are also known as 'skip connections'.

3.4 Engineering/implementation details

- *High level overview of physical system - justify need for multiple training agents*
- *Pruning & retraining setup - Distiller (Pruning & training)*
- *Benchmarking setup - openvino + benchmark (getting latency/throughput)*
- *Data processing - wandb + data visualisation steps*

3.4.1 High level overview of system

Figure 14 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 wandb system serves the next set of sweep parameters to each of the 'Producer' agents.

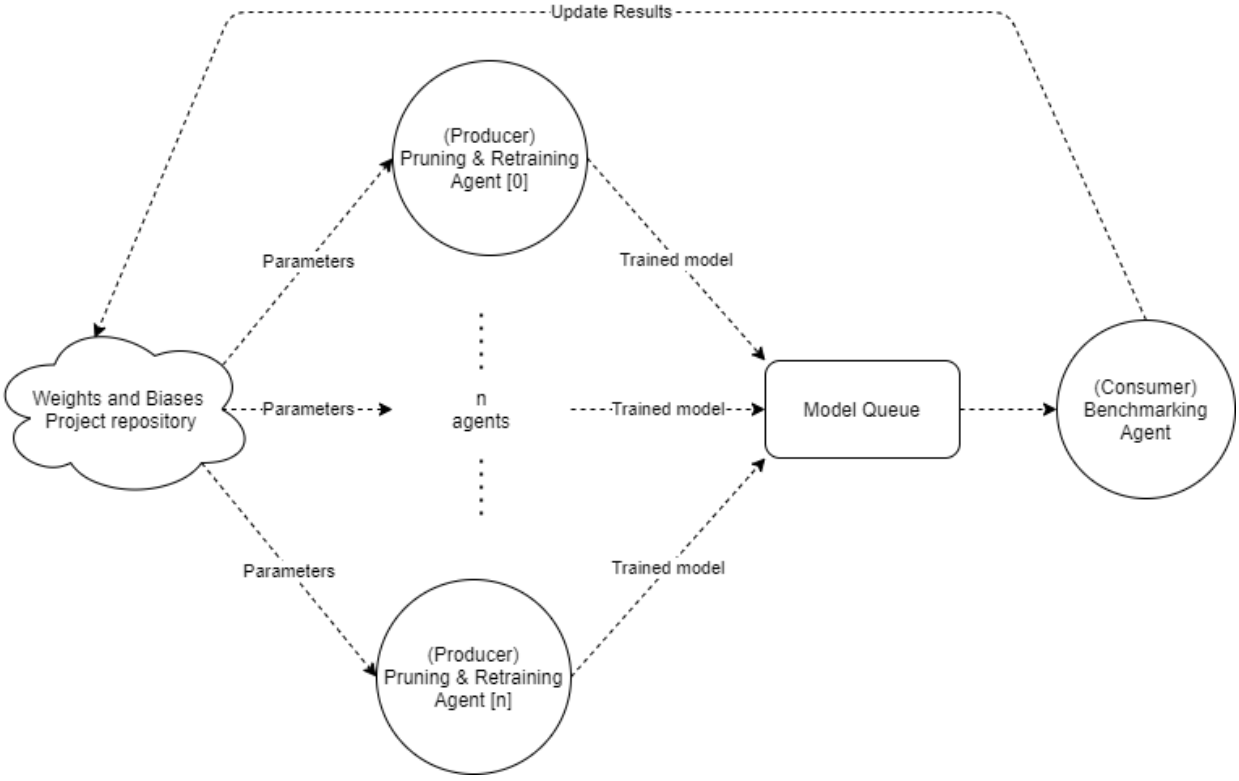

Figure 14: Diagram showing agent communication

When pruning begins, the producer agent requests the (initially random) pruning parameters from the Weights and Biases Project server, 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 weights and biases. As described in **(TBD)** the parameter importance and correlation with the target metric is re-computed each time results are logged this can help determine in what direction to tune the paramater settings to minimise (or maximise) the target metric.

The runtime of a full benchmark for one model on the NCS is usually at most 5 seconds, pruning and retraining the network however can take between 20 - 120 mins depending on the network size

and number of epochs. To imporve the efficiency of the training 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.

3.4.2 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 15: 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 15 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 respecitve 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 15 provides instructions to Distiller to use the ‘L1RankedStructureParameterPruner’ algorithm (as described in Section 3.2.2) 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 2.2.1).

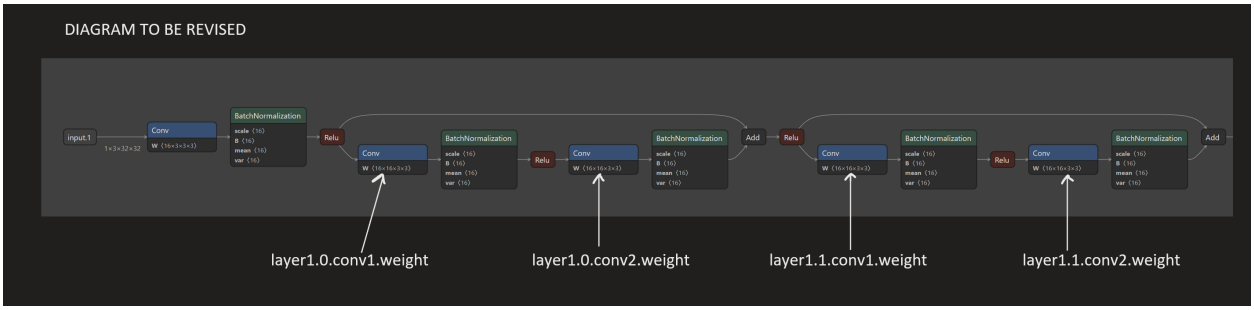

Figure 16: Resnet56 fragment showing first 2 residual block with the corresponding weights matrices labelled. **(TODO: rescale and redraw to highlight pertinent information)**

Each layer in the network is implicitly labelled (see figure 16), distiller uses these labels to identify which weight matricies are being referenced by the compression schedule.

3.4.3 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 17: WandB sweep configuration file

To explore the space of pruning paramater values the hyperparameter optimisation framework exposed by WandB 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 17), table 1 shows a desciption of each key in the wandb 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 1: Configuration setting keys, descriptions and values

This 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 float/integer. The pipeline script that recieves the arguments from wandb contains a mapping from the wandb arguments to the corresponding value in the distiller compression schedule. The pipeline then uses the values provided by wandb and writes out a new schedule that will be fed

into distiller.

3.4.4 Benchmarking

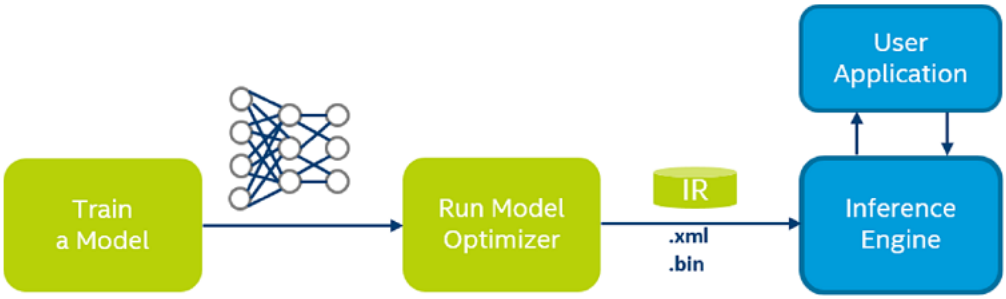

Figure 18: Workflow for deploying trained model onto NCS [36]

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 table of 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 optimizer, the IR can then be read by the Inference Engine and loaded into VPU memory. Once the model is loaded into VPU 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).

3.5 Experiment setup

- Wrapper on Distiller, reading schedule & paramaterise elements
- WandB implementation, defining parameters to optimise

- communication between producer & consumer (redis - pub/sub)
- running benchmark and logging results.

For the purpose of this experiment we chose to use the L1RankedStructureParameterPruner algorithm with filter pruning, .

We conducted three experiments using the Resnet56 model trained on the CIFAR10 dataset. These three experiemnts each used a different target metric: Latency, Top1, and a hybrid metric (see section (TBD)).

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, inlcuding loading the image into the stick memory.
- **Throughput** — Shows the number of images per second that can be processed by the NCS.
- **Top1** — The % accuracy of the most likely class the model predicts.
- **Top5** — The % accuracy of the top 5 predicted classes.

3.5.1 Schedules

Table 2 shows how the weights are grouped and labeled for Filter pruning in the selected Resnet56 model, the 3 labelled pruners and their corresponding weights were used in all 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 rate of pruned neural networks that lose all predictive ability. 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 (**Explanation needed**) is being pruned, because the convolutions following this will also have the kernels removed following the removed feature maps.

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 2: Mapping of pruners to filter weights

3.5.2 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, and second an 'off-the-shelf' version of ResNet56 with parameters hand picked by an expert in the field (**qualify this**) [34].

3.5.3 Latency Target Metric

This experiment targeted pure inference latency, no information regarding accuracy was encoded in the optimisation metric.

4 Evaluation

4.1 Evaluation of experimental design

- *Duration of training*
- *volume of data gathered*
- *(im)practicalities - power consumption?*
- *limitations - single optimisation metric*
- *Criticism of methodology*

The size of the pruned networks is not measured.

4.2 Evaluation of results

- *Summary of results per model/dataset*
- *Deep dive into results, detailed visualisations of accuracy & latency tradeoffs (maybe example with poor quality sensitivity analysis vs higher quality layer selection)*
-

We gathered data in 3 phases; a fast pruning phase targeting latency only with no retraining, targeting latency only with retraining, and targeting accuracy only with retraining.

4.2.1 Fast pruning phase

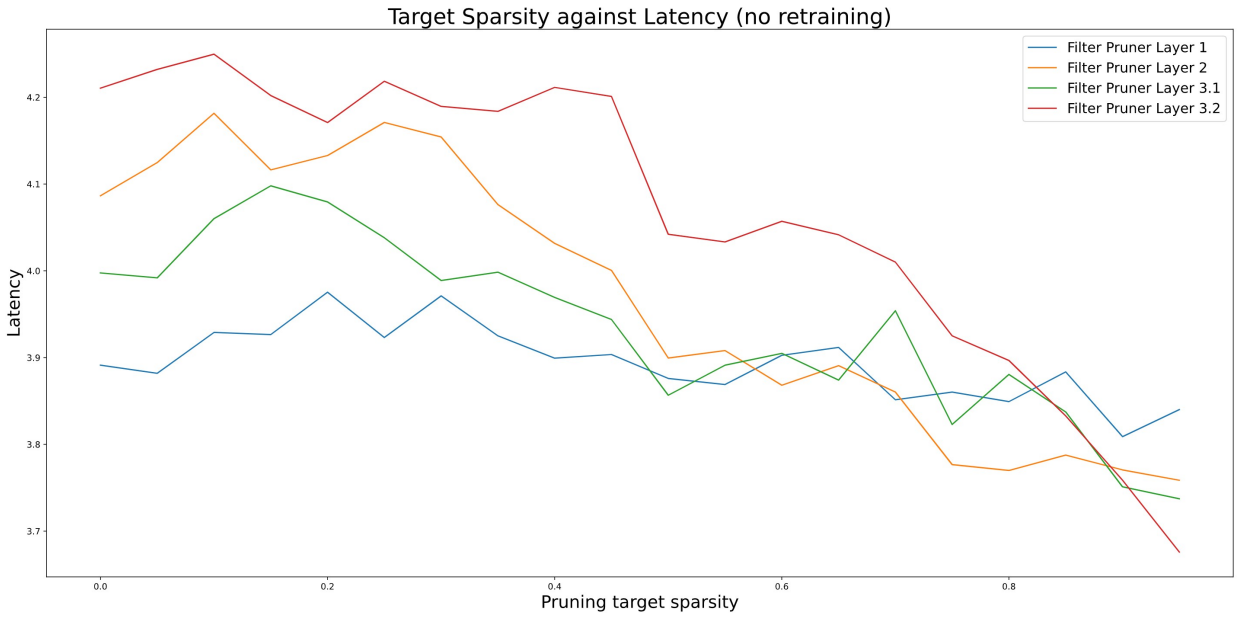

Figure 19: Each pruner target sparisity plotted against mean Latency.

During this phase of the experiment we gathered data to observe how pruning would effect 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. As discussed in section (**TBD**) for this experiment we set the training epochs to 0 and set the target metric to minimize latency. Figure 19 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 pruning the filter pruner on Layer 3.2 (red) has a much greater impact on latency than Pruning on Layer 1 (blue), this is also supported by computing the correlation between these values. Table 3 shows the correlations between each pruning parameter and Latency and Top1

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

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.

5 Conclusion

5.1 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?

5.2 Discussion

- *Discuss results*

What were the actual latency improvements over baseline?

A Back matter

A.1 References

References

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017, ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2017.2761740. [Online]. Available: http://ieeexplore.ieee.org/document/8114708/ (visited on 10/01/2020).

[2] L. Deng, "A tutorial survey of architectures, algorithms, and applications for deep learning," *APSIPA Transactions on Signal and Information Processing*, vol. 3, e2, 2014, ISSN: 2048-7703. DOI: 10.1017/atsip.2013.9. [Online]. Available: https://www.cambridge.org/core/product/identifier/S2048770313000097/type/journal_article (visited on 10/16/2020).

[3] J. Thierry-Mieg, "How the fundamental concepts of mathematics and physics explain deep learning.," p. 16,

[4] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, Jan. 1, 1962, ISSN: 00223751. DOI: 10.1113/jphysiol.1962.sp006837. [Online]. Available: http://doi.wiley.com/10.1113/jphysiol.1962.sp006837 (visited on 10/15/2020).

[5] Y. LeCun, Y. Bengio, and T. B. Laboratories, "Convolutional Networks for Images, Speech, and Time-Series," *The handbook of brain theory and neural networks MIT Press*, p. 15,

[6] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A Survey on Deep Learning: Algorithms, Techniques, and Applications," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–36, Jan. 23, 2019, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3234150. [Online]. Available: https://dl.acm.org/doi/10.1145/3234150 (visited on 10/15/2020).

[7] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980, ISSN: 0340-1200, 1432-0770. DOI: 10.1007/BF00344251. [Online]. Available: http://link.springer.com/10.1007/BF00344251 (visited on 10/18/2020).

[8] ——, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural Networks*, vol. 1, no. 2, pp. 119–130, Jan. 1988, ISSN: 08936080. DOI: 10.1016/0893-6080(88)90014-7. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0893608088900147 (visited on 10/18/2020).

[9] J. Snoek, H. Larochelle, and R. P. Adams. (Aug. 29, 2012). "Practical Bayesian Optimization of Machine Learning Algorithms." arXiv: 1206.2944 [cs, stat], [Online]. Available: http://arxiv.org/abs/1206.2944 (visited on 12/09/2020).

[10] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep Learning on Mobile and Embedded Devices: State-of-the-art, Challenges, and Future Directions," *ACM Computing Surveys*, vol. 53, no. 4, pp. 1–37, Sep. 26, 2020, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3398209. [Online]. Available: https://dl.acm.org/doi/10.1145/3398209 (visited on 10/01/2020).

[11] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. (Apr. 24, 2015). "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition." arXiv: 1412.6553 [cs], [Online]. Available: http://arxiv.org/abs/1412.6553 (visited on 11/23/2020).

[12] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating Very Deep Convolutional Networks for Classification and Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, pp. 1943–1955, Oct. 2016, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2015.2502579.

[13] S. J. Hanson and L. Y. Pratt, "Comparing Biases for Minimal Network Construction with Back-Propagation," p. 9,

[14] B. Hassibi and D. G. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon," p. 8,

[15] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," p. 8,

[16] N. Strom. (1997). "Phoneme probability estimation with dynamic sparsely connected artificial neural networks," undefined, [Online]. Available: /paper/Phoneme-probability-estimation-with-dynamic-neural-Strom/a9392b9299972452ea6fbc3c605f76bb1e21ae42 (visited on 11/13/2020).

[17] S. Han, J. Pool, J. Tran, and W. J. Dally. (Oct. 30, 2015). "Learning both Weights and Connections for Efficient Neural Networks." arXiv: 1506.02626 [cs], [Online]. Available: http://arxiv.org/abs/1506.02626 (visited on 10/30/2020).

[18] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally. (Jun. 4, 2017). "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks." arXiv: 1705.08922 [cs, stat], [Online]. Available: http://arxiv.org/abs/1705.08922 (visited on 11/17/2020).

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, South Korea: IEEE, Jun. 2016, pp. 243–254, ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.30. [Online]. Available: http://ieeexplore.ieee.org/document/7551397/ (visited on 11/02/2020).

[20] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," p. 14, 2017.

[21] S. Han, H. Mao, and W. J. Dally. (Feb. 15, 2016). "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." arXiv: 1510.00149 [cs], [Online]. Available: http://arxiv.org/abs/1510.00149 (visited on 11/06/2020).

[22] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. (Oct. 18, 2016). "Learning Structured Sparsity in Deep Neural Networks." arXiv: 1608.03665 [cs, stat], [Online]. Available: http://arxiv.org/abs/1608.03665 (visited on 11/23/2020).

[23] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2018/html/Jacob_Quantization_and_Training_CVPR_2018_paper.html (visited on 12/02/2020).

[24] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, New York, NY, USA: Association for Computing Machinery, Feb. 22, 2017, pp. 45–54, ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021736. [Online]. Available: https://doi.org/10.1145/3020078.3021736 (visited on 12/02/2020).

[25] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 65–74, 2017. DOI: 10.1145/3020078.3021744. arXiv: 1612.07119. [Online]. Available: http://arxiv.org/abs/1612.07119 (visited on 10/01/2020).

[26] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized Convolutional Neural Networks for Mobile Devices," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 4820–4828. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Wu_Quantized_Convolutional_Neural_CVPR_2016_paper.html (visited on 12/03/2020).

[27] Y. Gong, L. Liu, M. Yang, and L. Bourdev. (Dec. 18, 2014). "Compressing Deep Convolutional Networks using Vector Quantization." arXiv: 1412.6115 [cs], [Online]. Available: http://arxiv.org/abs/1412.6115 (visited on 12/03/2020).

[28] M. Antonini, T. H. Vu, C. Min, A. Montanari, A. Mathur, and F. Kawsar, "Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators," in *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, ser. AIChallengeIoT'19, New York, NY, USA: Association for Computing Machinery, Nov. 10, 2019, pp. 49–55, ISBN: 978-1-4503-7013-4. DOI: 10.1145/3363347.3363363. [Online]. Available: https://doi.org/10.1145/3363347.3363363 (visited on 12/10/2020).

[29] (). "Google wins MLPerf benchmark contest with fastest ML training supercomputer," Google Cloud Blog, [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/google-breaks-ai-performance-records-in-mlperf-with-worlds-fastest-training-supercomputer/ (visited on 11/15/2020).

[30] Y. E. Wang, G.-Y. Wei, and D. Brooks. (Oct. 22, 2019). "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning." arXiv: 1907.10701 [cs, stat], [Online]. Available: http://arxiv.org/abs/1907.10701 (visited on 11/15/2020).

[31] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. (Jul. 28, 2016). "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design." arXiv: 1602.08124 [cs], [Online]. Available: http://arxiv.org/abs/1602.08124 (visited on 10/30/2020).

[32] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, New York, NY, USA: Association for Computing Machinery, Feb. 21, 2016, pp. 26–35, ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847265. [Online]. Available: https://doi.org/10.1145/2847263.2847265 (visited on 11/02/2020).

[33] R. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," p. 455,

[34] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. (Mar. 10, 2017). "Pruning Filters for Efficient ConvNets." arXiv: 1608.08710 [cs], [Online]. Available: http://arxiv.org/abs/1608.08710 (visited on 10/30/2020).

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 770–778, ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90. [Online]. Available: http://ieeexplore.ieee.org/document/7780459/ (visited on 04/07/2021).

[36] (). "Model Optimizer Developer Guide - OpenVINO™ Toolkit," [Online]. Available: https://docs.openvinotoolkit.org/latest/openvino_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html (visited on 03/15/2021).