Heriot-Watt University

Masters Thesis

Enhanced Knowledge Distillation for Neural Network Accelerators

Author:
Andrew Nowlan

Supervisor:
Asst. Prof. Rob Stewart

A thesis submitted in fulfilment of the requirements for the degree of MSc Artificial Intelligence

in the

School of Mathematical and Computer Sciences

July 2020

Declaration of Authorship

I, **Andrew Nowlan**, declare that this thesis titled, "Enhanced Knowledge Distillation for Neural Network Accelerators" and the work presented in it is my own. I 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: *Andrew Nowlan*

Date: 24/07/2020

Abstract

Recent breakthroughs in computer vision can be attributed to advances in deep learning, access to large labelled image datasets and intelligent utilisation of available hardware. Increasing demand for computer vision solutions capable of being deployed on resource constrained devices has driven a huge amount of research in both model compression and hardware design. In recent years, a new class of hardware has emerged to satisfy this demand. In particular, accelerator devices such as the Intel Movidius Myriad X VPU and the Google Coral Edge TPU have been designed specifically to accommodate deep learning workloads with reduced power requirements, providing a state-of-the-art trade off between compute performance and power consumption. Offloading deep learning inference workloads to these accelerators enables low power single board computers to run computer vision applications in real time. Even on more conventional computing platforms, these accelerators are a competitive choice of co-processor compared with typical GPUs due to their superior performance per watt. However, many state-of-the-art classification models have an enormous number of parameters, making them highly computationally and resource intensive. As such, very large models are not supported by these memory constrained accelerators.

It is widely accepted that neural network models exhibit a high level of redundancy, with many parameters contributing little or nothing to the final output. This may be acceptable in scenarios where computation is executed on high performance hardware, but for memory constrained devices, such inefficiency is prohibitive. As a results, a wide range of neural network compression techniques have been developed to help tackle this problem.

In this thesis, we discuss the most popular compression techniques and propose a new method that combines two largely independent approaches. Namely, *knowledge distillation* and *multi-objective evolutionary algorithms*. This method predominantly follows the Neuro-Evolution with Multi-objective Optimisation (NEMO) approach presented in [1], but has been modified to incorporate knowledge distillation and profiles students based on their performance in early stages of training to reduce computational overhead. In summary, our proposed method employs the NSGAII algorithm to evolve knowledge distilled deep neural networks that exhibit an optimal trade-off between inference latency and accuracy on a specific hardware platform, namely the Myriad X VPU.

The ability to deploy highly accurate state-of-the-art models on resource constrained systems has the potential for exciting new developments in robotics, drone technology and smart security cameras. In addition to running state-of-the-art inference on resource constrained devices, exploring the nature of highly compressed models has the potential to provide a better understanding of neural networks in general.

Acknowledgements

I would like to thank Asst. Prof. Rob Stewart for the advice and guidance he has provided throughout supervision of this project. Additionally, i would like to thank Asst. Prof. Michael Lones for sharing his expertise and identifying resources that were valuable to this project. Finally, I wish to express my gratitude to my partner Charlotte and all of my family for the support they have provided throughout my studies.

Contents

Abbreviations

AI	**A**rtificial **I**ntelligence
DL	**D**eep **L**earning
GPP	**G**eneral **P**urpose **P**rocessor
CNN	**C**onvolutional **N**eural **N**etwork
DNN	**D**eep **N**eural **N**etwork
RTCV	**R**eal **T**ime **C**omputer **V**ision
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
VPU	**V**ision **P**rocessing **U**nit
IPU	**I**ntelligence **P**rocessing **U**nit
TPU	**T**ensor **P**rocessing **U**nit
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
FPS	**F**rames **P**er **S**econd
TOPS	**T**era **O**perations **P**er **S**econd
NCS2	**N**eural **C**ompute **S**tick 2
DLDT	**D**eep **L**earning **D**eployment **T**oolkit
MOP	**M**ulti **O**bjective **P**roblem
MOEA	**M**ulti **O**bjective **E**volutionary **A**lgorithm
NSGA	**N**on-Dominated **S**orting **G**enetic **A**lgorithm
NEMO	**N**eural **E**volution with **M**ultiobjective **O**ptimization
NEMOKD	**N**eural **E**volution with **M**ultiobjective **O**ptimization and **K**nowledge **D**istillation

Chapter 1

Introduction

1.1 Motivation

Currently, there are an estimated 50 billion devices connected to the internet and this number continues to grow. Many of these devices are connected at the network edge and produce large volumes of data. Despite the superior computational power provided by cloud platforms, it is often more efficient to process this data at the edge (on the device producing the data). For computer vision applications that require incredibly small response times, sending the data over the network to be processed in the cloud is not viable due to latency constraints. Here, the speed at which data is transferred becomes the bottleneck for applications that offload computation to the cloud. For example, an autonomous car can generate a gigabyte of data every second which must subsequently be processed in real time for the system to determine the appropriate actions to take. The response time incurred by processing all this data in the cloud would be too long and the safety of the systems would be compromised if such an approach was taken. This data must therefore be processed at the edge to ensure acceptable response times. For other video applications, security and privacy can also be major concerns. Processing image data at the edge provides a higher level of security as no data leaves the device. This eliminates the risk of interception when data is passed to the cloud over a network [2].

To facilitate AI applications at the edge, a new class of hardware has emerged in recent years. AI accelerators have been designed to accelerate deep learning workloads while

operating with resource constraints typical of embedded devices. These hardware configurations typically have much lower power requirements than general purpose processors (GPPs) but are also limited in memory. Currently, these devices can deploy a versatile range of computer vision models in real-time, but memory constraints prevent successful deployment of very large state of the art models.

The shift towards edge computing has driven a huge amount of research into neural network compression. It is widely accepted that neural networks contain a high number of redundant parameters which contribute significantly to the memory footprint of the model and negatively impacts inference time. Compression techniques aim to reduce the memory footprint of the model or improve inference time, either by removing redundant parameters or by reducing precision whilst maintaining accuracy as much as possible.

Being able to leverage the accuracy of large state of the art models at a much lower inference times would be hugely beneficial for a wide range of edge applications. Research in this area has already shown promise with hardware specific performance improvements being achieved on a range of different hardware platforms [3]. In this project, we present a method that combines aspects of well known compression and optimisation techniques to simultaneously improve accuracy and inference time for models deployed on a neural network accelerator: the Intel Movidius Myriad X VPU.

1.2 Hypothesis

By combining knowledge distillation and multi-objective evolutionary algorithms to evolve compressed models for deployment on a neural network accelerator, we can produce models that are both faster and more accurate on the target hardware in comparison to the baseline student model from which they evolved.

1.3 Neural Network Compression

This section provides a brief introduction to the four main types of compression techniques. These techniques are revisited in considerably greater depth in Chapter (2).

Improving the performance of neural networks on resource constrained devices is an increasingly popular focus of research. Within the computer vision domain, there are two main objectives that the majority of compression techniques aim to address. Namely, these are:

1. Preserving accuracy while reducing the memory footprint of the computer vision model, enabling the model to fit in the memory of the resource constrained devices.
2. Preserving accuracy while improving inference time on the target hardware.

Often these two objectives coincide, but it is important to note this assertion is not always true. A common misconception is that a linear decrease in memory footprint implies a linear decrease in inference time across all hardware architectures. This is not always the case and it is, therefore, important to distinguish between these two objectives. It has been shown [4, 5] that for specific hardware configurations, removing parameters provides no latency improvements, and in some cases, proves to be detrimental. Other work [3] has demonstrated that hardware specific latency optimisations made on one type of hardware architecture are not guaranteed to be optimal on different hardware architectures.

There are currently four main techniques used for neural network compression. Namely Quantisation, Pruning, Knowledge Distillation and Multi-Objective Optimization. Respectively, these methods deal with precision reduction, redundant parameter (or structure) removal, knowledge transfer from large to small architectures and finally neural network architecture search to optimise two conflicting objectives, typically inference time and accuracy.

In this thesis, we aim to address some of the limitations associated with knowledge distillation in order to optimise the performance of student models on a specific neural network accelerator device. The remainder of this chapter outlines the research aims, hypothesis and objectives formulated as part of this research project.

1.4 Research Aims

1. The main focus of this research project is to optimise knowledge distillation for hardware specific applications by employing multi-objective evolutionary algorithms to evolve student architectures. In doing so, we aim to provide a method that can be used to automate the design of student models for deployment on hardware specific platforms.

2. We wish to explore the extent to which the student's architecture affects accuracy. In particular we aim to establish which of the following approaches produces the most accurate models.

   (a) Selecting a fixed architecture and finding the optimal combinations of KD hyper-parameters through exhaustive search.

   (b) Evolving student models starting with the same baseline architecture used in (2a) with a fixed arbitrary combination of KD hyper-parameters to simultaneously optimise accuracy and latency on the target hardware.

3. Determine whether evolving student models based upon their performance in the early stages of training is a viable means of reducing the amount of computation required to evolve fully trained pareto optimal student models.

1.5 Project Objectives

In order to investigate the validity of our hypothesis and accomplish our research aims, we constructed the following project objectives:

1. Extend the NEMO methodology to incorporate knowledge distillation.

2. Adapt the NEMO methodology to measure objective values on the Myriad X VPU.

3. Identify two computer vision datasets that are suitable for testing our hypothesis using this extended methodology.

4. Design a suitable solution space to facilitate the evolution of student models.

5. Perform at least three experiments with different datasets and teacher-student combinations to determine whether the results generalize across datasets and models.

6. Leverage available hardware to make execution as efficient as possible. In particular, construct a pipeline where evolutionary computation is conducted on a CPU, knowledge distillation is conducted on a GPU and the objective metrics are measured on the Myriad X VPU.

7. Evaluate these experimental results against the baseline model and, where appropriate, compare our results to state-of-the-art results presented in recent knowledge distillation publications.

1.6 Research Contributions

This thesis explores a combination of two largely independent methods. Namely, *knowledge distillation* and *multi-objective evolutionary algorithms*. In combination, these techniques are capable of giving a baseline model an accuracy boost comparable to new state-of-the-art knowledge distillation techniques while reducing latency on the Myriad X VPU. The main contribution of this thesis is an extended NEMO methodology that:

- Replaces the standard training with knowledge distillation thereby automating the design of optimal student architectures.

- Reduces computational overhead by profiling student models based on their performance in early stages of training.

- Measures the objective values (accuracy and latency) on the Myriad X VPU, though this could be adapted to accommodate other hardware platforms.

Chapter 2

Literature Review

In this chapter we survey a range of literature relating to deep learning on hardware specific platforms and neural network compression, providing critical assessment where appropriate. We begin by introducing neural network preliminaries and applications of this technology to the computer vision domain in Sections (2.1), (2.2) and (2.3). We then survey the deep learning hardware landscape in Section (2.4) before discussing some popular compression frameworks and tool-kits in Section (2.5). We then cover the four main compression techniques: Quantisation (Section 2.6), Pruning (Section 2.7), Knowledge Distillation (Section 2.8) and Multi-Objective Optimisation (Section 2.9). We conclude this chapter by exploring combinations of compression techniques (Section 2.10) before providing a critical summary to identify open challenges (Section 2.11).

2.1 Neural Networks for Computer Vision

Convolutional Neural Networks (CNNs) have surpassed traditional image processing techniques and have attained exceptional results in computer vision tasks such as facial recognition and object detection to name only a few [6]. The first proposed model with resemblance to a CNN was the Neocognitron [7], a hierarchical neural network designed for visual pattern recognition. This model was largely based on findings from a study [8] examining the visual cortex of cats. The study revealed that the visual system appears to follow a hierarchical structure in which simple features such as lines and edges are first extracted and subsequently integrated into more complicated patterns, for example

a circle or square. To model this behaviour, Fukushima incorporated feature extraction into neural networks. This approach produced unprecedented results in hand-written digit recognition tasks and went on to inspire CNNs. Incorporating backpropagation, Yann Lacun later proposed the CNN model we are familiar with today. This enabled kernels to be learned, not specified [9].

CNNs have two major differences when compared to conventional Multi-Layer Perceptron architectures. While both architectures employ fully connected layers, CNNs have two additional types of hidden layer:

- **Convolutional layers** - These perform various kernel operations across input images, generating feature maps.
- **Pooling layers** - These reduce the spatial dimensions of the input volume for the next convolutional layer.

2.2 Examples of CNNs for Image Classification

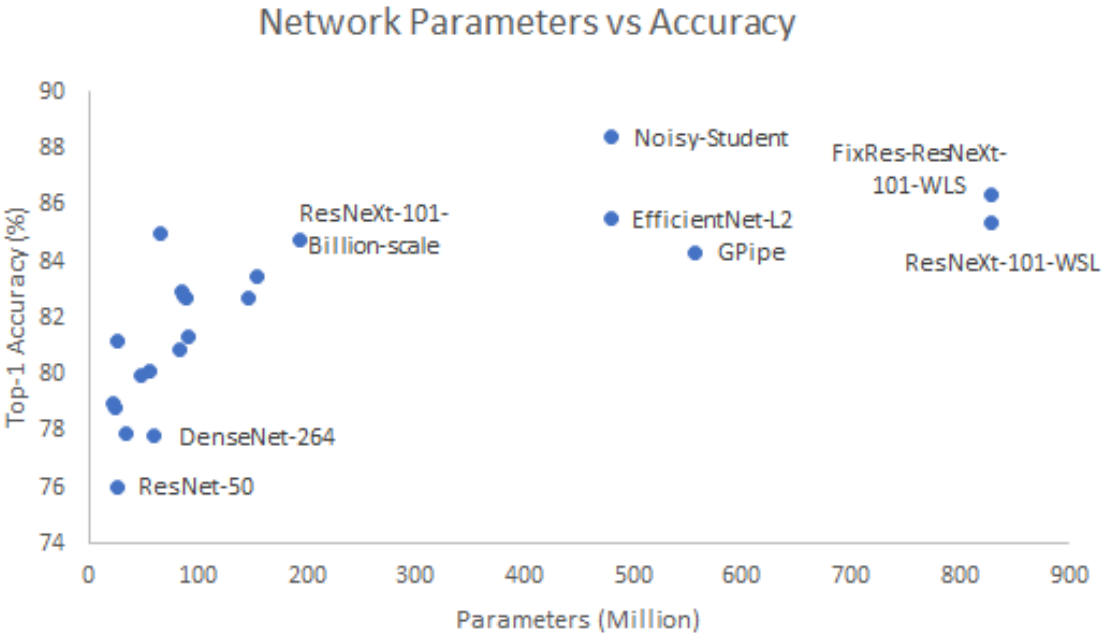

FIGURE 2.1: Top-1 ImageNet Classification accuracy plotted against number of network parameters. Data provided in [10].

A prominent example showcasing the capabilities of CNNs for object recognition is AlexNet [11]. AlexNet is a CNN consisting of 60 million parameters and 650,000 neurons with an architecture comprised of five convolutional layers, a number of max-pooling layers, three fully-connected layers and a final softmax layer. At the time (2012), the network was too large to fit on a single GPU. This challenge was overcome with a method to distribute the network equally across two GPUs, facilitating the training process. This intelligent use of general purpose processors (GPP) demonstrates the impact careful consideration of hardware has had in the Deep Learning (DL) domain. With this methodology, the network was trained to classify 1.2 million images into 1000 classes. Testing on the ImageNet [12] dataset revealed record breaking results with Top-1 and Top-5 accuracy of 62.5% and 83.0% respectively. The performance of AlexNet now continues to be surpassed by new state-of-the art CNNs with the current record holder, NoisyStudent EfficentNet-L2 [10], achieving Top-1 and Top-5 accuracy of 88.4% and 98.7% respectively. While these error rates are a significant improvement on AlexNet in terms of accuracy, the complexity of the model is inherently greater. Illustrated in Figure (2.1), we observe that an increase in accuracy is often accompanied by a significant increase in the number of network parameters. The increasing trend in complexity presents challenges such as large inference and training times, which in turn are driving the demand for innovative and higher performing hardware.

2.3 Real-Time Computer Vision

There are a wide range of real time computer vision (RTCV) applications ranging from robotics to smart security cameras, all of which place greater importance on inference time. Typically, a network must achieve a throughput of approximately 30 FPS to be considered capable of RTCV [13]. The networks discussed in Section (2.2) focus primarily on achieving the best accuracy with little consideration given to the drawbacks of high model complexity. The number of parameters significantly impacts inference time and as a result, many of these state-of-the-art networks are unsuitable for RTCV applications. Another important factor affecting inference time is the hardware on which the individual computations take place, for example, the time to run a single inference through a particular network on a low-end CPU, compared to a high-end GPU, will differ

significantly. The range of hardware options available also vary greatly in their power requirements and this is another important consideration when designing a RTCV system.

Autonomous vehicles are a popular example of a system reliant upon accurate real-time computer vision. A multitude of sensors generate enormous amounts of data by the second, all of which must be processed in real time to enable appropriate actions to be taken. Additionally, this computation must take place on board at the edge, as offloading to the cloud is unfeasible due to time/latency constraints [2]. Computer vision and in particular CNNs, have shown promise in lane detection and redundant object detection with strict inference time and accuracy requirements in place [14]. Experimenting on different hardware platforms, the authors reported network throughputs ranging from 2.5 – 44 FPS, which demonstrates the importance of considering both the model in conjunction with the target hardware. For this particular application, power consumption is not considered a high priority as autonomous cars are easily capable of powering GPUs alongside a range of other sensors and processors. That being said, providing more efficient inference whilst maintaining accuracy could prove beneficial to this application in terms of power efficiency. Other smaller vehicles such as aerial drones have a much smaller supply of power and RTCV applications in this space have to make use of more energy efficient hardware such as AI accelerators, embedded GPUs and so forth.

2.4 Hardware for Neural Network Computation

In this section we discuss the hardware landscape for neural network computation and compare some popular platforms designed specifically for AI workloads.

As illustrated in Figure (2.2), there are a wide range of hardware platforms on which neural network computation can be performed. These range from highly specific non-configurable platforms, exclusively designed to execute a specific workload, through to general purpose processors that are ubiquitous in every modern-day computing device. The CPU has a few cores optimised for fast sequential execution while GPUs have thousands of cores to handle highly parallel workloads. Both ASICs and FPGAs implement digital circuits to perform computation, but since ASICs cannot be reconfigured, their

performance benefits over GPPs are often outweighed by the lack of flexibility. FPGAs are, however, re-configurable through hardware description programming languages but these require digital circuit knowledge and can be extremely difficult to debug.

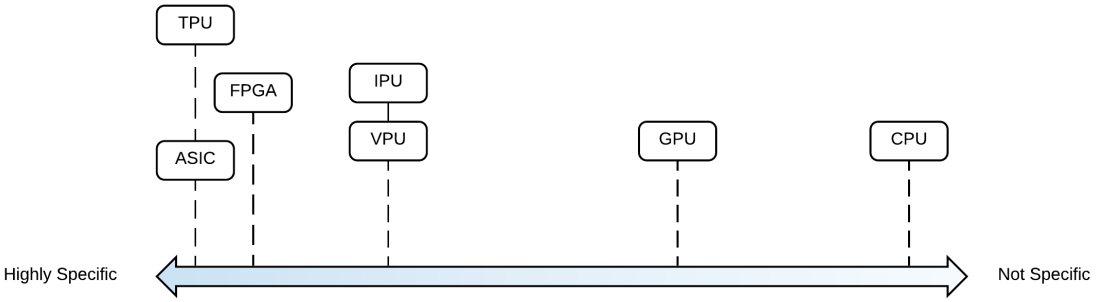

FIGURE 2.2: Domain Specificity of Hardware Platforms for Neural Network Computation.

2.4.1 AI Accelerators

Between general purpose and application specific hardware, there is the new emerging hardware category: AI accelerators. These compact devices have been designed to efficiently perform deep learning workloads with relatively low power requirements. These devices are accompanied by software toolkits that make deployment of DL models very simple by abstracting the complexities of hardware programming. These are of particular interest for embedded system applications due to their ease of use, low power, small physical size, and high performance capabilities. These devices are, however, limited in memory when compared to GPPs. We shall now explore some of these devices in further detail.

2.4.1.1 VPU

The Intel Movidius Myriad X Vision Processing Unit (VPU) is a low power processor designed specifically for accelerating deep learning inference for computer vision applications. An overview of the hardware architecture is provided in Figure (2.3). As shown, this type of processor differs significantly from standard general-purpose processors. The neural compute engine comprises dedicated accelerators for deep neural network computation and with the 16 programmable 128-bit VLIW Vector Processors optimised for

processing highly parallel workloads, the device can compute up to 1TOPs while operating with a power consumption of just 1.5 Watts. The centralised 2.5MB of on chip memory facilitated by the intelligent memory fabric enables memory access latencies of 400 GB/s and reduces the requirements for more costly off-chip data transfer [15].

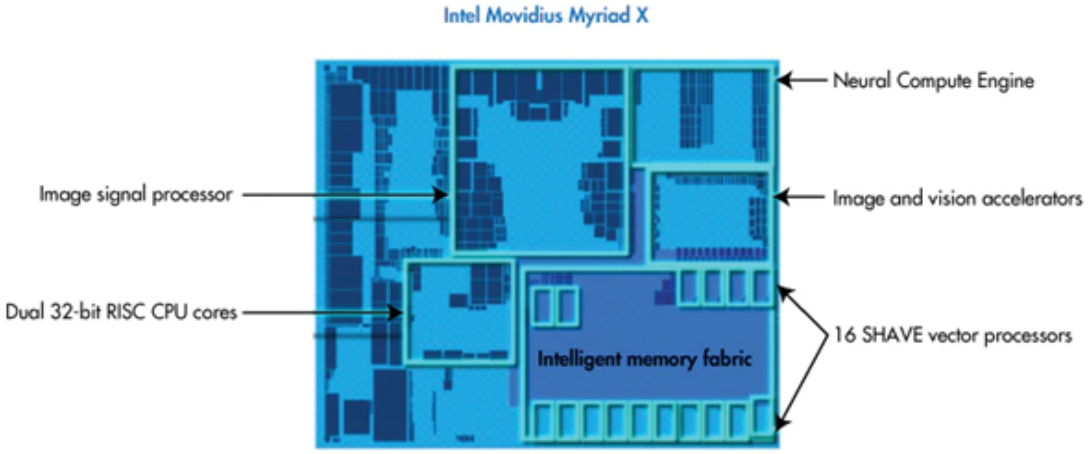

FIGURE 2.3: Intel Movidius X VPU Architecture [16].

Furthermore, Intel have released an easily accessible and deployable USB 3.0 device named the Neural Compute Stick 2 (NCS2) that houses the aforementioned VPU with 4GBits (512MB) of LPDDR4 memory. The NCS2 requires DNN models to be optimised using the OpenVINO framework [17] such that much of the computation can be performed in parallel. When DL inference is performed on this device, memory is gradually allocated at three distinct points. First, when the model is loaded, then at the first inference (warm-up) and finally, during subsequent inferences. Loading and warm-up memory remains allocated for all subsequent inferences and remains so until the model is unloaded. The compilation pipeline is optimised to store as much of the network as possible on the on-board memory to ensure low latency access and to reduce the need to use memory in the host device [18].

2.4.1.2 IPU

As described in [19], Graphcore's Intelligence Processing Unit (IPU) has been designed specifically to process machine intelligence workloads. Design choices made in the creation of this hardware diverge significantly from typical processing platforms. Analogous

to GPUs, IPUs comprise of a large number of cores, 1,216 to be exact, each sophisticated enough to execute individual programs. A radical approach taken to minimise latency involves discarding shared memory entirely. Instead the IPU implements small distributed memories that are allocated to each core. The local memory of each core has a capacity of 256KB with a collective capacity of 304MB per IPU. Utilising static random access memory (SRAM) a much higher bandwidth of 45 TB/s can be attained, resulting in lower latency. This level of performance is comparable to some type of CPU caches and surpasses GPU shared memory. IPUs accomplish their optimal efficiency by assigning multiple threads to individual cores, each of which, offers support for 6 threads. These threads are then rotated according to a static round robin schedule. The programming of this hardware is facilitated through a Graphcore's software toolkit Poplar. This toolkit supports a range of popular DL frameworks.

2.4.1.3 TPU

Similar to the NSC2, the Google Coral is a USB accelerator designed for AI computation at the edge. This device, however, takes a different approach to acceleration and houses Google's Edge TPU (Tensor Processing Unit). The TPU further comprises an ASIC designed to provide high performance neural network computation for TensorFlow Lite models while operating with relatively low power requirements [20].

As discussed in [21], Google have not released specific details on how the Edge TPU works. However, an explanation of Google's Cloud TPU is provided under the assumption that the Edge TPU works in an analogous way.

Google's Cloud TPU is comprised of integrated circuits designed specifically to conduct common neural network operations such as activation and pooling operations. Additionally, the chip comprises a Matrix Multiply Unit and, as the name suggests, is designed for matrix multiplication. This unit uses a principle called systolic execution to help minimise memory bandwidth by storing intermediate computations in processing elements and not in memory [21].

2.4.2 Discussion

Many of the individual computations involved in DL inference and training can be performed in parallel [22]. Kernel operations in convolutional layers are one such example, where each operation is independent of all others. GPUs are particularly well suited for processing these massively parallel workloads, as their large number of cores can perform such operations concurrently. Due to this fact, GPU usage in the DL training phase has exploded in recent years. They are also highly capable of running inference, but high power requirements make them unsuitable for some resource constrained applications. In some cases, CPUs can also provide inference performance similar to that of GPUs, however, they too have large power requirements [23].

Table (2.1) shows a range of devices that offer an excellent trade-off between power consumption and performance, making them suitable candidates for DL edge computing [21]. Figure (2.4) illustrates the trade off between performance and ease of use of these devices. While each of these devices may be best suited for a specific application, we shall discuss the mains strengths and weaknesses.

Programming an FPGA offers a more significant challenge over the other devices, for example the OpenVINO toolkit for the NCS2 abstracts complex hardware programming via high level APIs. This enables models from a wide range of popular DL frameworks to be deployed with little expertise. FPGAs also tend to consume more power than other devices but also have better performance. The Google Coral also provides comparable performance to the FPGA device with lower complexity, however the main drawback of this device is its limitation to 8-bit Tensorflow Lite models. Both the Jetson Nano and the NCS2 are equal in terms of ease of use, with the GPU having a slightly better performance. The difference in power requirements is what sets these devices apart: the Jetson requires 10 times more power than the NCS2. In summary the NCS2 provides the best performance per watt.

The IPU is the most recent AI accelerator to be released and has not been included in the comparison. This device supports a small subset of popular deep learning frameworks through its Graphcore's Poplar Graph Toolchain, currently this hardware platform is not available in an easily accessible USB stick configuration [19]. Since this technology is in its infancy and is not yet widely available, there is not enough information to draw a meaningful comparison with the other devices.

Device	Description	Size	Price	Power Rating
Intel Movidius NCS2	A USB accelerator using a 16-core Visual Processing Unit (VPU)	72.5 mm x 27 mm	$ 87.99	≈ 1W
Google Coral USB	A USB accelerator using a custom Google Application-Specific Integrated Circuit (ASIC)	65 mm x 30 mm	$ 74.99	≈ 2.5W
NVIDIA Jetson Nano	NVIDIA Jetson Nano A single-board computer using a combination of CPU and 128-core CUDA GPU	100 mm x 79 mm	$ 99.00	≈ 10W
PYNQ-Z2	A single-board computer using the combination of CPU and 50k CLB Field-Programmable Gate Array (FPGA)	140 mm x 87 mm	$ 119.00	≈ 13.8W

TABLE 2.1: Comparison of Embedded Devices for AI at the Edge [21].

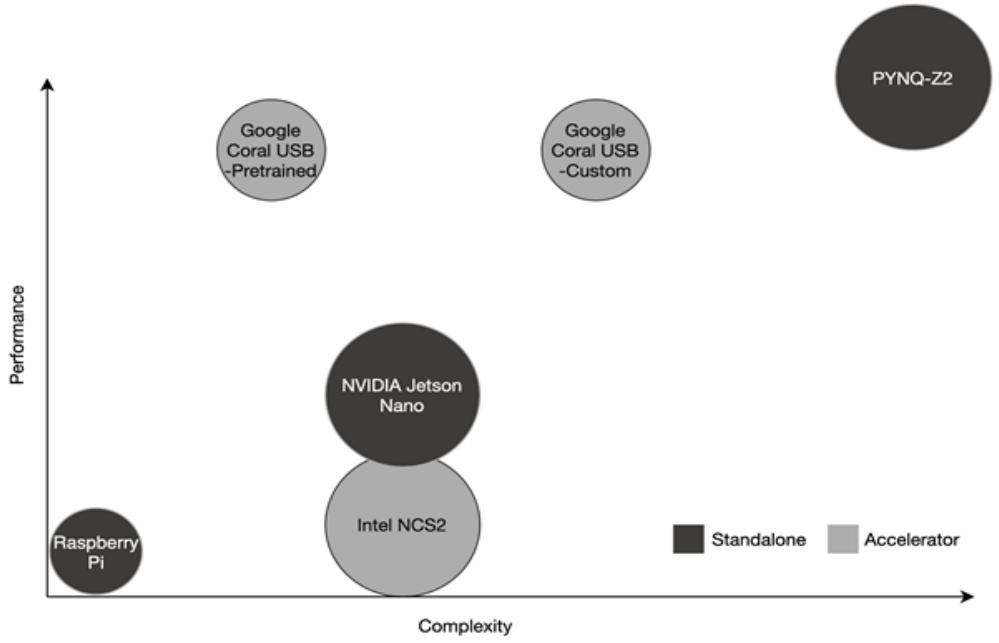

FIGURE 2.4: Performance versus complexity-to-use of embeded devices [21].

2.5 Compression Toolkits and Deep Learning Frameworks

2.5.1 Neural Network Distiller

Neural Network Distiller is an open-source Python package designed to facilitate deep neural network compression research in PyTorch. It houses a library of compression

algorithms that can be applied to a wide range of deep learning models. Additionally, it provides a means to perform experiments under user defined test conditions which aids reproducibility and provides stronger support to experimental results. Another particularly interesting feature of this package is the ability to experiment with combinations of different compression algorithms [24].

2.5.2 RepDistiller

RepDistiller[1] is a publicly available python program for distillation-based compression research and was built to produce the following publication [25]. Built using PyTorch, this compression framework can be used to compare a range of distillation techniques and also investigate the affects of different combinations of hyper-parameters.

2.5.3 OpenVINO

The information provided in this section has been obtained from the OpenVINO website [26] and a case study [27].

The Intel Distribution of OpenVINO is a software toolkit that can be used to develop and deploy computer vision applications on a range of Intel hardware platforms. It includes a library of specific algorithms designed to optimise performance on various Intel target hardware such as CPUs, GPUs, VPUs and FPGAs. The toolkit is comprised of a number of components, however, we shall only discuss the most relevant to this project: The Deep Learning Deployment Toolkit (DLDT).

The DLDT is a high-level API that facilitates efficient execution of deep neural network models on specified target hardware devices. This toolkit is further comprised of two main components:

1. **Model optimizer** – this is a cross platform command line tool that can be used to convert pre-trained models from several popular DL framework formats to an optimised intermediate representation (IR) format for deployment on OpenVINO compatible hardware. As illustrated in Figure (2.5), this IR format consists of two

[1] https://github.com/HobbitLong/RepDistiller

files: an XML file describing the network topology and a binary file containing the network parameters.

2. **Inference Engine** – this is a C++ library providing both C++ and Python APIs for deep learning inference on Intel hardware platforms. The inference engine requires deep learning models to be in the IR format generated by the model optimiser. Optimisations are then applied to these models to improve inference performance. These optimisations include construction and analysis of computational graphs, effective calculation planning and compression.

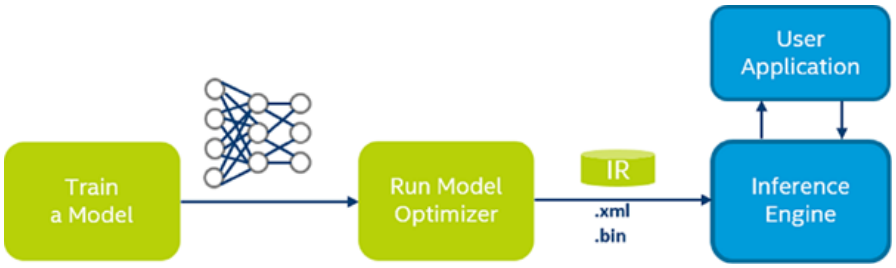

FIGURE 2.5: OpenVINO deep learning deployment workflow [26].

2.5.4 Available Framework to Hardware Options

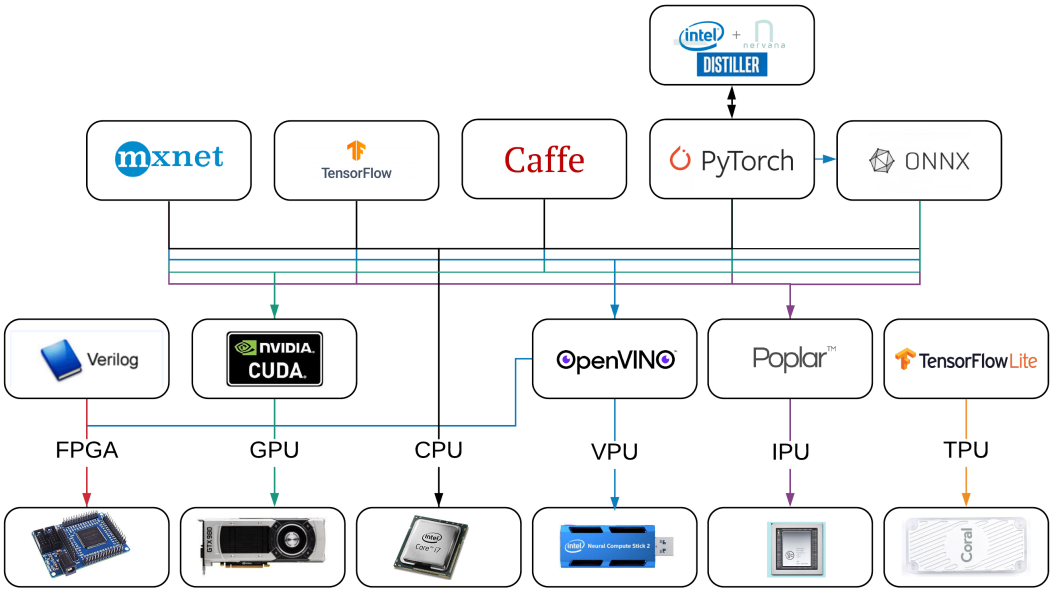

FIGURE 2.6: Examples of Framework-to-hardware Options.

There are many possible options for performing deep learning computation and there are many devices upon which this computation can be performed. Figure (2.6) illustrates

some popular software to hardware paths for deep learning computation on various hardware platforms.

2.6 Quantisation

Quantisation is a compression technique that reduces the precision of weights and activations in deep neural network models. Lower precision models can have significantly faster compute times when deployed on appropriate hardware platforms. This field has recently gained a lot of traction and there are now a wide range of different approaches and algorithms that can be used to perform hardware or application specific quantisation [28].

2.6.1 Fixed Point Representation

Floating-point-quantized models permit individual parameters a range of exponent values. Larger exponent (higher precision) values can induce more computational overhead, leading to higher power consumption and longer compute times. Fixed-point-quantised models use (usually smaller) fixed exponent values for all network parameters. That is, all parameters have the same precision. This imposed restriction brings a range of benefits such as faster and more power efficient mathematical operations but can also potentially impact the model accuracy [28].

2.6.2 Binary Quantisation

As the name suggests, binary quantisation restricts the permissible values of the model parameters to just two values. Normally 1 and -1. This method greatly simplifies the arithmetic operations during inference, but is often accompanied by a drop in accuracy. It is important to note that the back propagation algorithm applied during training is sensitive to the precision of weights, as such, weights are only binarized during feed forward passes [28].

2.6.3 Logarithmic Quantisation

Logarithmic quantisation converts weight parameters to powers of 2 with a scaling factor. This representation can allow a huge range of values to be specified using only a few bits. This technique has been shown to enable significant layer compression without incurring accuracy loss [28].

2.6.4 Discussion

Quantisation has proven to be a successful method of reducing the memory footprint and improving the performance of deep learning models on resource constrained hardware [28, 29]. While different quantisation algorithms make this compression technique very versatile, the degree to which quantisation can be applied ultimately depends on the capabilities of the target hardware. If a certain hardware architecture only supports a limited type of quantisation, then other compression techniques often prove more useful.

2.7 Pruning

Before explaining what is meant by the term pruning, it is important to understand the notion of sparsity. Tensors have become the data structure of choice for storing network parameters and facilitating efficient mathematical operations for deep learning computation. For simplicity, tensors can be considered multidimensional arrays comprised of individual elements. Sparsity is a measure of the number of zero elements in a tensor, relative to its size. A tensor is considered "*sparse*" if the majority of tensor elements are zeros. Conversely, density is the measure of the non-zero elements in a tensor relative to size and is the complement of sparsity [30].

A Convolutional Neural Network (CNN) architecture is specified prior to the training process, leaving no scope for improvements to be made to the network structure. Pruning has been devised to tackle this limitation [31]. Pruning is a form of DNN compression concerned with eliminating redundant parameters [28]. It works by removing these redundant parameters according to some pruning criteria, in some cases introducing a higher level of sparsity into the network model and in other cases, creating smaller dense architectures. Pruning can be performed at two different levels of granularity:

- **Fine-grained** pruning refers to the removal of individual weight elements.
- **Coarse-grained** pruning refers to the removal of entire groups of elements, for example channels/filters.

2.7.1 Fine-Grained Pruning Methods

Son Han et al [31] successfully showed that pruning can significantly reduce the number of parameters in DL models without impacting accuracy. This was achieved by following the three-step procedure described below:

1. An initial training phase is conducted to identify the most important connections.
2. Weights are pruned according to a specified criterion (post-training).
3. The network is then re-trained to compensate for the lost connections.

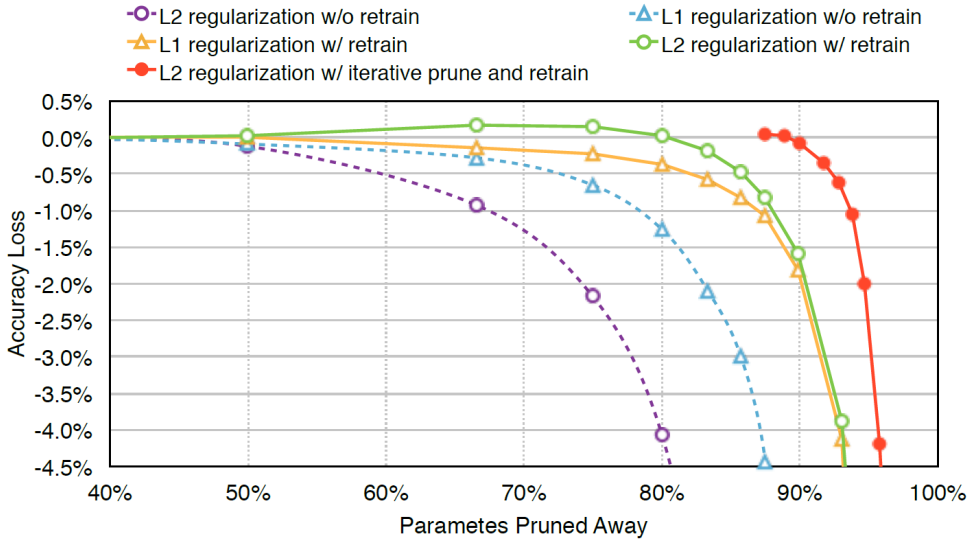

FIGURE 2.7: Experimental results showing the relationship between accuracy and the number of parameters that have been pruned, subject to other confounding factors [31].

Figure (2.7) shows that as the number of pruned parameters increases, model accuracy decreases. The best results, remarkably, do not see an accuracy drop until nearly 90% of parameters have been pruned. This work interestingly reveals that the type of regularisation used can have a significant impact on the pruning outcome. L2 regularisation is shown to produce better results when used in conjunction with retraining, while L1 regularisation shows better performance only when this step is omitted. This figure also

highlights the importance of retraining and, even more so, the importance of an iterative pruning process. It is clearly shown that accuracy drops off relatively early when retraining is omitted and that repeated application of the three-step process produces by far the best results. Other important findings revealed that different layer types have varying sensitivities to pruning. In particular, convolutional layers have a higher sensitivity when compared to fully connected layers. Results obtained from measuring these sensitivities allowed appropriate pruning thresholds to be devised for individual layers.

Applying this pruning technique to the AlexNet model, the authors reduced the number of parameters by a factor of 9. That is, a reduction from 68M to 6.7M parameters without incurring any accuracy loss. Similarly, VGG-15 was reduced by a factor of 13 without loss of accuracy.

As described by the authors, the motivating factor for this work was predominantly the deployment of pruned state-of-the-art models on mobile devices. While these remarkable results are extremely promising, deploying these pruned networks on mobile devices is impractical due to the limitations of general-purpose and some accelerator hardware for sparse computation. Additional work demonstrating and comparing model inference time on general purpose and application specific platforms would have been an informative inclusion in these works. Another drawback of this pruning method is the requirement of high-end GPUs and data for re-training. Despite utilizing such hardware, relatively long retraining times should be expected with the pruned AlexNet taking over a week to retrain.

Guo et al [32] proposed a different dynamic approach to fine-grained pruning. As an alternative to the greedy method where parameters are permanently removed, continual connections splicing was incorporated to provide a means to prevent poor pruning choices. The splicing operation in this method allows connections to be recovered if they are found to be important at a later stage. This approach arose from the idea that parameter importance may radically change once others have been pruned away. Figure (2.8) illustrates this dynamic process, highlighting the splicing stage where connections are restored (shown in green). As with the previous pruning methods, hardware optimised for sparse computations would be required to deploy this type of sparse tensor

representation with meaningful benefits. Additionally, this process happens dynamically during training and would be subject to large training times for complex models.

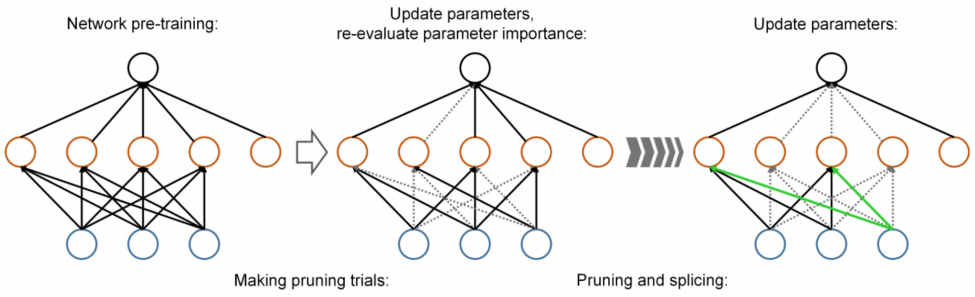

FIGURE 2.8: High level representation of the dynamic pruning process [32].

Other related work by Zhu et al [33] have proposed yet another approach that employs a gradual pruning process that can be easily incorporated in standard DNN model training. Weights in a particular layer are sorted by their magnitude. The smallest are then pruned until a specified sparsity level is achieved, that is, the values with lowest magnitude have their values in the corresponding mask matrix set to zero. Weights that are masked in the feed forward pass are ignored in the back-propagation process. The ingenuity of this approach arises from an algorithm that automatically increases the sparsity level over a period of pruning steps. This gradual process allows the network to recover accuracy loss caused by pruning. Experiments carried out with this pruning method showed negligible accuracy loss with a 50% reduction in parameters. Furthermore, pruning 87% of parameters resulted in an accuracy drop of less than 4%. The authors additionally went on to show that large sparse models outperformed smaller dense models by a significant margin in terms of accuracy. It was, however, acknowledged that specific hardware optimised for sparse computation would also be needed to bring improved inference and power consumption to edge devices.

2.7.2 Coarse-Grained Pruning Methods

Abbasi et al [34] proposed a greedy pruning algorithm that removes filters in trained CNNs. In order to decide which filter should be removed, the algorithm incorporates a filter importance index. This index is determined by the classification accuracy reduction (CAR) of the network, after pruning that filter. This method was shown to produce significantly better compression when compared to other previously defined filter pruning

schemes. Notably, the authors combined further weight pruning with their algorithm such that the first or second convolutional layer of AlexNet can be reduced by a factor of 42 with negligible accuracy loss. Acknowledged by the authors, a greedy algorithm may not provide the best overall solution for determining the best filters to remove. As it would be far too computationally expensive to search every subset of filters to find candidates for pruning, the author suggests investigating genetic algorithms and particles swarm optimisation as a method to select filters for pruning as possible future works.

Fisher Pruning [35] is another greedy pruning algorithm that can be used to remove channels with minimal contribution to model output. While this method can be used to prune individual parameters, it is often more beneficial to prune whole channels as sparse kernel operations provide little benefit in terms of performance. This pruning method assumes model parameters are trained to minimise cross-entropy loss and approximates the change in error incurred upon removal of a given filter. Using notation from the Fisher Pruning summary provided in [5], the change in error approximation for an arbitrary channel, C, is given by

$$\Delta_c = \frac{1}{2N} \sum_{n}^{N} \left(- \sum_{i}^{W} \sum_{j}^{H} C_{nij} g_{nij} \right)^2 \tag{2.1}$$

where W and H are the channel's spatial width and height. N is the number of examples in an arbitrary mini-batch input. C_{nij} denotes the entry corresponding to the n^{th} example in N at location (i, j) with g_{nij} denoting the corresponding loss function gradient.

This Δ_c value is accumulatively calculated for each channel during fine-tuning and the channel with the lowest value is pruned. This prune-retrain cycle is repeated until the network has been compressed to some specified size.

2.7.3 Performance Aware Pruning

The fine-grained pruning algorithms discussed in previous sections simply replace redundant parameters with zeros, introducing a higher degree of sparsity into the tensor

representation of the network. This approach is limited in its applicability as the reduction in parameters can only be leveraged for performance gains by using hardware specifically designed for sparse computation. As such, GPPs do not benefit from better inference performance when executing these large sparse networks. This is also true of neural network accelerators not specifically designed for this type of computation. Prior to coarse grained pruning research with a focus on inference time, parameter reduction whilst maintaining accuracy was the sole focus and little consideration was given to the target hardware executing the inference computations. Additionally, it was also widely assumed that a reduction in parameters/groups of parameters always lead to improved inference regardless of target device. As presented in Radu et al's [4] recent work, this assumption is not always correct. In this work, the authors focus on computational performance gains that can be achieved through coarse-grained pruning. Similar to filter pruning discussed in Section (2.7.2), channel pruning reduces the networks size by removing the least significant channels according to some metric, resulting in a smaller network size that does not compromise the accuracy of the original model. By considering the target hardware in conjunction with their corresponding programming libraries, the authors demonstrated that inference times on embedded GPUs can benefit from channel pruning, though in some cases pruning was also shown to be prohibitive. More specifically, it was shown a 12% reduction in size can, in some cases, be detrimental and increase inference time by a factor of two. In contrast, performance-aware pruning was shown to improve inference time by a factor of three. In the remainder of this section we highlight some of the experimental results presented in this study.

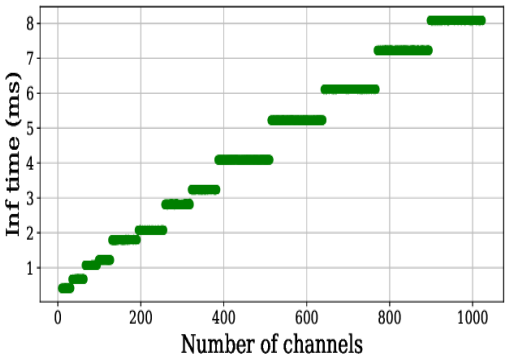

FIGURE 2.9: Staircase Correlation between inference time and number of channels in a ResNet-50 layer on the Jetson TX2. [4].

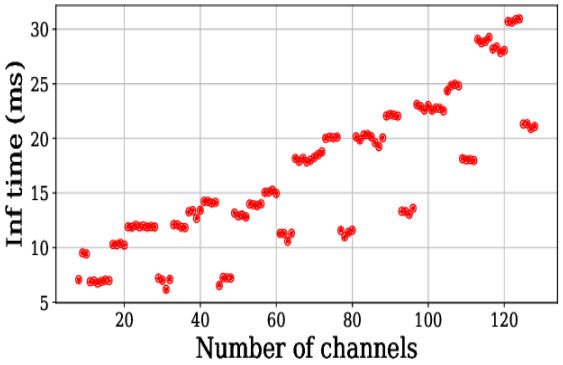

FIGURE 2.10: Inference time of a convolutional layer of ResNet-50 run with Arm Compute Library with varying amounts of channel pruning. [4].

The naïve assumption that pruning channels sequentially always leads to a linear decrease in inference time, regardless of target hardware is shown to be false in the counter example illustrated in Figure (2.9). Here, the experimental results show the effect of filter pruning (for a given layer) on inference time for a specified hardware device. These measurements exhibit a clear staircase correlation in which the inference time remains constant for many subsequent pruning iterations until an abrupt decrease in inference time is observed. This staircase pattern was attributed to the workgroup size on the target device and demonstrates the importance of considering the platform in conjunction with pruning. These gaps can potentially result in large inference time gains for layer configurations with a similar number of channels. Preferably, the number of channels in the convolutional layer should be chosen such that it falls close to the right of the steps observed in this figure. This will yield the highest number of channels for the same inference time.

This observation is not necessarily typical of all embedded GPUs. Figure (2.10) illustrates another pruning experiment which exhibits a parallel staircase pattern. The consequences of not considering the hardware properties are even more significant in this case, as certain amounts of pruning can be detrimental to inference time. This is particularly apparent in the top right of Figure (2.10), here it shows pruning a relatively small number of channels can produce a significant increase in inference time compared with the original network.

This work demonstrates how channel pruning can be used to improve inference times on embedded GPUs but also highlights the need to consider the target hardware when utilising this compression technique. In particular, the optimal amount of pruning for a given layer must be investigated by profiling the kernel execution to avoid the situations in which pruning can be detrimental to performance. In this work, accuracy was not considered in detail, though the authors have begun future works that consider both inference time and accuracy simultaneously.

2.7.4 Discussion

As mentioned previously, the main drawback of sparsity inducing pruning algorithms is that performance benefits cannot be leveraged unless the pruned models are executed

on hardware designed specifically for sparse computation. This severely limits the practicality of this compression technique for deployment of compressed models on widely available hardware. Coarse grained pruning such as channel pruning has been shown to provide performance benefits on general purpose processors, however channel pruning (and pruning in general) can be a time-consuming and computationally expensive procedure. Crowley et al [36] have recently questioned the effectiveness of such pruning. Through experimentation, the authors concluded that

- Smaller architectures trained from scratch are a better option if time is limited.
- The architectures obtained through pruning are valuable and the weights are not!

Interestingly, when the pruned architectures are re-initialised and trained from scratch they out perform their fine-tuned counterparts. Additionally, these smaller architectures can be approximated with relative ease by deriving a family of "copycat" architectures through examination of the connections that remain intact after a pruning. These copycat architectures can be scaled to produce architectures for different memory requirements.

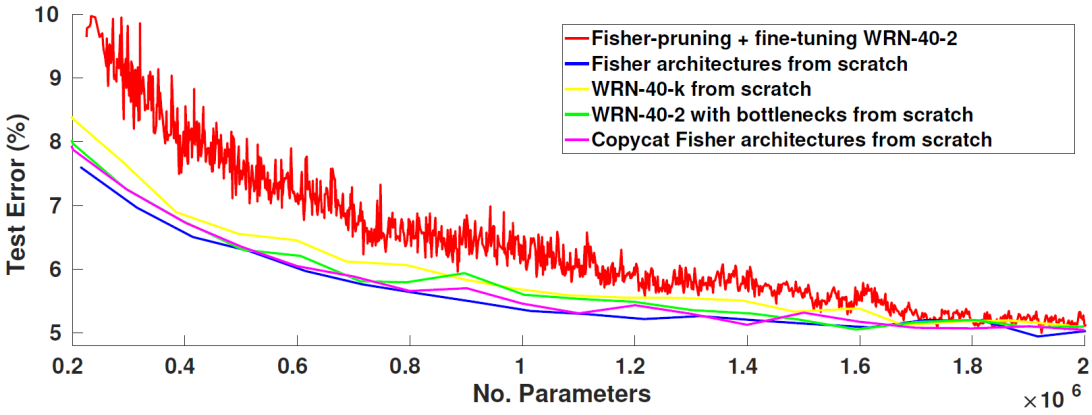

FIGURE 2.11: Comparison of different families of networks on CIFAR-10 [36].

Figure (2.11) demonstrates experimental results from Crowley's work. Here the test error is plotted against the number of parameters for different families of networks. It is clear to see that the simple networks trained from scratch all outperformed the Fisher pruned and tuned networks.

2.8 Knowledge Distillation

Knowledge Distillation is a compression technique concerned with the transfer of knowledge from a large complex model or ensemble of models to smaller network architectures. Here, the large complex models are commonly referred to as the "*teacher*" and the smaller distilled model, the "*student*" [30].

The technique of Knowledge Distillation was first explored by Bucila et al [37] and later extended by Hinton et al [38], who introduced the notion of temperature, motivating the term "Knowledge distillation". In his work, Hinton et al demonstrated this method can produce significantly smaller model architectures while maintaining the accuracy of their larger counterparts.

2.8.1 Temperature

Neural networks often comprise a softmax output layer that produces probabilities of given inputs belonging to each class. These probabilities are calculated using the logits computed for each class in the penultimate layer. Knowledge distillation incorporates the notion of temperature into this softmax calculation as follows: Let i denote an arbitrary index representing a class in the output vector, where $i \in S$ such that S is the set of all class label indices. Then class probability p_i is calculated by comparing logit z_i with the other logits. That is

$$p_i = \frac{\exp\left(z_i/T\right)}{\sum_{j \in S} \exp\left(z_j/T\right)} \tag{2.2}$$

Here T represents the temperature. In the case of normal softmax $T = 1$. Softer probability distributions over classes are obtained by using higher temperatures [38].

Similar to a previous demonstration [39] with a different model and dataset, Figures (2.12, 2.13, 2.14) show the application of the softmax functions (parameterized by different temperatures) to logits produced by AlexNet, given an input image of a Samoyede breed of dog. Shown in Figure (2.12), the standard softmax function has a very high probability density around the correct class with all but a few classes having probabilities close to zero. This provides little information beyond the ground truth labels. As

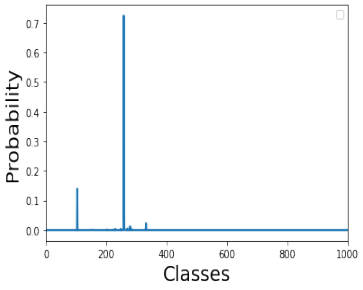

FIGURE 2.12: $T = 1$

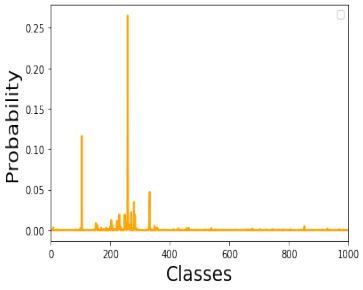

FIGURE 2.13: $T = 2$

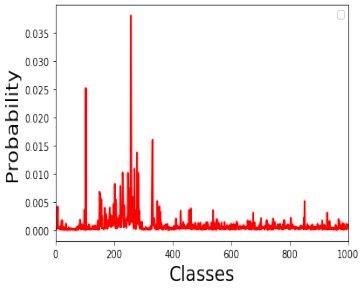

FIGURE 2.14: $T = 4$

can be seen in Figure (2.13, 2.14), increasing the temperature softens the distribution revealing information about which classes the model found most alike. This is useful information which can be transferred to the student using knowledge distillation [38].

2.8.2 Cross Entropy Loss

There are a wide range of loss functions that can be utilised in deep learning. For models trained on a large number of classes and that have a final softmax layer, it is appropriate to use the cross entropy loss function. In this context, cross-entropy loss referrers to the negative log-likelihood of the soft-max distribution with respect to the target distribution (labeled data). In particular, this function measures the similarity of the softmax output vector against the ground truth vector defined by the training set [40]. Given ground truth label vector $\mathbf{y}$ and a softmax prediction vector $\mathbf{p}$, the cross entropy loss is given by

$$\mathcal{H}(\mathbf{y}, \mathbf{p}) = -\sum_{i}^{N} y_i \ln(p_i) \tag{2.3}$$

where N is the number of classes represented in the vector. This function is typically employed for knowledge distillation.

2.8.3 Knowledge Distillation Methodology

Knowledge Distillation employs a loss function containing two weighted objective functions [38]. The first is the cross entropy of the students standard ($T = 1$) softmax out put with the ground truth vector, the second is the cross entropy of the teachers high temperature ($T = \tau$) output with the students high temperature output, these terms are referred to as the student loss and distillation loss respectively.

Using notation borrowed from [30], let $\mathbf{x}$ be an input, $\mathbf{W}$ the parameters of the student network, $\mathbf{y}$ the ground truth vector and let $\sigma(\mathbf{z}; T = \tau)$ be the softmax function applied to logit vector $\mathbf{z}$ parameterized by temperature T=τ. The loss function can then be defined as follows:

$$\mathcal{L}(\mathbf{x}; \mathbf{W}) = \underbrace{\alpha\, \mathcal{H}(\mathbf{y}, \sigma(\mathbf{z_s}; T = 1))}_{Student\ Loss} + \underbrace{\beta\, \mathcal{H}(\sigma(\mathbf{z_t}; T = \tau), \sigma(\mathbf{z_s}; T = \tau))}_{Distillation\ Loss} \tag{2.4}$$

where the subscript s and t identify the student and teacher logit vectors respectively and α and β are arbitrary constants. As described by Hinton et al, α, β and T are all hyperparamters. This process is visualised in Figure (2.15).

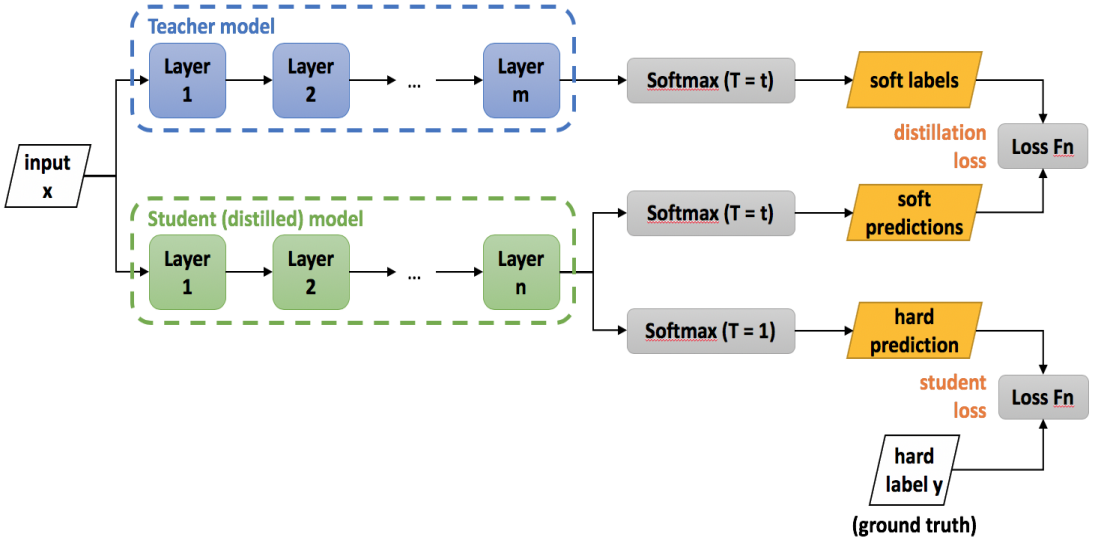

FIGURE 2.15: High Level Diagram of the Knowledge Distillation Process [30].

2.8.4 Attention Transfer

Attention transfer [41] is a form of Knowledge Distillation where attention maps of the teacher network are transferred to the student at a number of discrete attention points. Using the examples presented in [5], we explore this concept further.

Figure (2.16) illustrates a block diagram of a WideResNet where outputs of activation groups have been selected as attention points (1,2,3). Activations are converted into attention maps by squaring each element and taking an average along the channel dimension. This produces a matrix where each entry encodes a mapping to a spatial location. This matrix is then L-2 normalised.

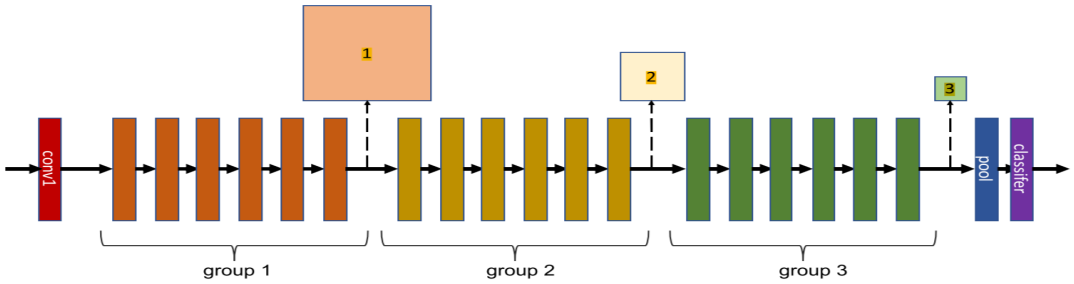

FIGURE 2.16: Block Schematic of WideResNet [5].

Similar to standard Knowledge Distillation, the loss is calculated using cross-entropy but Attention Transfer incorporates and additional term that penalises the L-2 distance between the attention maps of the teacher and student. This additional term can be weighted by altering its coefficient.

2.8.5 Contrastive Representation Distillation

Contrastive Representation Distillation (CRD) is a relatively new development in state-of-the-art distillation based compression techniques that has been shown to outperform knowledge distillation, attention transfer and a wide range of other distillation based techniques [25]. This technique incorporates aspects of two different areas of research, namely *knowledge distillation* and *representation learning*. While knowledge distillation attempts to minimise the cross entropy loss of the student and teacher logits, CRD also attempts to transfer representational knowledge. This type of knowledge is highly structured with complex dimensional independencies and is unable to be transferred using knowledge distillation. As such, the authors proposed a contrastive-based approach to capture correlations and higher order output dependencies that form this structured knowledge.

By utilising powerful methods from representation learning, the authors were able to improve knowledge transfer from the teacher to the student. To demonstrate how well this method performs compared to 12 other distillation based techniques, the authors performed benchmark testing across a range of teacher-student combinations. Table (2.2) presents the results of one set of experiments in which the student models have a different architecture type to that of the teacher. Here, we see that CRD outperforms all other methods with knowledge distillation performing second best overall.

Teacher Student	vgg13 MobileNetV2	ResNet50 MobileNetV2	ResNet50 vgg8	resnet32x4 ShuffleNetV1	resnet32x4 ShuffleNetV2	WRN-40-2 ShuffleNetV1
Teacher	74.64	79.34	79.34	79.42	79.42	75.61
Student	64.6	64.6	70.36	70.5	71.82	70.5
KD*	67.37	67.35	73.81	74.07	74.45	74.83
FitNet*	64.14 (↓)	63.16 (↓)	70.69 (↓)	73.59 (↓)	73.54 (↓)	73.73 (↓)
AT	59.40 (↓)	58.58 (↓)	71.84 (↓)	71.73 (↓)	72.73 (↓)	73.32 (↓)
SP	66.30 (↓)	68.08 (↑)	73.34 (↓)	73.48 (↓)	74.56 (↑)	74.52 (↓)
CC	64.86 (↓)	65.43 (↓)	70.25 (↓)	71.14 (↓)	71.29 (↓)	71.38 (↓)
VID	65.56 (↓)	67.57 (↑)	70.30 (↓)	73.38 (↓)	73.40 (↓)	73.61 (↓)
RKD	64.52 (↓)	64.43 (↓)	71.50 (↓)	72.28 (↓)	73.21 (↓)	72.21 (↓)
PKT	67.13 (↓)	66.52 (↓)	73.01 (↓)	74.10 (↑)	74.69 (↑)	73.89 (↓)
AB	66.06 (↓)	67.20 (↓)	70.65 (↓)	73.55 (↓)	74.31 (↓)	73.34 (↓)
FT*	61.78 (↓)	60.99 (↓)	70.29 (↓)	71.75 (↓)	72.50 (↓)	72.03 (↓)
NST*	58.16 (↓)	64.96 (↓)	71.28 (↓)	74.12 (↑)	74.68 (↑)	74.89 (↑)
CRD	**69.73** (↑)	**69.11** (↑)	**74.30** (↑)	**75.11** (↑)	**75.65** (↑)	**76.05** (↑)

TABLE 2.2: Top-1 test accuracy (%) of student models on CIFAR100 for a range of distillation based methods [25].

2.8.6 Discussion

Experiments conducted by Hinton et al on the MNIST dataset show that knowledge from a strongly regularised teacher model could be transferred to a significantly smaller student model with a negligible drop in accuracy. Further experiments exploring application to the speech recognition domain also showed promising results. In all these experiments, hyperparameter values were chosen through trial and error with only a handful of combinations being used. The search landscape for an optimisation problem to find the best combinations of these parameters could be an interesting avenue of research, though this could be severely limited by computational overhead. Methods such as particle swarm optimisation may be useful in the search, given teacher/student models were sufficiently small. Similarly, little discussion regarding the choice of student architecture is provided. Exploring this problem faces similar challenges to the one previously discussed, though finding the most optimal student architecture for a particular problem is an interesting concept.

Further research [42] examining the efficacy of Knowledge Distillation has revealed some important findings that should be considered when utilising this compression technique. The first finding stipulates that "*larger models do not always make better teachers*". Results of an empirical study strongly support the hypothesis that this finding is a result of miss-matched capacity between the student and the teacher. More specifically,

the small student model is unable to retain knowledge from the large teacher models due to its inferior size.

Secondly, the authors discovered "*while KD loss improves validation accuracy initially, it begins to hurt accuracy towards the end of training*". Knowledge distillation typically achieves poor results on the ImageNet dataset with the authors showing that a range of selected teacher-student combinations lead to lower student accuracy when compared to the student model trained from scratch. The authors showed that, due to the more challenging nature of the ImageNet problem, the low-capacity student suffers from under-fitting. In particular, the students capacity to minimise both the distillation and student loss simultaneously is somewhat limited and as a result, one loss is minimised at the expense of the other, particularly towards the end of training. This is illustrated in Figure (2.17) with a knowledge distilled student being compared to one trained from scratch. As shown, knowledge distillation performs better in the early stages of training and for complex problems such as ImageNet, becomes detrimental in the later stages.

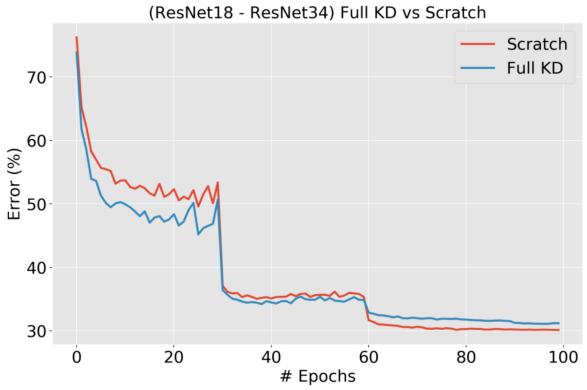

FIGURE 2.17: Comparison of Knowledge Distillation and standard training on the ImageNet Dataset. [42]

2.9 Multi-Objective Optimization

Multi-Objective Optimization problems are typically used to solve optimisation problems with at least two conflicting objectives. In particular, for computer vision applications deployed on resource constrained devices, these conflicting objectives are typically accuracy and inference latency. Generally, these objectives are simultaneously optimised

through some form of neural network architecture search using machine learning techniques such as evolutionary algorithms or reinforcement learning. Metrics are typically measured on the target hardware as optimisations made for deployment on a specific device may not be optimal for other hardware architectures. These techniques tend to be highly computationally intensive but also automate the DNN design process which removes the requirements for human resources.

As discussed in Section (2.2), a large amount of research has been dedicated to designing highly accurate deep learning models. Optimising this single objective generally leads to very large models resulting in very large inference times. One aim of neural network compression is to optimise accuracy (minimize error) while reducing the inference time. Both of these objectives are conflicting, as improvement in one objective generally has a negative impact on the other objective and vice versa. As such, problems of this nature can be defined by the following multi-objective optimisation problem (MOP):

$$min_{a \in \mathbf{A}}(f_1(a), f_2(a)) \tag{2.5}$$

where $\mathbf{A}$ is the DNN solution space: a set containing all permissible DNN configurations defined for a particular problem, the objective function f_1 is a measure of the test error of model a and f_2 is a measure of the inference latency of model a. In contrast to single objective optimisation problems, MOPs do not have a single solution but rather a set of solutions, each of which represent an optimal trade off between multiple objectives. That is, improvements to one objective cannot be made without worsening at least one other objective. This set of solutions is widely referred to as the *Pareto optimal set* or *Pareto optimal front* when plotted in the objective space. The objective space is a multi-dimensional space in which each axis represents a distinct objective from a MOP. MOP solutions are often compared using a notion of dominance. For a MOP with two objectives, a solution a, is said to dominate solution b if and only if:

$$\forall i \in [1,2] : f_i(a) \leq f_i(b) \text{ and } \exists\, [1,2] : f_i(a) < f_i(a) \tag{2.6}$$

That is, a is better than b in at least one objective and no worse in all other objectives. Pareto optimal solutions are not dominated by any other solution in the solution space. A solution, a, is said to be Pareto optimal if and only if:

$$\forall b \in \mathbf{A}, \forall i \in [1,2], f_i(a) \leq f_i(b) \text{ and } f(a) \neq f(b) \tag{2.7}$$

2.9.1 Multi-Objective Evolutionary Algorithms

This section introduces the fundamental components common to the majority of Genetic Algorithms (GAs) as described in [43]. GAs are a subclass of evolutionary algorithms an have single and multi-objective variants.

The term evolutionary algorithm encompasses a wide range of algorithms, all of which aim to gradually produce better solutions over discrete time steps called generations. To remain within the scope of this project, we shall only consider the subset of EAs called Genetic Algorithms (GAs). This class of population based algorithm are inspired by Darwin's theory of evolution. Each individual in a population represents a solution in the solution space and is encoded as a genotype (or chromosome). This genotype consists of multiple genes, each of which correspond to a parameter of the optimisation problem. The three fundamental components employed by GAs are described below:

- **Selection** - In the natural world, the fittest members of a population have the highest chance of reproducing. As a result, their genes are more prevalent in the subsequent generation. Similarly, for GAs, each solution in the population is evaluated by a fitness function which assigns each genotype a fitness value. Solutions with better fitness values have a higher probability of being selected to form the next generation through the application of crossover and mutation operators.

- **Crossover** - Once solutions have been selected, they are used to create a new generation. Similar to the natural world, the genes of two parent solutions are combined to produce two new child solutions. Many different techniques can be used to simulate crossover. Two common methods are illustrated in Figure (2.18).

- **Mutation** - Members of the new population are subject to mutations in which one or more genes are altered at random. The inclusion of mutation helps maintain diversity in the new population. Similar to crossover, there are a range of approaches that can be taken to simulate mutation. An example is shown in Figure (2.19).

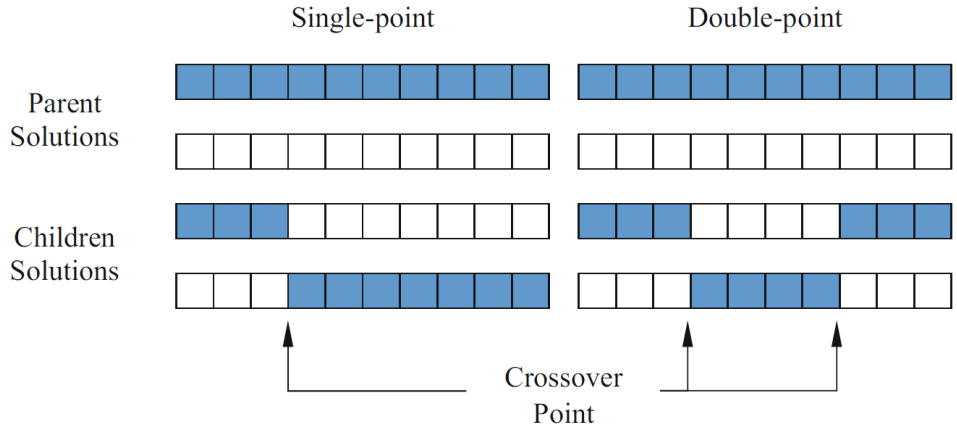

FIGURE 2.18: Single-point and Double-point Crossover Operations [43].

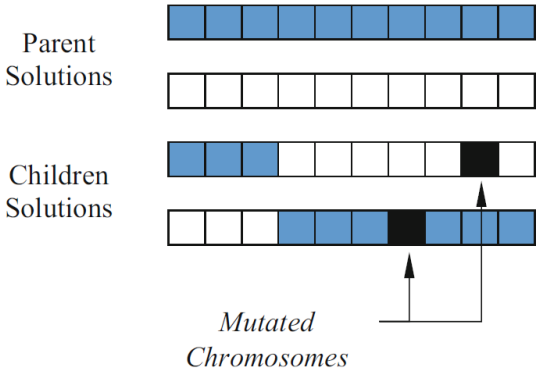

FIGURE 2.19: An example of the mutation operator altering multiple genes in the child solutions after crossover [43].

Through application of these operators, GAs aim to produce better populations, generation by generation. The fundamentals of single-objective and multi-objective evolutionary algorithms (MOEAs) are much the same. Two distinct differences are that:

1. MOEAs have more than one single measure of fitness determined by their multiple objective functions.

2. MOEAs do not produce a single solution, but rather an optimal set of non-dominated solutions that represent an optimal trade off between solutions in the objective space.

2.9.1.1 NSGAII

The Non-Dominated Sorting Genetic Algorithm version II (NSGAII) is a popular MOEA that has had much success in a wide range of multi-objective optimisation problems. It is an elitist algorithm that constructs a breeding pool by combining the parent and child population to select the best solutions with respect to fitness and spread [44]. The algorithm works by storing the best n individuals that have been discovered so far in an archive, A. The new population, P, is bread from solutions stored in A by application of crossover and mutation operators. All solutions in $A \cup P$ then compete to be added to, or remain in the archive. Spread is the measure of the distance between neighbouring solutions in the objective space and solutions with a larger distance between its neighbours are given a higher selection priority to encourage diversity along the pareto optimal front. A notion of rank is used to determine whether solutions remain in the archive. The algorithm first identifies all the non-dominated solutions in the breeding pool and assigns these a rank value of 1. These solutions are temporarily disregarded and the non-dominated solutions in remaining pool are selected and assigned a rank value of 2. This process is repeated until every member of the pool has been assigned a rank. Solutions are then selected based upon their rank and spread, those with the lowest rank and highest spread are prioritised for selection. This process is typically repeated for a specified number of generations, or until a desired solution is obtained [45]. An abstract version of the NSGAII algorithm is provided in Figure (2.20).

```
m ← desired population size
a ← desired archive size                                   ▷ Typically a = m

P ← {P_1, ..., P_m} Build Initial Population
A ← {} archive
repeat
    AssessFitness(P)                  ▷ Compute the objective values for the Pareto front ranks
    P ← P ∪ A                         ▷ Obviously on the first iteration this has no effect
    BestFront ← Pareto Front of P
    R ← Compute Front Ranks of P
    A ← {}
    for each Front Rank R_i ∈ R do
        Compute Sparsities of Individuals in R_i          ▷ Just for R_i, no need for others
        if ||A|| + ||R_i|| ≥ a then   ▷ This will be our last front rank to load into A
            A ← A ∪ the Sparsest a − ||A|| individuals in R_i, breaking ties arbitrarily
            break from the for loop
        else
            A ← A ∪ R_i                                   ▷ Just dump it in
    P ← Breed(A), using Algorithm 103 for selection (typically with tournament size of 2)
until BestFront is the ideal Pareto front or we have run out of time
return BestFront
```

FIGURE 2.20: Abstract version of the NSGAII algorithm [46].

2.9.2 Multi-Objective Optimisation of Deep Neural Networks

This section reviews relevant literature related to the multi-objective optimisation of deep neural networks on differnt hardware platforms.

2.9.2.1 Neuro-Evolution with Multi-objective Optimisation of Deep Neural Networks

Neural-Evolution with Multi-objective Optimisation (NEMO) is an automatic machine learning technique that uses MOEAs to simultaneously optimise both accuracy and execution-time of DNN models by evolving their architecture [1]. Empirically, it has been shown that this method discovers faster and more accurate models compared to those designed by humans. Some evolutionary deep learning approaches [47] have replaced stochastic gradient descent (SGD) with evolutionary algorithms, however, due to the enormity of the search space typical of state-of-the-art DNNs (millions of parameters), EAs can often fail to discover optimal solutions. As such, recent neuro-evolution techniques retain SGD and back propagation for training while using EAs or MOEAs to search for optimal architectural configurations.

The NEMO approach encodes the number of outputs or channels for each layer of a pre-defined CNN as a genotype and in some cases, the number of layers are flexible as well. To evaluate a given genotype, it is first decoded and the information is used to construct a corresponding CNN architecture, which is then subject to standard deep learning training and testing procedures. Upon completion of training, the two objective values (test accuracy & execution time) are measured. The algorithm employed for multi-objective optimisation used in this methodology is NSGAII, as this ranks evolved solutions in the population using non-dominated sorting and maintains a balance of exploration and exploitation throughout the evolutionary process.

A high level overview of this methodology is shown in Figure (2.21). The left side illustrates the workflow which is typical of most MOEAs. The right side illustrates the decoding, training and evaluation procedure. Figure (2.22) provides more detail of the decoding procedure and shows a genotype representation with its corresponding CNN (phenotype). Here, the genotype is represented as an array of integers with upper

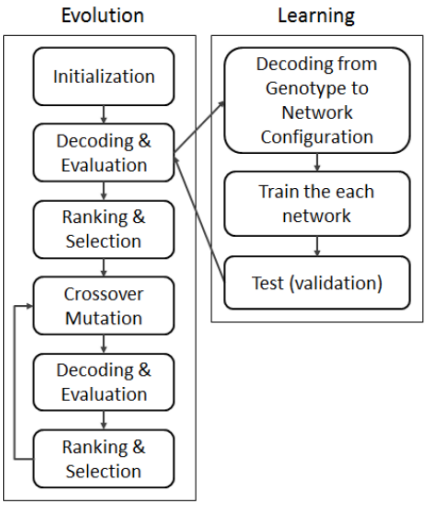

FIGURE 2.21: NEMO Workflow [1].

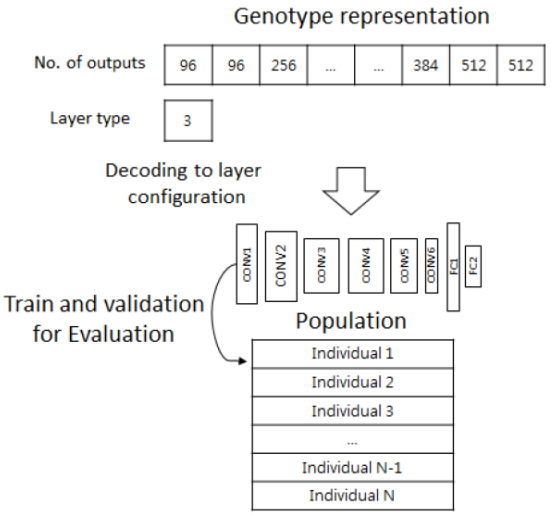

FIGURE 2.22: NEMO decoding procedure [1].

bounds. The integer values determine the number of outputs for each layer, and in some cases, the number of layers.

Experiments conducted using this methodology on three different datasets all produced architectures that were better in terms of both accuracy and latency than the baseline architecture from which they evolved, even after a relatively small number of generations. Faster convergence was, however, encouraged by initialising each member of the initial population to a baseline network that already demonstrated good performance on the problem specific dataset.

2.9.2.2 Device-aware Progressive Search for Pareto-Optimal Neural Architectures

Device-aware Progressive Search for Pareto-Optimal Neural Architectures (DPP-Net) [3] is another method of neural architecture search that has been shown to simultaneously optimise device-related objectives such as inference time and device-agnostic objectives such as accuracy. This search algorithm uses progressive search and mutation operators to explore the trade-offs between these objectives.

Applying this algorithm to problems on a range of different hardware devices from a NVIDIA Titan X GPU to a mobile phone with an ARM Cortex-A35, the authors were

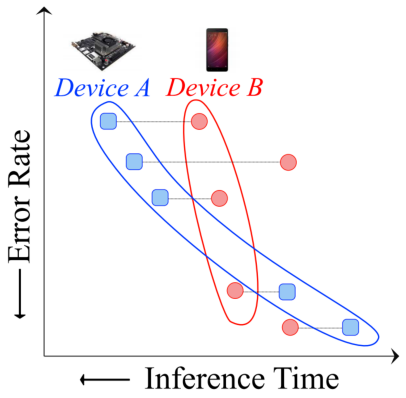

FIGURE 2.23: An example of how different hardware architectures can share different Pareto-Optimality [3].

able to obtain higher accuracy and shorter inference times compared to the state-of-the-art CondenseNet [48]. What is more, the authors highlighted the important fact that pareto optimal models discovered on one device are not necessarily guaranteed to have good performance on another device. An illustrative example is provided in Figure (2.23) and shows how an optimal solution on Device A's pareto front does not lie on Device B's front. As has been mentioned previously, a small number of FLOPs does not always indicate fast inference time. This behavior is likely a result of varying hardware optimisations and software implementations accross different platforms.

To show that inference time is highly device related, the author measured the inference time for a range of different 4-layer DNN configurations on three devices. Namely, these devices comprise a work station (WS) with a NVIDIA Titan X GPU, an embedded systems (ES) GPU (NVIDIA Jetson TX1) and finally a mobile (Mobile) device with an ARM Cortex-A35 CPU. Figure (2.24) plots the inference times of these models with the x-axis representing the index of all the DNN configurations, sorted by inference time on the WS in ascending order. As shown, inference time on devices that are similar in architecture and have identical software setting can still be sensitive to a specific device. As a result of this empirical evaluation, it was concluded that multi-objective neural architecture search should be conducted on the target hardware to ensure the robustness of the results.

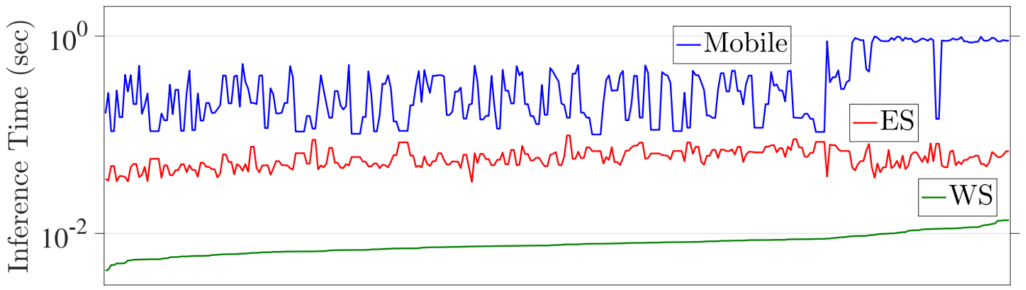

FIGURE 2.24: Inference time measured on different hardware platforms [3].

2.10 Combining Compression Techniques

Successful attempts combining Knowledge Distillation with Quantisation have been shown to achieve accuracy comparable with state-of-the-art full-precision models. Polino et al [29] presented two such methods in which knowledge is transferred to shallower quantised architectures. This reduction in depth automatically improves inference times and further reduction is gained from quantisation. The results presented by the authors were obtained through manual architecture and bit-width search, which was acknowledged to be time consuming and error prone. Examining reinforcement learning and evolutionary strategies to find optimal student architectures have been highlighted as possible future works.

Other work aimed at improving inference time through a combination of compression techniques has also shown promise in recent years. Turen et al [5] have proposed a method which utilises both pruning and knowledge distillation to achieve superior performance over pruning alone. In particular, the authors devised a method to discover performance enhanced student models using a two-step process:

1. This step utilises Fisher-Pruning and searches the pruning space on a given target hardware device to determine a latency profile for each prunable layer in the original model. This process begins by benchmarking latency for a single inference on the original model before pruning, then iteratively removing one filter at a time and recording the corresponding inference time. Optimal points are then identified from the results obtained from the empirical search. As previously discussed in Section (2.7.3), these optimal points occur where a maximum number of channels

can be retained for a given inference time. That is, the right of each step illustrated in Figure (2.25). A resultant Fisher-pruned model is then modified such that the number of channels in each layer is chosen to be as close as possible to the optimal points in this search space.

2. The performance enhanced architectures produced by the first step then have their parameters randomly re-initialised and are subsequently distilled using attention transfer with the original model as the teacher.

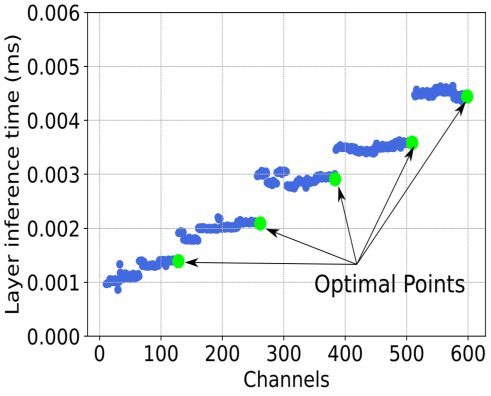

FIGURE 2.25: Inference time for a layer of ResNet-34 plotted against the number of channels in that layer on an ARM Cortex-A57 [5].

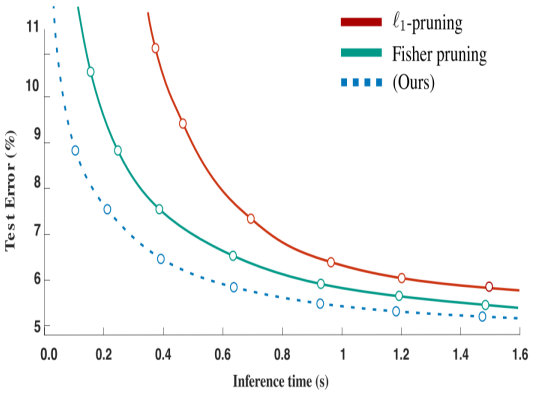

FIGURE 2.26: CIFAR-10 test error plotted against inference time for a WideResNet-40-2 on the ATM Cortex CPU [5].

Figure (2.26) illustrates the effectiveness of this method in comparison to two other popular pruning methods. Here we see that this method attains both better accuracy and inference time compared to other popular techniques. Similar results are obtained on other hardware platforms using a wide range of models.

2.11 Summary and Open Challenges

Of all the hardware devices discussed in Section (2.4.2), the NCS2 implementation of the Myriad X VPU offers the best trade-off between performance and power consumption whilst providing a high level API that makes deploying models from the most popular DL frameworks relatively simple. These properties make it an ideal choice of hardware for scenarios where power is in limited supply. While this device can already facilitate computer vision models in real time, simultaneously optimising inference time and accuracy on this device would enable more sophisticated applications to be deployed

with more power efficiency. As such, further research investigating optimal compression methods for this type of platform would undoubtedly be of value.

As we have seen, there are a wide number of compression techniques that have been developed to improve the performance of DNN models deployed on various hardware platforms. The degree to which quantisation can be employed depends on the hardware's support for reduced precision computation. In the case of the NCS2, only FP16 precision models are supported. As such, there is little scope to pursue this type of compression to enhance models deployed on this platform. Unfortunately fine-grained pruning techniques are also redundant for the NCS2, as this platform has not been designed specifically for sparse tensor computation. This leaves coarse-grained pruning, knowledge distillation based methods and multi-objective optimisation as the main remaining candidates to enhance the performance of models deployed on the NCS2.

Coarse-grained pruning techniques discussed in Section (2.7.2) and Section (2.7.3) do provide performance benefits on various hardware platforms, however, these algorithms typically perform better as a form of neural architecture search (NAS), with pruned-and-tuned models often being outperformed by the corresponding model trained from scratch. When considering pruning as a form of NAS, it is limited in that it can only remove a single type of structure (channels/filters) from a model. Other popular NAS methods are capable of doing this and much more, potentially enabling a more optimal architecture to be found. For example, a faster model may be obtained by removing an entire layer while adding output channels to another.

Distillation based compression is typically a device-agnostic method that can be used to significantly reduce model inference time without compromising accuracy. This, however, is based on the premise that knowledge from a large teacher model can be successfully transferred into a smaller student model. While this approach has proven to be successful on relatively simple datasets such as MNIST, more complex datasets such as ImageNet pose a greater challenge. There is strong evidence to support that the reason for this is a mismatch in capacity between teacher and student models (Section 2.8.6) and this highlights the difficulty of choosing an appropriate student model. Currently, there is no method to determine what capacity for information a student model will have without actually testing it. As such, creating an optimal student model manually is likely to be a time-consuming task that requires a high level of expertise and experience

in DNN design. To add to the challenge, the effectiveness of these distillation methods are often dependant on associated hyper-parameters. On simple problems, good combinations of knowledge distillation hyper-parameters can be predicted relatively easily based on patterns observed from empirical findings presented in [38]. More complex datasets, however, require these parameters to be tuned for a specific problem. This further increases the complexity of obtaining optimal results from this family of compression techniques. As such, finding an optimal student with optimal hyper-parameters is expensive in terms of both human and computational resources.

The multi-objective optimisation methods discussed in Section (2.9.2) provide a more versatile type of NAS compared to coarse-grained pruning and can be employed to find a range of architectures that offer an optimal trade-off between inference time and accuracy on a specific hardware platform. Being able to select a pareto optimal model from a range of solutions enables a developer to choose the solution that best fits the application requirements. One of the most significant limiting factors to these types of multi-objective NAS problems is the computational overhead. For example, NEMO fully trains and tests hundreds and in some cases thousands of DNN models in order to find optimal models. As a result, these methods typically require a large quantity of GPUs to provide good results in a reasonable time frame.

Section (2.10) highlighted the effectiveness of combining different compression methods. Drawing inspiration from this line of research and upon review of other relevant literature, it is apparent that many limitations of knowledge distillation and NEMO has potential to be addressed by combining these methods. In particular:

1. Evolving student models may allow their architectures to expand or contract to adapt their capacity for retaining knowledge distilled from teacher models.

2. Instead of tuning the knowledge distillation hyper-parameters, student models may evolve to accommodate any reasonable choice, reducing the computational overhead involved in tuning these parameters.

3. Automating the production of student models that have been optimised for deployment on a hardware specific platform would remove the need to manually build, train and test an ensemble of student models, ultimately saving valuable human resources.

4. As demonstrated previously in Figure (2.17), knowledge distillation has the most significant impact in the early stages of training. Allowing NEMO to profile student models based upon their performance during these early stages of training could significantly reduce the amount of computation required to execute this algorithm.

The remainder of this thesis will focus on investigating the efficacy of combining these two techniques by conducting an empirical evaluation. In doing so, we hope to determine whether our intuition is correct. It should also be noted that other distillation techniques such as CRD could be substituted for knowledge distillation but their increased complexity make encoding the solution space to evolve a student model inherently more difficult.

Chapter 3

Methodology

This chapter provides implementation details and a high level overview of the methodology we have employed to evolve faster and more accurate models for deployment on the Myriad X VPU.

3.1 Overview and Intuition

The methodology employed in this project largely follows the Neuro-Evolution with Multi-objective Optimisation (NEMO) approach presented in [1] and discussed in Section (2.9.2.1), but with three important distinctions:

1. Knowledge Distillation (KD) replaces standard training in the learning phase of the evaluation procedure.
2. To conserve time and computational resources in the learning phase, partial training is provided as opposed to training each model fully. Upon termination of the algorithm, selected individuals from the final generation can be provided with "Extra Tution" (Section 3.2.1).
3. The chosen objective values, inference *test error* and *latency*, are measured on a specific hardware platform, namely the Myriad X VPU.

This approach combines well established compression techniques and is intended to provide an automated method to design deep learning models for deployment on hardware

specific platforms with memory constraints. To find an appropriate solution using KD alone, one may have to experiment with a wide range of student architectures, and for each student architecture, the knowledge distillation hyper parameters must then be tuned for each specific configuration to obtain an optimal solution. Such a methodology does not explore potential architectural sweet spots that may fall in between two student architectures. NEMO, however, has been shown to simultaneously optimise accuracy and inference speed by evolving model architectures [1]. Since both these metrics can be of equal importance when designing computer vision models for memory constrained hardware, finding architectures that provide the optimal trade off between these two conflicting objective values is an interesting and worthwhile pursuit. This technique has potential to be improved further if used in conjunction with other compression techniques. Extending NEMO to include knowledge distillation was motivated by a limitation of the latter. In particular, student architectures are fixed and " *there is no clear way to predict up front what kind of capacity for information the student model will have*" [5]. By combining the aforementioned techniques, student architectures will have the potential to evolve and adapt their capacity for retaining knowledge distilled from highly accurate teacher models with significantly greater memory footprints relative to that of the student.

3.2 NEMO with Knowledge Distillation: NEMOKD

Figure (3.1) illustrates a high level overview of our modified NEMO implementation. The left hand side illustrates the evolutionary process. The middle and right sections illustrate the modifications we have made to the evaluation procedure. Namely, we have replaced the standard learning process with Knowledge Distillation and measured the objective values on a specific hardware platform.

Analogous to the standard NEMO procedure, we have employed the NSGAII algorithm to facilitate evolutionary multi-objective optimisation. Evolutionary algorithms generally require an optimisation problem to be encoded as an array of values. Such an array is commonly referred to as a genotype or chromosome. The fundamentals of our NEMOKD algorithm are described as follows:

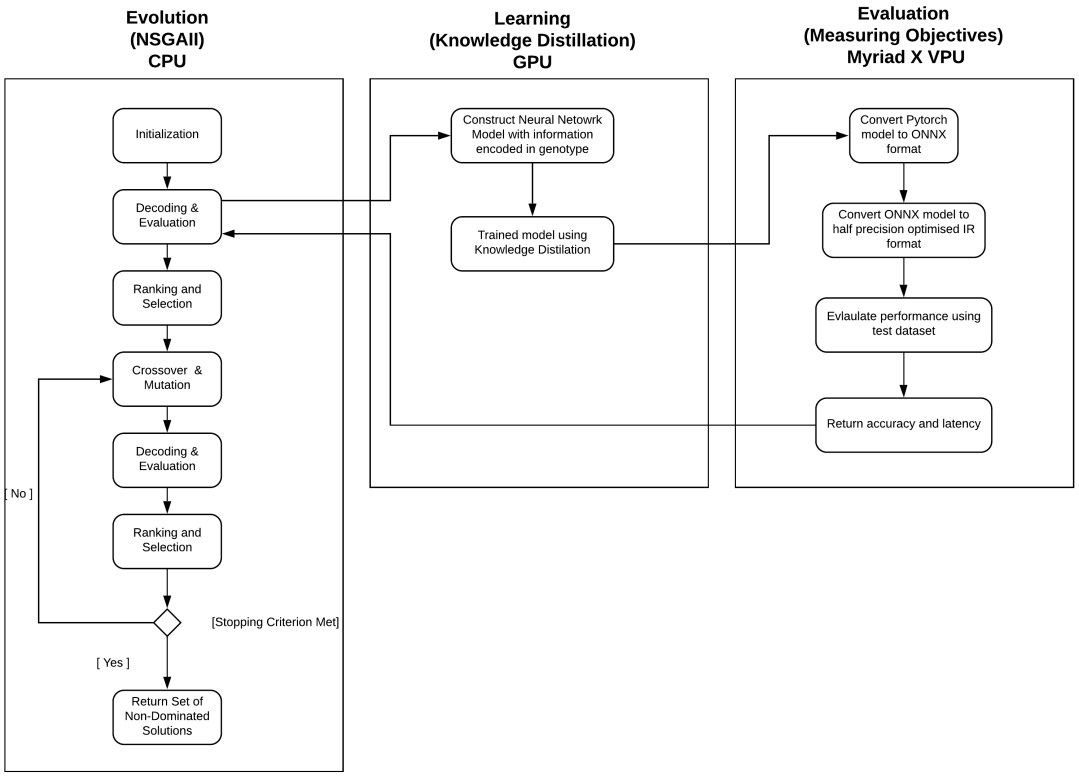

FIGURE 3.1: High level overview of the modified NEMO methodology. Adapted from [1].

1. A population of size N is initialized. Each member of this population is a genotype that represents a solution in a defined solution space. For this particular problem, each genotype represents a specific CNN architecture in the solution space defined for a particular experiment (Section 3.3). Initialisation can be random or specified. Our implementation initialises each member of the initial population to encode a baseline architecture that has been shown to perform reasonably well on the CIFAR10 dataset. Random perturbations are then applied to half the population to encourage more diverse solutions. This choice of initialisation is intended to increase the chance of finding comparable or better solutions in a relatively small number of generations.

2. The fitness of each individual genotype is then assessed according to some specified criterion. In muti-objective problems, both objective values are used to determine the fitness of a particular genotype. Our method executes the following steps to measure the objective values:

   (a) Each genotype passed to the fitness function are first decoded (Section 3.3)

and a CNN model is constructed using the information encoded in the genotype.

(b) The model is then partially trained. Training is, however, aided by a larger teacher model using Knowledge Distillation. As discussed in Section (2.8), there are three Knowledge Distillation hyper-parameters. In this implementation, we set $\beta = 1 - \alpha$, which is typical across the literature.

(c) Once partial training is complete, the model undergoes a number of transformations to facilitate deployment on the Myriad X VPU. The PyTorch model is first converted to the ONNX format before being converted to a half precision OpenVINO IR format. In this format, the model is then loaded onto the VPU and inference accuracy and latency are measured. These two measurements comprise the objective values that are passed to the NSGAII algorithm.

3. A number of genotypes are then selected, based on their fitness values, to form part of a new population.

4. Crossover and Mutation operators are applied to members of the selected population to form the remaining part of the the new population, P.

5. For each iteration of the algorithm or *generation*, each member of the new population is evaluated using the procedure outlined in (2).

6. Steps (2) - (5) are repeated until some stopping criterion is met. In our implementation, the algorithm will terminate after a specified number of generations.

7. The algorithm outputs the final population including Pareto Optimal Set. This set of solutions provides optimal trade-offs between the two objectives in the objective space. Selected individuals from the Pareto Optimal Set that meet latency requirements for an intended use case can then be sent for "Extra Tuition" as described in (Section 3.2.1).

3.2.1 Extra Tuition

To convey the intuition behind the decision to use partial training in our modified NEMO methodology, the following analogy may be helpful. Pilot specialisations in the RAF are

largely determined by performance in Elementary Flying Training: the initial phase of pilot training. Cadets or students who show the greatest potential in this initial training phase are selected to fill the most demanding roles, which are often the most costly in terms of training. The premise behind this decision making is that student performance in early stages of training is indicative of performance in later stages of training and it makes sense to invest the most in the students with the most potential. We have adopted a similar approach to training each generation of student architectures in our modified NEMO approach. Since our methodology is a very computationally expensive procedure (even more so that standard NEMO), it is not practical to train each model fully in the training phase. As discussed in [42], knowledge distillation has been shown to have the greatest impact in the early stages of training. As such, the decision was made to train each student model for only 30 epochs in the learning phase (Figure 3.1). Upon termination of the algorithm, the "elite" student models in the pareto optimal set can be selected to undergo a full training regime. In summary, the algorithm profiles students based upon their performance in early stages of training under the assumption that students who perform best at an early stage of training will perform best after being subject to a complete training regime.

3.3 Encoding the Solution Space for CNN Evolution

3.3.1 FlexStudent Solution Space

To facilitate CNN evolution, an appropriate solution space must first be defined. This solution space defines the permissible values the encoded hyper-parameters can take. Evolving a set of well performing CNNs from scratch in a small number of generations is extremely unlikely due to the enormous size of the solution space. To increase the chance of finding favorable solutions, we decided to only allow some key architectural hyper-parameters to be modified by the evolutionary process. Additionally, we used a simple five layer configuration to provide a starting point in the design of our solution space, as a similar configuration has been shown to perform well as a student architecture on the CIFAR10 dataset [1]. This PyTorch model was modified to enable the key architectural properties of the network to be specified by passing the decoded genotype to the

[1] https://github.com/peterliht/knowledge-distillation-pytorch

FlexStudent Hyper-parameters			
Parameter	**Fixed**	**Value / Range**	**Step Size**
Kernel Size	Yes	3×3	N/A
Padding	Yes	1	N/A
Stride	Yes	1	N/A
MaxPool Stride	Yes	2	N/A
No. Conv Layers	No	$[2, 5]$	1
No. FC Layers	No	$[2, 5]$	1
C[i] out channels	No	[6, 240]	6
FC[j] neurons	No	[25,1000]	25

TABLE 3.1: FlexStudent Hyper-parameters.

PyTorch model object upon instantiation. We named this flexible model FlexStudent. Table (3.1) provides the hyper-parameters of our FlexStudent model and specifies which parameters are fixed, and which are subject to modification throughout the evolutionary procedure. It should be noted that the first block of layers are always convolutional layers and the last block of layers are always fully connected layers. For example, if the decoded genotype specified the number of convolutional layers to be $N \in [2, 5]$ and the number of fully connected layers to be $M \in [2, 5]$, then the first N layers will be convolutional and the last M layers will be fully connected. Additionally, for each convolutional layer, the number of output channels can be specified independently of the other convolutional layers. The output of each convolutional layer is always followed by Batch Normalisation and Max Pooling. The permissible number of output channels, say c, for the ith convolutional layer ranges from 6 to 240 with discrete incremental steps of 6. That is $\forall c, c \in \{6, 12, 18, \ldots, 240\}$. Similarly, the permissible number of neurons in the fully connected layers, say f, for the jth fully connected layer ranges from 25 to 1000 with discrete incremental steps of 25. That is $\forall f, f \in \{25, 50, 75, \ldots, 1000\}$.

3.3.1.1 FlexStudent Genotype Encoding

A vast number of different FlexStudent configurations can be represented by an a genotype consisting of 11 genes. Programmatically, this genotype takes the form of an array, with the values at each index determining the value of a corresponding hyper-parameter. Illustrated in Figure (3.2), the values at index 0 and 1 determine the number of convolutional and fully connected layers respectively. These can be thought of regulatory genes that determine which of the subsequent genes are on or off. The indices, $i \in [2, 6]$ determine the number of output channels in convolutional layers corresponding to each

index. Similarly, indices $i \in [7, 11]$ encode the number of neurons in the fully connected layers. Since the number of neurons in the first fully connected layer is determined by the number of convolutional layers and the number of output channels in the last convolutional layer, the genotype does not contain a gene that specifies the number of neurons in this layer. As a result, the number of neurons can only be specified for a maximum of four layers, even if there are five fully connected layers. For example, if an integer value of M is specified at index 1, then $(M - 1)$ fully connected genes are "on".

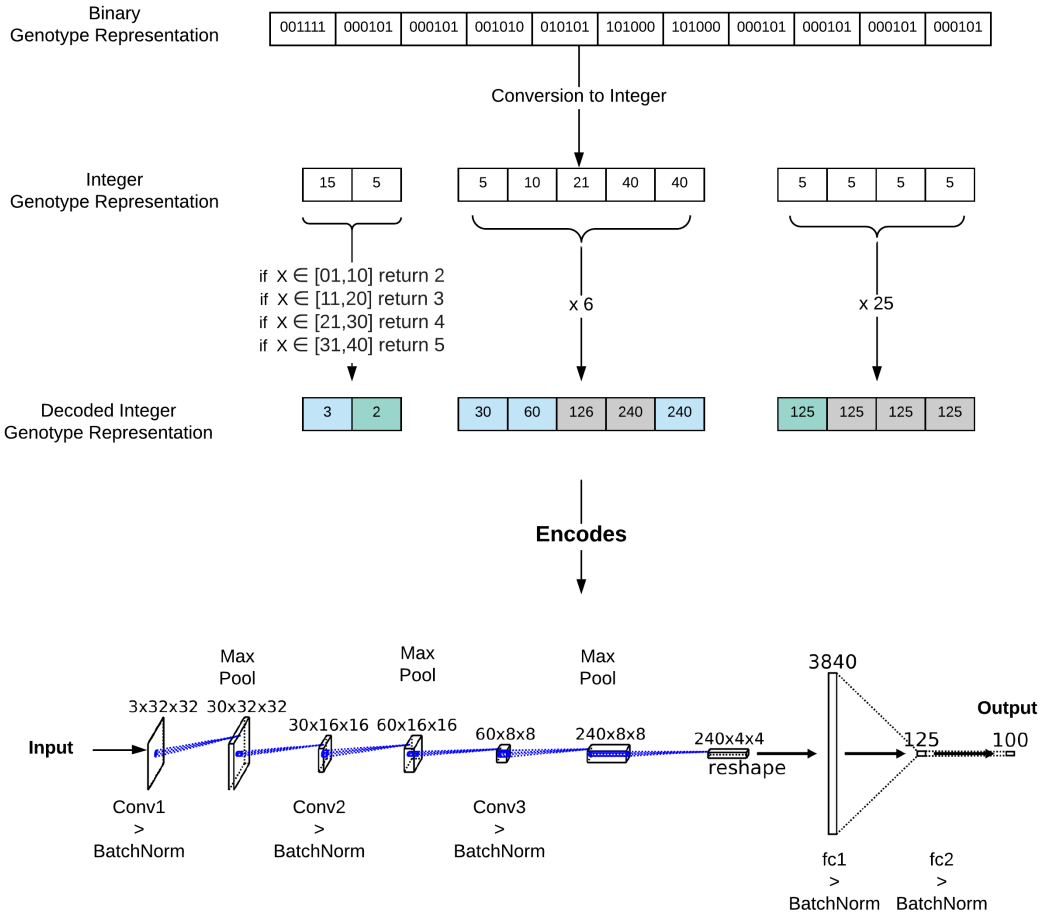

FIGURE 3.2: Decoding the genotype representation of the baseline FlexStudent

Figure (3.2) illustrates how our NEMOKD algorithm decodes the genotype representation of the baseline FlexStudent model used in subsequent experiments. This is the architecture from which all FlexStudent variations will evolve from. The Platypus library facilitates problems encoded with discrete integer values and automates the conversion from integer to binary and vice versa. For this particular problem, the values each gene can take lies in the interval $[1, 40]$. Since different genes have different functions, we have

added another layer of encoding that enforces different boundaries and intervals for different genes, based upon the value of the intermediate integer representation. Again, this is illustrated in Figure (3.2) and details of the intervals are provided in Table (3.1).

Crossover and mutation operators are applied to solutions at the binary level. Through this decoding process, resultant modifications to the genotype are realised in the model architecture. For example, if crossover or mutation resulted in a value of the Decoded Integer Genotype's 0th index changing from a three to four, than an extra convolutional layer will be added to the architecture when the genotype is passed to the PyTorch model object upon instantiation.

3.3.2 Resnet8x4 Solution Space

To evaluate this methodology with another type of DNN student architecture, we modified a Resnet8 $\times$ 4 architecture to enable an evolutionary process to modify four hyper-parameters that determine the number of output channels of the convolutional layers contained in each block. The values of these hyper-parameters can take belong to the set $\{8, 16, 24 \ldots, 512\}$. For this experiment, the number of layers remained fixed.

3.3.2.1 Resnet8x4 Genotype Encoding

With only four hyper-parameters subject to changes during evolution, the problem was encoded with a genotype of length four. Each gene corresponds with a block containing multiple layers and determines the number of output channels in the convolutional layer contained in the block. A simple illustation is provided in Figure (3.3).

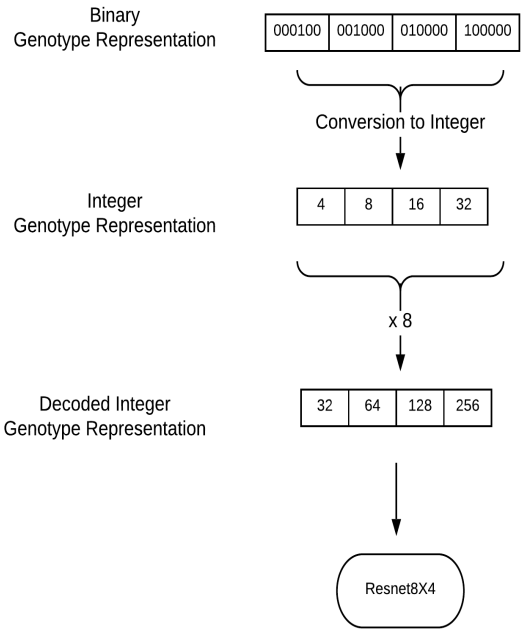

FIGURE 3.3: Decoding the genotype representation of the Baseline Resnet model.

3.4 Implementation Details

The algorithm workflow outlined in Section (3.2) was implement using Python, the PyTorch[2] deep learning framework, the Platypus[3] library for Multiobjective Optimization, OpenVINO[4] software with it's Python API and RepDistiller[5], a python program built for knowledge distillation research that was used to produce the following publication [25]. In this section we describe how each of these components were used to produce the algorithm illustrated in Figure (3.1). The algorithm spanned two computers with the majority of computation being performed on one of the ECR Cluster's[6] AMD compute nodes. The objective values were, however, measured on a local computer hosting the $NCS2$ implementation of the Myriad X VPU. The following three subsections are directly related to the three sections in Figure (3.1) and provided more specific implementation details.

[2] https://pytorch.org/
[3] https://platypus.readthedocs.io/en/latest/
[4] https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html
[5] https://github.com/HobbitLong/RepDistiller
[6] https://ecr-cluster.github.io/

3.4.1 Evolution

This section describes the implementation details of the "**Evolution**" section of Figure (3.1). This part of the pipeline employs the NSGAII algorithm provided by the Platypus library. This library implementation supports problems that require the genotype to be encoded with discrete integer values, making the task of encoding neural network hyper-parameters, such as the number of output channels or layers, relatively simple. Additionally, the library also provided a simple and intuitive method to specify an initial population. This enabled us to start the algorithm at a particular point in the solution space but also provided a layer of redundancy if the algorithm were to unexpectedly terminate. All the computation in this section of the pipeline is run on the AMD compute node's CPU.

3.4.2 Learning

This section describes the implementation details of the "**Learning**" section of Figure (3.1). This part of the pipeline has by far the highest computational cost. As discussed in Section (2.8), Knowledge Distillation is more computationally expensive than the standard DNN training procedure since data is passed through both the student and teacher model to calculate the loss value used for back propagation. As such, this part of the pipeline was conducted on the AMD compute nodes NVIDIA K20 GPU. The knowledge distillation implementation employed in this workflow was taken from the RepDistiller GitHub[7] repository.

3.4.3 Evaluation

Since the ERC cluster does not currently facilitate the use of a Myriad X VPU, the "**Evaluation**" section of the pipeline in Figure (3.1) had to be conducted on a local machine. This engineering challenge was overcome through the use of Python sockets and a server script that was run on a local machine hosting a Myriad X VPU and the Intel distribution of OpenVINO. Once a model has completed the "**Learning**" phase illustrated in Figure (3.1), the model was sent from the AMD compute node over a TCP socket connection established with a locally running server program that automates

[7] https://github.com/HobbitLong/RepDistiller

the evaluation procedure. This locally running server program was written in Python and utilises PyTorch, OpenVINO and some shell scripting. Upon receipt of a PyTorch model, the server program first converts the model to the ONNX[8] format using the inbuilt functionality provided by PyTorch. This ONNX model is then converted to an FP16 (half precision) IR format using the OpenVINO model optimiser. Using the OpenVINO Python API, the model is loaded onto the Myriad X VPU and testing is conducted with the relevant test dataset. Inference accuracy and latency are measured and returned to the main client program running on the ECR cluster over the same TCP socket connection. The connection is then terminated and the server program continues to listen for new connections.

[8] https://onnx.ai/

Chapter 4

Evaluation and Analysis

This chapter presents the empirical evaluation of the methodology presented in Chapter (3). To strengthen support for conclusions that may be drawn from this evaluation, we conducted three similar experiments that use different datasets and teacher-student combinations to determine whether the results generalize across datasets and models. Section (4.1) details the format that all these experiments follow. The subsequent three sections follow the naming conventions *Student-Teacher on Dataset* and present the results of the experiment conducted using these model and dataset combinations. The final section repeats the experiments conducted in Section (4.3) but extends the solution space to facilitate the optimisation of the knowledge distillation hyper-parameters in conjunction with the student architecture. The results of this extended experiment are then compared to another state-of-the-art distillation technique.

4.1 Experimental Setup

To determine the validity of the hypothesis proposed in Section (1.2) and explore our research aims from Section (1.4), we conducted three experiments that employ different student-teacher combinations. One experiment was conducted on the CIFAR10 dataset and the remaining two were conducted on the CIFAR100 dataset. Each experiment follows the same format and consists of three parts:

1. **NEMOKD**: We evolve student architectures using the NEMOKD methodology, training each model for just 30 epochs. In doing so, our algorithm profiles students

based upon their performance in early stages of training. Here, a baseline student architecture is used as a starting point in the evolutionary process. It is important to note that the knowledge distillation hyper-parameters are fixed. These are normally chosen depending on the size of the student model in relation to the teacher model as discussed in [38], the intuition behind this experiment is that the student will adapt their architecture to accommodate the selected hyper-parameters as the student model has the capacity to expand and contract. As such, reasonable, albeit arbitrary choices for these hyper-parameters were made. Since this methodology employs a multi objective evolutionary process with accuracy and latency as the two objective values, we aim to determine how effective this method is at automating the design of student architecture (Research Aim 1).

2. **Plain Knowledge Distillation**: For the same baseline student architecture used in part (1) and using the same teacher model, we iterate through different combinations of knowledge distillation hyper-parameters to identify the optimal hyper-parameters that yield the best student accuracy after 30 epochs. As the student architecture remains fixed, there is no scope to improve the latency of the baseline student model with this approach. These results will be used as a baseline for comparison to determine the extent to which the student architecture can adapt to accommodate fixed knowledge distillation hyper parameters (Research Aim 2).

3. **Extra Tuition**: Both (1) and (2) only perform knowledge distillation based training for 30 epochs. Here, student solutions are selected from the final generation of solutions produced by (1) and provided with further knowledge distillation based training. In doing so, we hope to determine whether evolving student models based upon their performance in the early stages of training is a viable means of reducing the amount of computational overhead (Research Aim 3) and most importantly, the validity of our hypothesis (Section 1.2). Where appropriate, results may also be compared with other state-of-the-art student-teacher training techniques.

For each experiment, the NEMOKD part was designed to run for 7 days with the setup described in Section (3.4), as this was the maximum time limit imposed on experiments running on the ERC cluster. As such, experimental parameters such as population size and number of generations had to be altered depending on the size of the teacher and student models to ensure a satisfactory experiment could be performed within this time

limit. In the sections that follow, Table (4.1), Table (4.3) and Table (4.5) provide the precise hyper-parameters selected for each experiment.

4.2 FlexStudent-MobileNetV2 on CIFAR10

This experiment was conducted on the CIFAR10 dataset and employs MobileNetV2 as the teacher model. Student models evolve from the baseline FlexStudent presented in Section (3.3.1.1). All relevant hyper-parameters for each part of the experiment are detailed in Table (4.1).

NEMOKD	
Teacher	MobileNetV2
Evolutionary Starting Point	Baseline FlexStudent
Dataset	CIFAR10
Epochs	30
Temperature	$\tau = 1.5$
Alpha	$\alpha = 0.1$
Population Size	20
Number of Generation	27
Plain Knowledge Distillation	
Teacher	MobileNetV2
Student	Baseline FlexStudent
Dataset	CIFAR10
Epochs	30
Temperature Range	$\tau \in [1, 1.5, 2, \ldots, 20]$
Alpha Range	$\alpha \in [0.1, 0.2, \ldots, 0.9]$
Extra Tuition	
Teacher	MobileNetV2
Students	6 $\times$ from Pareto Optimal Front
Dataset	CIFAR10
Epochs	120
Temperature	$\tau = 1.5$
Alpha	$\alpha = 0.1$

TABLE 4.1: Experiment Hyper-parameters.

4.2.1 NEMOKD

Figure (4.1) illustrates the population at two distinct generations of the evolutionary process, in addition to the baseline architecture from which all the students have evolved. As shown, the chosen knowledge distillation parameters for this experiment have greatly increased the accuracy of the baseline model. With 30 epochs of training, many students

in the final generation have evolved to attain a better accuracy than the baseline model but with the same or better latency. The same is also true of the baseline model trained with knowledge distillation. The most accurate students, however, have larger latency values with respect to the baseline model. This experiment produced a diverse range of student models and to illustrate this, we have included diagrams of the fastest, slowest and best trade-off students in Section (A.1). The best trade-off model evolved five convolutional layers with a relatively small number of output channels and just two fully connected layers with a low to moderate number of neurons.

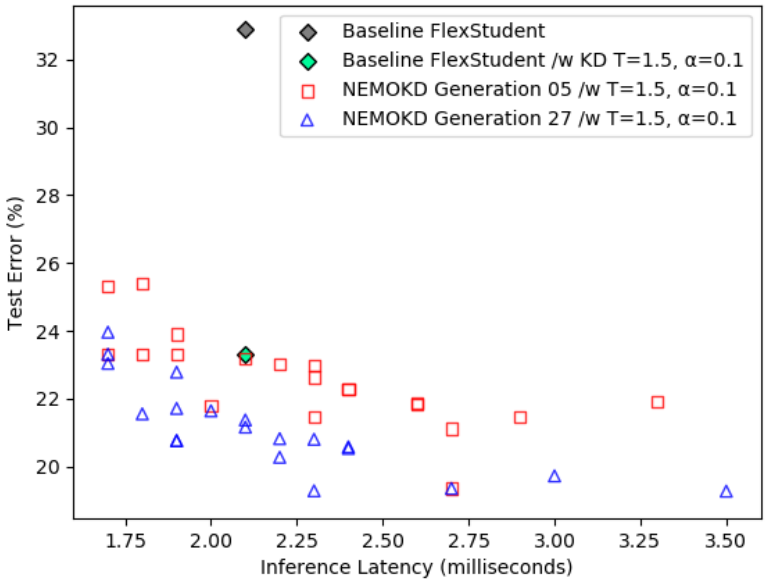

FIGURE 4.1: Student performance after 30 epochs for two distinct NEMOKD generations.

4.2.2 Knowledge Distillation Parameter Search

Figure (4.2) illustrates how different combinations of knowledge distillation parameters affect the accuracy of the baseline model after 30 epochs. The blue surface illustrates the error rate of the baseline model with respect to different combinations of knowledge distillation hyper-parameters. The orange plane indicates the baseline test error performance without knowledge distillation. On this dataset, any choice of KD hyper-parameters provides a significant increase in accuracy over the baseline model. As can be seen more clearly in Figure (4.3), we generally observe better accuracy with lower alpha values. In this case, no clear patter is observed with changing temperature as

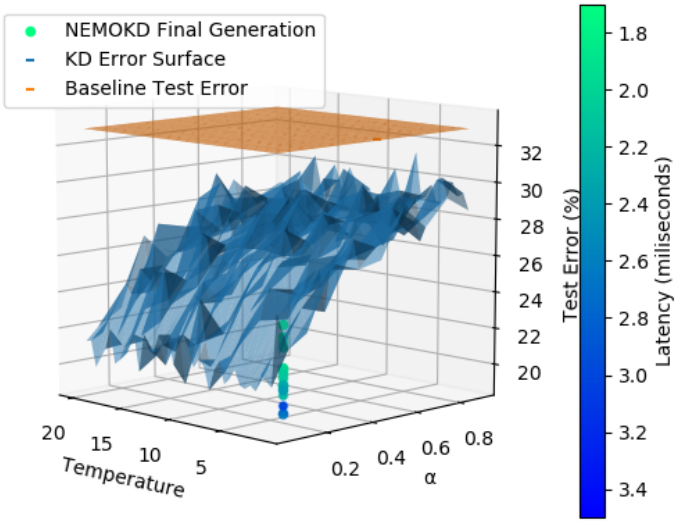

FIGURE 4.2: Knowledge Distillation Error Surface /w 30 epochs.

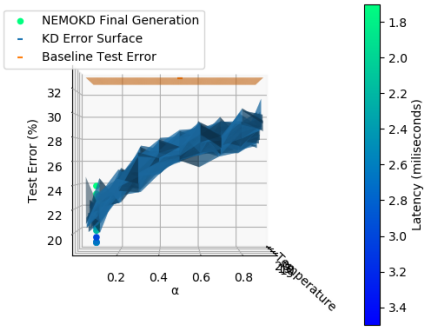

FIGURE 4.3: α vs Test Error

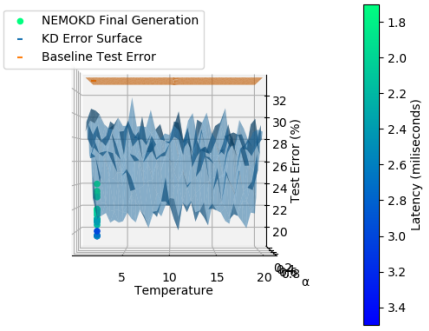

FIGURE 4.4: τ vs Test Error

shown in Figure (4.4). Plotting the final NEMOKD generation in the same 3D space, we observe that with 30 epochs of training, this method produces better accuracy than can be obtained by distilling knowledge into the baseline model. Though this increase in accuracy is obtained at the expense of latency.

4.2.3 Extra Tuition

As shown in Figure (4.5), a spread of six students were selected from the final generation, each of these were subject to a further 90 epochs of teacher assisted training as specified in Table (4.1) and subsequently plotted in the objective space. The assumption that

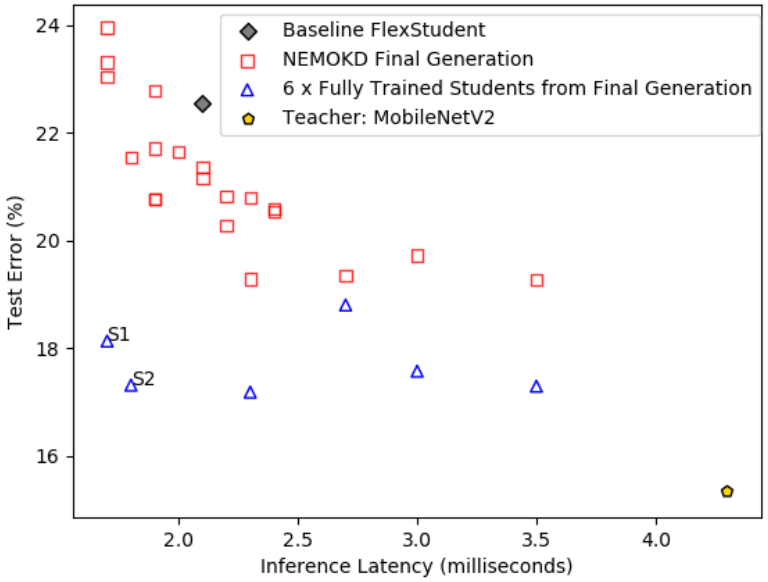

FIGURE 4.5: Final NEMOKD generation compared to a subset of fully trained students.

student performance after 30 epochs is indicative of student performance after further training has been confirmed for the majority of the selected students, albeit with a significant outlier. This is illustrated in Figure (4.5) where the approximate shape of the pareto optimal front is recognisable in the set of students that have received extra tuition. In this particular experiment, the accuracies that have been attained by the fully trained students are approaching that of the teacher model while maintaining a significantly lower latency. Most importantly, we observe that two of our selected models have both lower latency and higher accuracy which supports our hypothesis from Section (1.2). Table (4.2) details the specific metrics for these students along with the percentage change with respect to the baseline. Due to constraints on time and resources, we did not fully train the entire final generation. However, a large proportion of students from the final generation of student satisfy the hypothesis after just 30 epochs of training.

	Accuracy	Latency
Baseline	**77.46**	**2.1**
S1	81.87 (↑5.69%)	1.7 (↓19.05%)
S2	82.69 (↑6.75%)	1.8 (↓14.29%)

TABLE 4.2: Metrics for the fully trained student models from Figure (4.5) that support our hypothesis.

4.3 FlexStudent-Resnet32x4 on CIFAR100

This experiment was conducted on the more difficult CIFAR100 dataset and employs a larger teacher model: Resnet32x4. Student models evolve from the same baseline FlexStudent as per the previous experiment. All relevant hyper-parameters for each part of the experiment are detailed in Table (4.3).

NEMOKD	
Teacher	Resnet32 $\times$ 4
Evolutionary Starting Point	Baseline FlexStudent
Dataset	CIFAR100
Epochs	30
Temperature	$\tau = 4$
Alpha	$\alpha = 0.5$
Population Size	20
Number of Generation	14
Knowledge Distillation	
Teacher	Resnet32 $\times$ 4
Student	Baseline FlexStudent
Dataset	CIFAR100
Epochs	30
Temperature Range	$\tau \in [1, 1.5, 2, \ldots, 20]$
Alpha Range	$\alpha \in [0.1, 0.2, \ldots, 0.9]$
Extra Tuition	
Teacher	Resnet32 $\times$ 4
Students	6 $\times$ from Pareto Optimal Set
Dataset	CIFAR100
Epochs	240
Temperature	$\tau = 4$
Alpha	$\alpha = 0.5$

TABLE 4.3: Experiment Hyper-Parameters.

4.3.1 NEMOKD

Analogous to the previous experiment, Figure (4.6) illustrates the population at two distinct generations of the evolutionary process, in addition to the baseline architecture from which all the students have evolved. Interestingly, the combination of knowledge distillation hyper-parameters we chose for this experiment have had a negative impact on the accuracy of the baseline model. However, the evolved students appear to have adapted their architecture to accommodate these parameters, resulting in student models with significant accuracy improvements for the same inference latency. As observed in

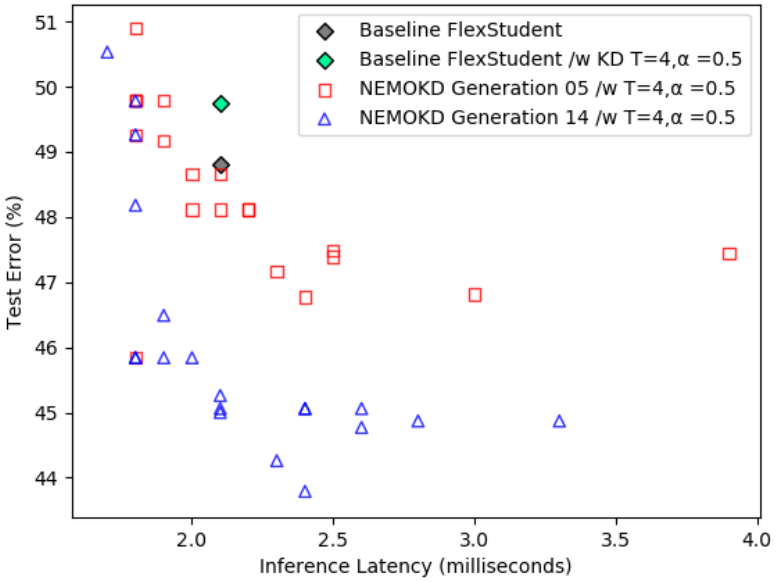

FIGURE 4.6: Student performance after 30 epochs for two distinct NEMOKD generations.

the previous experiment, many of these students have significantly better accuracy with the same or better latency with respect to the baseline model. The best accuracy produced by the NEMOKD algorithm, however, is obtained by an architecture with a higher latency. In contrast to the previous experiment, every student model in the final generation evolved to have the same layer structure as the baseline model. Similarly, we have provided diagrams of the fastest, slowest and best trade-off students in Section (A.2). While the layer structure remains much the same across the models, the number of output channels and neurons in the fully connected layers appear to play the most significant role in determining pareto optimal students in this experiment.

4.3.2 Knowledge Distillation Parameter Search

In contrast to the previous experiment, and as shown in Figure (4.7), some combinations of the KD parameters have a negative effect on the accuracy of the baseline model. This is not uncommon to see with more challenging datasets as discussed previously in Section (2.8.6). That being said, improvements are seen with the vast majority of combination. As can be seen more clearly in Figure (4.8), we generally observe better accuracy with $\alpha \in [0.1, 0.3]$. Similarly, as shown in Figure (4.9), we generally observe better accuracy for results when the temperature, $\tau \in [5, 20]$. Again, plotting the final NEMOKD

generation in the same 3D space, we observe that this method produces better accuracy than can be obtained by distilling knowledge into the baseline model, once again, at the expense of latency.

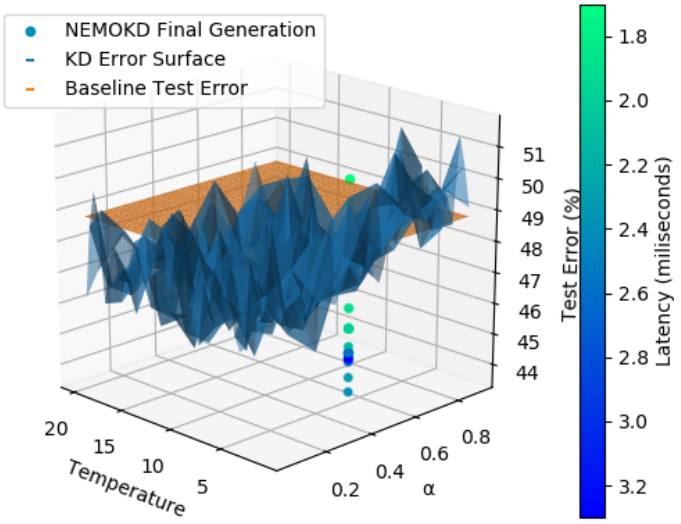

FIGURE 4.7: Knowledge Distillation Error Surface /w 30 epochs.

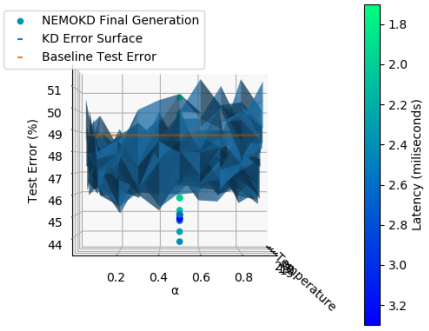

FIGURE 4.8: Error with respect to α

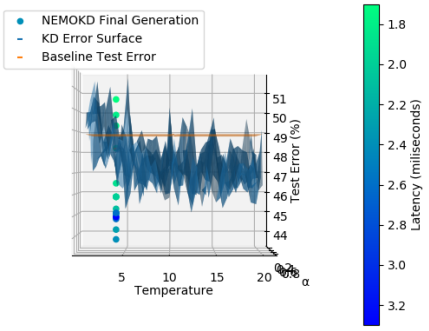

FIGURE 4.9: Error with respect to τ

4.3.3 Extra Tuition

Shown in Figure (4.10), a spread of six students were selected from the final generation, each of these were subject to a further 210 epochs of teacher assisted training as specified in Table (4.3) and subsequently plotted in the objective space. The assumption that student performance after 30 epochs is indicative of student performance after further

training has, once again, largely been confirmed. Here, the approximate shape of the pareto optimal front has been preserved, albeit with smaller y-axis separation. Similar to our previous experiment, two students that meet the requirements that satisfy our hypothesis have been produced by this method. The metrics associated with these students are compared to the baseline model in Table (4.4).

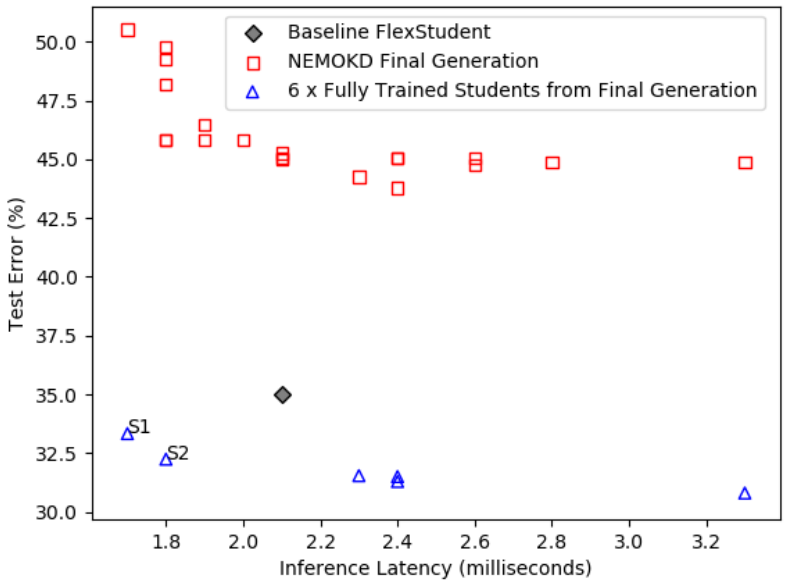

FIGURE 4.10: Final NEMOKD generation compared to a subset of fully trained final generation students.

	Accuracy	Latency
Baseline	**65.0**	**2.1**
S1	66.67 (↑2.56%)	1.7 (↓19.05%)
S2	67.76 (↑4.25%)	1.8 (↓14.29%)

TABLE 4.4: Metrics for the fully trained student models from Figure (4.10) that satisfy an increase in accuracy with a decrease in latency.

4.4 Resnet8x4 - Resnet32x4 on CIFAR100

This experiment was conducted on the CIFAR100 dataset and differs from the previous two experiments in that:

1. A different evolutionary starting point is used for the student. Namely, ResNet8×4.
2. The layers of our flexible ResNet model are fixed, only the output channels of the convolutional layers are modified in the evolutionary process.

The experiment hyper-parameters are provided in Table (4.5).

NEMOKD	
Teacher	ResNet32 × 4
Evolutionary Starting Point	ResNet8 × 4
Dataset	CIFAR100
Epochs	30
Temperature	$\tau = 4$
Alpha	$\alpha = 0.9$
Population Size	10
Number of Generation	13
Knowledge Distillation	
Teacher	ResNet32 × 4
Student	ResNet8 × 4
Dataset	CIFAR100
Epochs	30
Temperature Range	$\tau \in [1, 1.5, 2, \ldots, 20]$
Alpha Range	$\alpha \in [0.1, 0.2, \ldots, 0.9]$
Extra Tuition	
Teacher	ResNet32 × 4
Students	Pareto Optimal Set
Dataset	CIFAR100
Epochs	240
Temperature	$\tau = 4$
Alpha	$\alpha = 0.9$

TABLE 4.5: Experiment Hyper-paramters

4.4.1 NEMOKD

As with the previous experiments, Figure (4.11) plots the population for two distinct generations of the evolutionary process, in addition to the baseline architecture from which all the students have evolved. Here we see that for this baseline architecture,

the selected knowledge distillation hyper-parameters have had a slight negative impact on accuracy. With 30 epochs of training, a number of architectures, do however, attain a better accuracy with improved inference times. Similar to both the previous experiments, the highest accuracy is attained by models with higher latency that of the baseline model. Due to the size and complexity of the student models in this experiment, it was not practicable to provide illustrative diagrams. As previously mentioned, the layer structure of the student models in this experiment was fixed and only the number of channels in convolutional layers were subject to evolutionary changes.

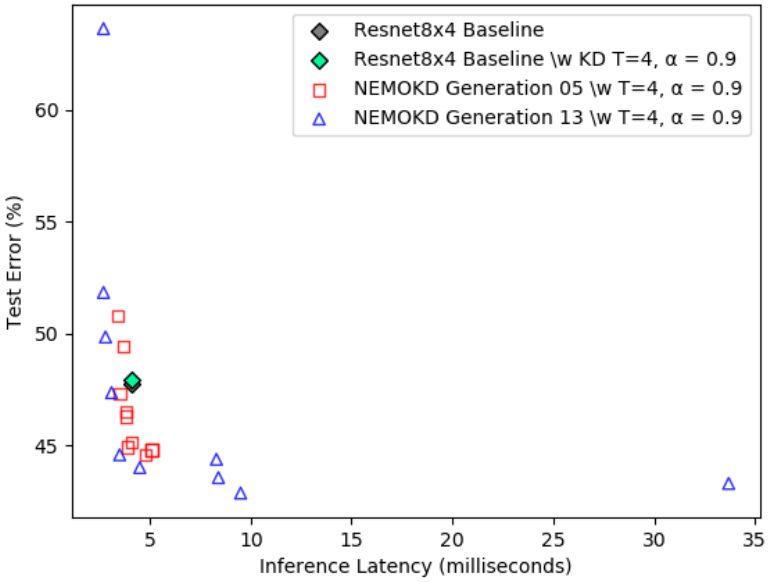

FIGURE 4.11: Student performance after 30 epochs for two distinct NEMOKD generations.

4.4.2 Knowledge Distillation Parameter Search

In this experiment, the majority of combinations of knowledge distillation hyper-parameters have a negative impact on the baseline model accuracy, though certain combinations do provide improvements as shown in Figure (4.12). In this case, no major trends are observed with respect to the individual hyper-parameters as shown in Figure (4.13) and Figure (4.14). By plotting the final NEMOKD generation in the same 3D space, we observe that this method, once again, produces better accuracy than can be obtained by distilling knowledge into the baseline model with 30 epochs of training. As seen in

the previous two experiments, students with the highest accuracy typically achieve this gain at the expense of latency.

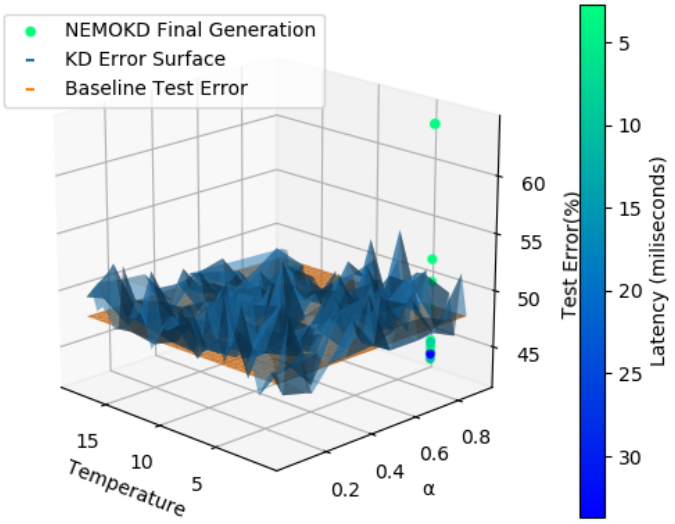

FIGURE 4.12: Knowledge Distillation Error Surface /w 30 epochs

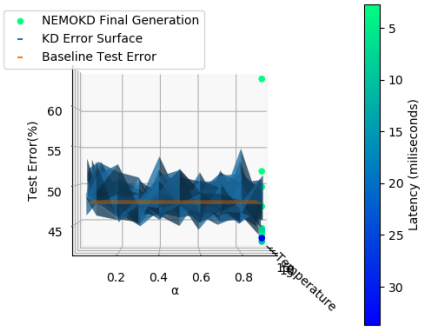

FIGURE 4.13: Error with respect to α

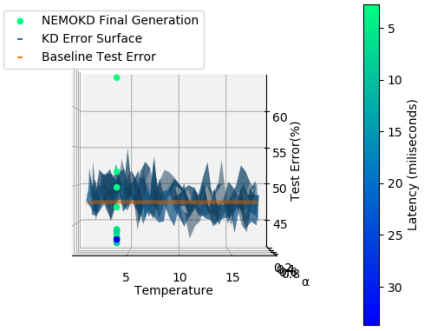

FIGURE 4.14: Error with respect to τ

4.4.3 Extra Tuition

As shown in Figure (4.15), the entire final generation were subject to a further 210 epochs of teacher assisted training as specified in Table (4.5) and subsequently plotted in the objective space. The assumption that student performance after 30 epochs is indicative of student performance after additional training, as been confirmed again in this experiment with the approximate shape of the pareto optimal front being preserved,

again with smaller y-axis separation. Figure (4.15) also compares our fully trained students with some recent state of the art teacher-student training methodologies and their respective benchmarks. The important observations are outline below:

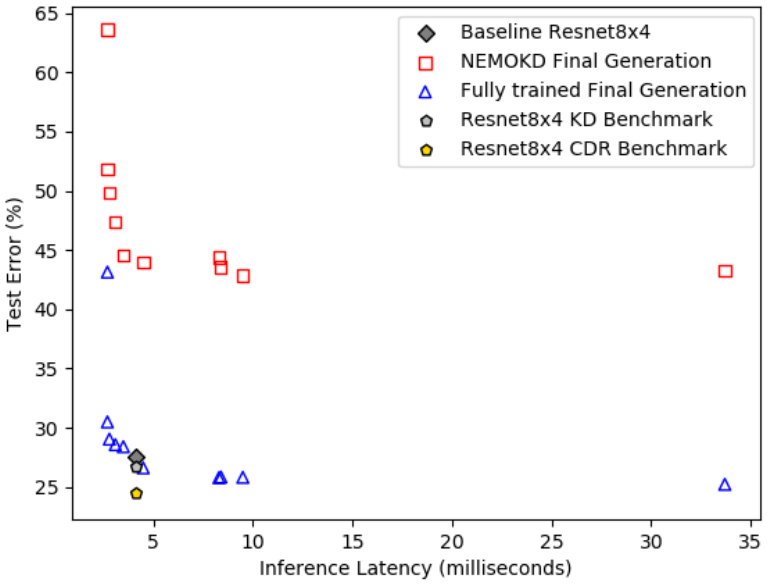

FIGURE 4.15: Final NEMOKD generation compared to the fully trained final generation.

- None of our fully trained students offer both a higher accuracy and lower latency compared to the fully trained baseline model. This could be for a number of reasons, but the most likely explanation is the prohibitive tendencies of knowledge distillation in the later stages of training. This argument is supported by the fact that a number of students exhibit higher accuracy and lower latency with respect to the baseline in early stages of training (30 epochs).

- Some of our evolved student models supersede the Resnet8 $\times$ 4 knowledge benchmark provided in Section (2.8.5) Table (2.2). These students, however, have done so at a significant latency increase.

- None of our students have attained comparable or better accuracy than the CRD benchmark, even though the latency of some are significantly larger than the baseline model. In this case, the CRD benchmark represents the best trade-off between accuracy and latency in the objective space.

- The most accurate student model has evolved to have double the inference latancy of the teacher but provides a poor trade off between accuracy and latency when compared with other student models.

4.5 Extended NEMOKD Experiment

This section presents an experiment almost identical to the one presented in Section (4.3) but with a few modifications. In particular, the solutions space was extended such that the knowledge distillation hyper-parameters and the student architecture are optimised simultaneously. The evolutionary starting point for these parameters were those selected in the NEMOKD part of Section (4.3). Additionally, we reduced the population size by half and doubled the generations.

Interestingly, many of the solutions have evolved knowledge distillation hyper-parameters that are similar to the combination that yields the highest accuracy for the baseline model ($\alpha = 0.2$, $\tau = 12.5$). This behavior is more pronounced with temperature as can be seen in Figure (4.17). As shown in Figure (4.16) alpha values typically remain around the evolutionary starting point of $\alpha = 0.5$.

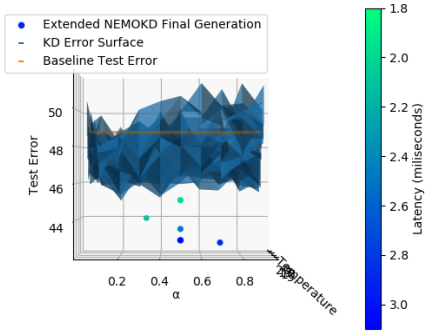

FIGURE 4.16: Error with respect to α

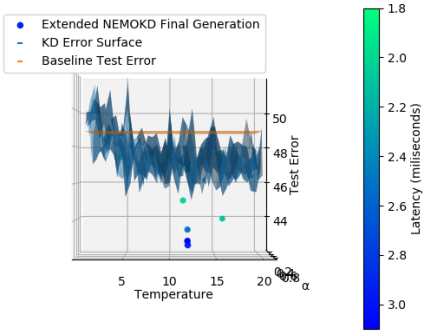

FIGURE 4.17: Error with respect to τ

Figure (4.18) demonstrates the superiority of this extended NEMOKD approach after students have undergone a full training regime. Here, the vast majority of students trained with evolved KD parameters exhibit a better trade-off between accuracy and latency in the objective space. Moreover, benchmarks for a MobileNetV2 student with a Resnet50 teacher from Table (2.2) have been plotted to provide a comparison. Here,

we see that the state-of-the art CRD method significantly improves the accuracy of MobileNetV2 over both normal training methods and plain knowledge distillation. Since our baseline model is more accurate than the MobileNetV2 baseline, a direct comparison between our highest scoring model and the CRD MobileNetV2 benchmark is, perhaps, not the most appropriate way to compare these two distillation methods. Instead, we select a number of pareto optimal solutions and compare them with the CRD benchmark by calculating the percentage change of the two objective metrics with respect to each corresponding baseline. It should be noted that CRD does not modify the student architecture and therefore the latency remains fixed. This comparison is illustrated in Figure (4.19) with negative changes representing a percentage decrease and positive changes representing a percentage increase. As the teacher-student combinations differ, we have also included the average percentage increase CRD produces across all teacher-student combinations in Table (2.2). As shown, CRD produced an 6.71% studnet accuracy increase on average but does not alter the latency. Similarly, the comparable MobileNetV2 example attains a slightly higher 6.98%. Interestingly, our method can produce a smaller but comparable percentage increase in accuracy compared with the baseline but with a simultaneous reduction in latency. As we move from the leftmost student to right, we are able to visualise how a small increase in accuracy becomes increasingly expensive in terms of latency. The student labeled $S4$ exhibits a marginally smaller accuracy increase than is observed with CRD on average, but similar to CRD no change in latency is observed. $S5$, however, achieves the largest accuracy increase over all the student models and both CRD examples, but does so at a significant cost to latency. Precise numeric values for the metrics relating the the MobileNetV2 CRD benchmark and our NEMOKD students are presented in Table (4.6) an Table (4.7) respectively.

	Accuracy	Latency
Baseline	**64.60**	**4.3**
KD	67.35 (↑4.26%)	4.3
CRD	69.11 (↑6.98%)	4.3

TABLE 4.6: MobileNetV2-ResNet50 benchmarks from Table (2.2).

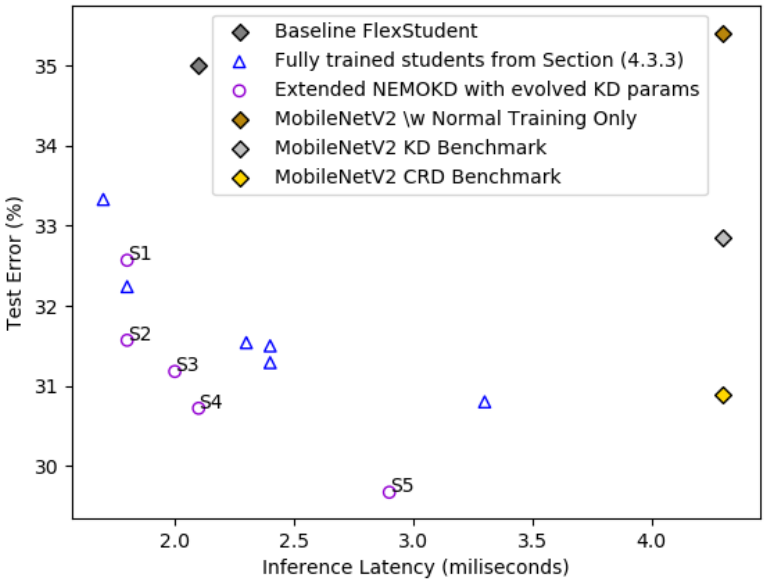

FIGURE 4.18: Fully trained students from the final generation of: (1) NEMOKD with fixed KD parameters and (2) NEMOKD with evolved KD parameters. Additionally, we have the MobleNetV2-ResNet50 CRD benchmarks from Table (2.2) are provided for comparison.

	Accuracy	Latency
Baseline	**65.00**	**2.1**
S1	67.43 (↑3.73%)	1.8 (↓14.29%)
S2	68.43 (↑5.28%)	1.8 (↓14.29%)
S3	68.82 (↑5.88%)	2.0 (↓04.76%)
S4	69.82 (↑6.58%)	2.1 (00.00%)
S5	70.33 (↑8.20%)	2.9 (↑38.10%)

TABLE 4.7: Metrics for the labeled student models presented in Figure (4.18).

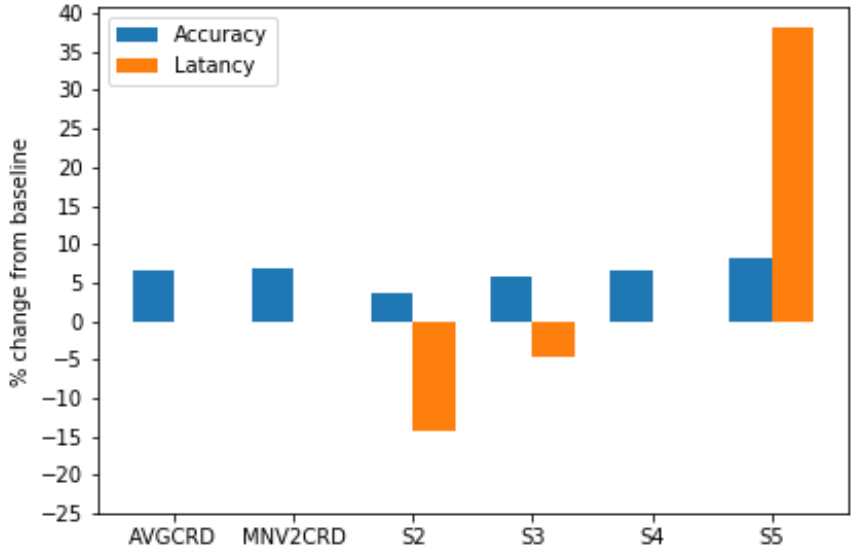

FIGURE 4.19: From left to right: AVGCRD - the average percentage increase in accuracy provided by CRD across all examples in Table (2.2), MNV2CRD the MobleNetV2-ResNet50 CRD benchmark from Table (2.2), $S2 \ldots S5$ - respective student models from Figure (4.18).

Chapter 5

Conclusion

To conclude this thesis, we summarise our key findings (Section 5.1) and discuss the main limitations associated with our proposed methodology (Section 5.2). Finally, in Section (5.3), we suggest some avenues of research that should be considered for future works.

5.1 Project Summary

The main aim of this research project was to develop an automated method to produce performance and accuracy enhanced knowledge distilled student models for deployment on the Myriad X VPU. We hypothesised that our proposed methodology could produce better performing student models in terms of both accuracy and latency compared to a baseline model. Results obtained from an empirical evaluation spanning two datasets and three distinct student-teacher combinations provided some support for this hypothesis, with two out of three experiments producing student models satisfying the hypothesis. Furthermore, comparing the results of our extended methodology with another state-of-the-art distillation technique that, on average, provides a 6.71% accuracy boost, we were able to show that our method is capable of providing a lower but comparable student accuracy increase with a simultaneous reduction in latency (5.28% increase in accuracy with a 14.29% latency decrease & 5.88% increase in accuracy with

a 4.76% latency decrease). Additionally, our method was able to provide a higher accuracy increase but at the expense of latency (8.20 % increase in accuracy but also with a detrimental 38% increase in latency).

As we saw from the exhaustive search results of knowledge distillation hyper-parameters, it is very difficult to predict the optimal combination. Some trends were observed and appeared to be more prominent with datasets containing fewer classes. Unfortunately, such trends do not appear in other more complex problems. Across all 30 epoch experiments, our method consistently provided higher accuracy students than could be obtained through an exhaustive KD parameter search with the baseline model, irrespective of the choice of KD hyper-parameters selected in our method. This highlights the importance of the student's architecture in the knowledge distillation process. Evolving the students appeared to enable the model to adapt and accommodate an arbitrary choice of knowledge distillation hyper-parameters, even if the choice was initially detrimental to the accuracy of the baseline model. However, our best result were obtained when we evolved the knowledge distillation hyper-parameters in conjunction with the student architecture.

In each of our experiments, the decision to profile student models based upon their performance in early stages was largely successful and significantly reduced the amount of computation required to execute our algorithm. The vast majority of student selected from the final generation for further training retained their accuracy position relative to the other students.

5.2 Limitations

While our methodology proved to be successful in producing enhanced student models in two of our experiments, it should be noted that typical GA experiments perform multiple runs of the same experiment to account for the stochastic nature of these algorithms. In two of our experiments, a single run was sufficient to achieve the desired result. However, results may differ if these experiment were to be repeated multiple times. Since each experiment took a week to complete on the hardware available to us, running the same experiments multiple times was not practical.

Another related criticism from a GA standpoint is the relatively small population size and number of generations employed in our experiments. Running these experiments with larger population sizes for more generations would likely provide better results. However, with the time and resources available to us, increasing these hyper-parameters was not feasible. With the use of a single GPU, the learning phase was conducted sequentially for each member of the population at each generation. Since the learning and evaluation of each individual is independent of the rest of the population, the entire population could be trained and evaluated in parallel if one had access to a large enough number of GPUs. This would considerably reduce the algorithm execution time and would facilitate experiments with a larger population and number of generations.

The solution space of our experiments allowed a modest number of hyper-parameters to be modified during the evolutionary process. Encoding the problem to allow more architectural hyper-parameters to be optimised could potentially provide even better results. Doing this would, however, increase the size of the solution space which may increase the number of generations required to find acceptable solutions. Additionally, upper and lower bounds enforced in the solution space may need to be tailored for specific problems.

The datasets chosen for this research both consist of relatively simple images and do not represent the complexities of current real world applications. As previously discussed in Section (2.8.6), knowledge distillation has performed poorly on complex datasets such as ImageNet[1]. As such, the success we have seen in our experiments cannot be guaranteed when applying this methodology to more complex problems. Additionally, we have tested this method on a very specific hardware platform and it should be noted that latency improvements as a result of changes to the student architectures will not necessarily provide better inference times on other hardware platforms.

The comparison between our approach and the state-of-the-art CRD method we presented in Section (4.5) is somewhat limited. In particular, we have only compared one result using our methodology to the average of many CRD result. As such, a more robust comparison would require an average to be taken across a range of further extended NEMOKD experiments.

[1] http://www.image-net.org/

5.3 Future Works

More work is required to establish whether our method can be employed as a useful compression technique for real world computer vision applications. As such, testing this method with more complex datasets such as ImageNet should be the first consideration for future works. Additionally, enabling more parameters to be subject to modification throughout the evolutionary process could prove to be beneficial.

More recent teacher-student methods have been shown [25] to significantly outperform knowledge distillation in a wide range of problems. Designing a flexible student model that accommodates both evolution and more complex distillation methods would be considerably more challenging, but given the success we have observed with knowledge distillation, is likely to be another worthwhile pursuit for future works.

The methods presented in this project have only been tested on one specific target hardware platform but could easily be adapted to optimise knowledge distillation for deployment on a wide range of devices. Further work could also include extending this method to other platforms and comparing the performance on different devices.

Appendix A

Evolved FlexStudent Architectures

A.1 FlexStudent-MobileNetV2 on CIFAR10

A.1.1 Lowest Latency

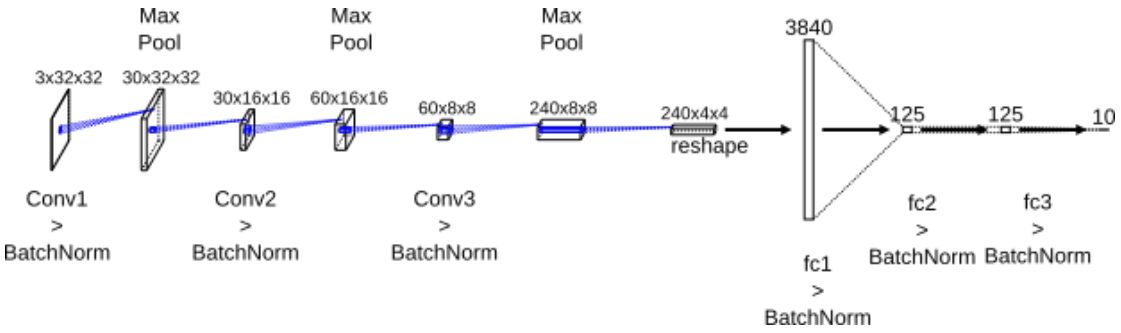

FIGURE A.1: Lowest Latency Student

A.1.2 Best Latency-Accuracy Trade-Off

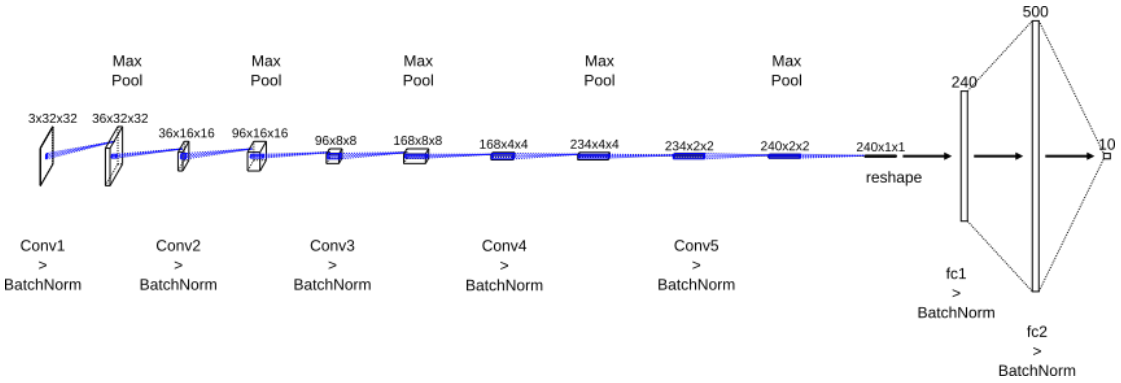

FIGURE A.2: Best Trade-Off Student

A.1.3 Highest Latency

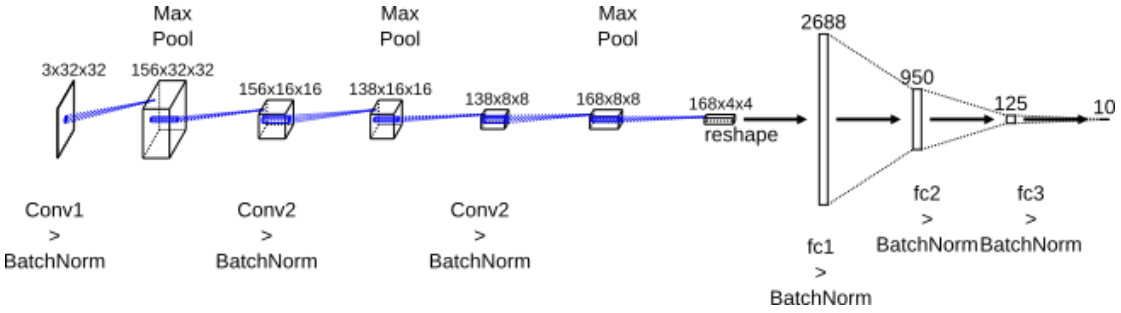

FIGURE A.3: Highest Latency Student

A.2 FlexStudent-ResNet32x4 on CIFAR100

A.2.1 Lowest Latency

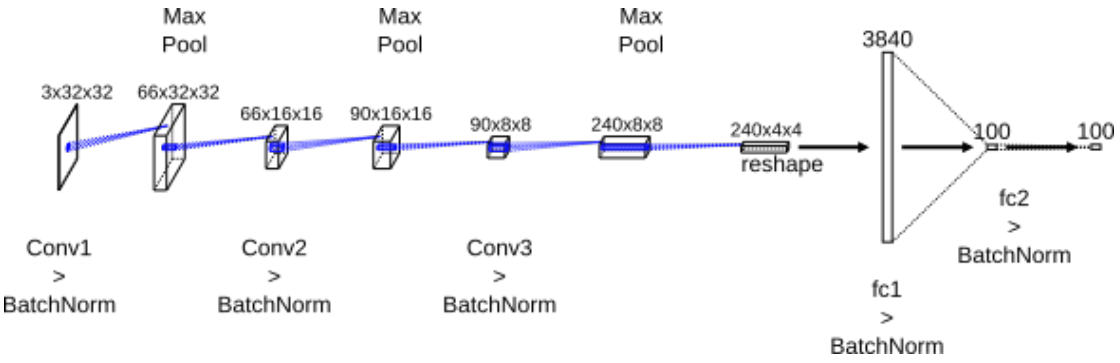

FIGURE A.4: Lowest Latency Student

A.2.2 Best Latency-Accuracy Trade-Off

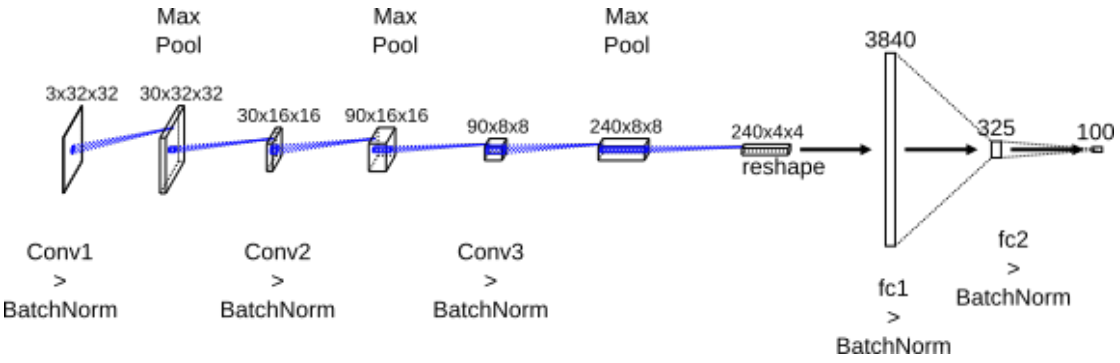

FIGURE A.5: Best Trade-Off Student

A.2.3 Highest Latency

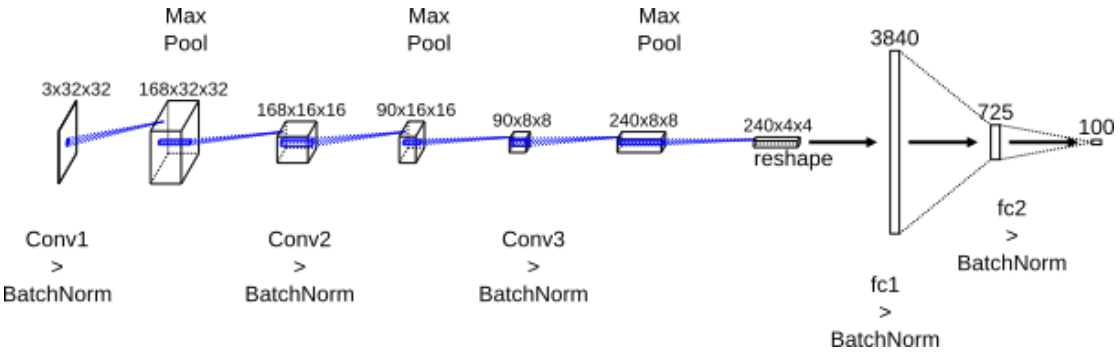

FIGURE A.6: Highest Latency Student

Bibliography

[1] Y.-H. Kim, B. Reddy, S. Yun, and C. Seo, "Nemo: Neuro-evolution with multi-objective optimization of deep neural network for speed and accuracy," in *JMLR: Workshop and Conference Proceedings*, vol. 1, pp. 1–8, 2017.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[3] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei, and M. Sun, "Dpp-net: Device-aware progressive search for pareto-optimal neural architectures," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 517–531, 2018.

[4] V. Radu, K. Kaszyk, Y. Wen, J. Turner, J. Cano, E. J. Crowley, B. Franke, A. Storkey, and M. O'Boyle, "Performance aware convolutional neural network channel pruning for embedded gpus," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 24–34, IEEE, 2019.

[5] J. Turner, E. J. Crowley, V. Radu, J. Cano, A. Storkey, and M. O'Boyle, "Distilling with performance enhanced students," *arXiv preprint arXiv:1810.10460*, 2018.

[6] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[7] K. Fukushima, "Neocognitron: A hierarchical neural network capable of visual pattern recognition," *Neural networks*, vol. 1, no. 2, pp. 119–130, 1988.

[8] 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, 1962.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, "Self-training with noisy student improves imagenet classification," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

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

[13] Q.-Y. Gu and I. Ishii, "Review of some advances and applications in real-time high-speed vision: Our views and experiences," *International Journal of Automation and Computing*, vol. 13, no. 4, pp. 305–318, 2016.

[14] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, *et al.*, "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[15] Intel, "Intel Myriad X VPU Product Brief." Accessed on: June, 23. 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/movidius-vpu/myriad-x-product-brief.html.

[16] William G. Wong, "Intel's Myriad X Vision Chip Incorporates Neural Network." Accessed on: June, 23. 2020. [Online]. Available: https://www.electronicdesign.com/industrial-automation/article/21805511/intels-myriad-x-vision-chip-incorporates-neural-network.

[17] Smith, N, "Transitioning from Intel® Movidius™ Neural Compute SDK to Intel® Distribution of OpenVINO™ toolkit." Accessed on: March, 17. 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/transitioning-from-intel-movidius-neural-compute-sdk-to-openvino-toolkit.html.

[18] 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*, pp. 49–55, 2019.

[19] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore ipu architecture via microbenchmarking," *arXiv preprint arXiv:1912.03413*, 2019.

[20] Coral, "USB Accelerator datasheet." Accessed on: Match, 28. 2020. [Online]. Available: https://coral.ai/docs/accelerator/datasheet/.

[21] A. Koul, S. Ganju, and M. Kasam, *Practical Deep Learning for Cloud, Mobile and Edge: Real-World AI and Computer Vision Projects Using Python, Keras and TensorFlow*. O'Reilly Media, Incorporated, 2019.

[22] E. BUBER and B. DIRI, "Performance analysis and cpu vs gpu comparison for deep learning," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pp. 1–6, Oct 2018.

[23] G.-B. D. L. Inference, "A performance and power analysis," *NVIDIA Whitepaper, Nov*, 2015.

[24] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for dnn compression research," *arXiv preprint arXiv:1910.12232*, 2019.

[25] Y. Tian, D. Krishnan, and P. Isola, "Contrastive representation distillation," in *International Conference on Learning Representations*, 2020.

[26] Intel, "OpenVINO™ toolkit Documentation." Accessed on: March, 8. 2020. [Online]. Available: https://docs.openvinotoolkit.org/latest/index.html.

[27] V. Kustikova, E. Vasiliev, A. Khvatov, P. Kumbrasiev, I. Vikhrev, K. Utkin, A. Dudchenko, and G. Gladilov, "Intel distribution of openvino toolkit: A case study of semantic segmentation," in *International Conference on Analysis of Images, Social Networks and Texts*, pp. 11–23, Springer, 2019.

[28] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where

we've been, where we're going," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–39, 2019.

[29] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *CoRR*, 2018.

[30] Intel AI Lab: Nervana Systems, "Neural Network Distiller." Accessed on: April, 16. 2020. [Online]. Available: https://nervanasystems.github.io/distiller/index.html.

[31] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[32] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances in neural information processing systems*, pp. 1379–1387, 2016.

[33] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[34] R. Abbasi-Asl and B. Yu, "Structural compression of convolutional neural networks based on greedy filter pruning," *arXiv preprint arXiv:1705.07356*, 2017.

[35] L. Theis, I. Korshunova, A. Tejani, and F. Huszár, "Faster gaze prediction with dense networks and fisher pruning," *arXiv preprint arXiv:1801.05787*, 2018.

[36] E. J. Crowley, J. Turner, A. Storkey, and M. O'Boyle, "Pruning neural networks: is it time to nip it in the bud?," 2018.

[37] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 535–541, 2006.

[38] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[39] Abinesh B, "Knowledge Distillation in Deep Learning." Accessed on: March, 15. 2020. [Online]. Available: https://medium.com/analytics-vidhya/knowledge-distillation-dark-knowledge-of-neural-network-9c1dfb418e6a.

[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org, **p129**.

[41] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," *arXiv preprint arXiv:1612.03928*, 2016.

[42] J. H. Cho and B. Hariharan, "On the efficacy of knowledge distillation," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4794–4802, 2019.

[43] S. Mirjalili, "Evolutionary algorithms and neural networks," *Studies in Computational Intelligence*, 2019.

[44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[45] Michael Lones, "Multiobjective Evolutionary Algorithms." Accessed on: June, 3. 2020. [PowerPoint slides]. Available: https://heriotwatt-my.sharepoint.com/:b:/g/personal/ml355_hw_ac_uk/EXAa1OKrUsNBqoGUJDwwlTgBTfyB86yrjfc_GoWq6PMYWA?e=HtLt4f.

[46] S. Luke, *Essentials of Metaheuristics*. Lulu, second ed., 2013. Accessed on: June, 3. 2020. [Online]. Available: https://cs.gmu.edu/~sean/book/metaheuristics/, **p141**.

[47] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[48] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2752–2761, 2018.