# Efficient Deep Neural Network Inference for Embedded Systems: A Mixture of Experts Approach

**Benjamin David Taylor**

*BSc. (Hons), Computer Science, Lancaster University, UK, 2015*

This dissertation is submitted for the degree of

*Doctor of Philosophy*


School of Computing and Communications

Lancaster University, UK

November 2020

# Efficient Deep Neural Network Inference for Embedded Systems: A Mixture of Experts Approach

*Benjamin David Taylor*

## Abstract

Deep neural networks (DNNs) have become one of the dominant machine learning approaches in recent years for many application domains. Unfortunately, DNNs are not well suited to addressing the challenges of embedded systems, where on-device inference on battery-powered, resource-constrained devices is often infeasible due to prohibitively long inferencing time and resource requirements. Furthermore, offloading computation into the cloud is often infeasible due to a lack of connectivity, high latency, or privacy concerns. While compression algorithms often succeed in reducing inferencing times, they come at the cost of reduced accuracy.

The key insight here is that multiple DNNs, of varying runtimes and prediction capabilities, are capable of correctly making a prediction on the same input. By choosing the fastest capable DNN for each input, the average runtime can be reduced. Furthermore, the fastest capable DNN changes depending on the evaluation criterion.

This thesis presents a new, alternative approach to enable efficient execution of DNN inference on embedded devices; the aim is to reduce average DNN inferencing times without a loss in accuracy. Central to the approach is a Model Selector, which dynamically determines which DNN to use for a given input, by considering the desired evaluation metric and inference time. It employs statistical machine learning to develop a low-cost predictive model to quickly select a DNN to use for a given input and the optimisation constraint. First, the approach is shown to work effectively with off-the-self pre-trained DNNs. The approach is then extended by combining typical DNN pruning techniques with statistical machine learning in order to create a set of specialised DNNs designed specifically for use with a Model Selector.

Two typical DNN application domains are used during evaluation: image classification and machine translation. Evaluation is reported on a NVIDIA Jetson TX2 embedded deep learning platform, and a range of influential DNN models including convolutional and recurrent neural networks are considered. In the first instance, utilising off-the-shelf pre-trained DNNs, a 44.45% reduction in inference time with a 7.52% improvement in accuracy, over the most-capable single DNN model, is achieved for image classification. For machine translation, inference time is reduced by 25.37% over the most-capable model with little impact on the quality of the translation. Further evaluation utilising specialised DNNs did not yield an accurate premodel and produced poor results; however analysis of a perfect premodel shows the potential for faster inference times, and reduced resource requirements over utilising off-the-shelf DNNs.

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university.

Benjamin David Taylor
November 2020

# Acknowledgements

This thesis would not have been possible without the help and support of so many people. I genuinely don't think I could have gotten through my PhD without the support and encouragement of those around me, both academically and otherwise. I will thank some of them next, and no doubt apologise profusely to anyone I forget.

First, I want to thank my supervisors: Dr. Zheng Wang, and Dr. Barry Porter, the time and effort you have expended on me is invaluable. I will be forever grateful for your advice, discussions, and crucial feedback; it has been essential in my success, not only throughout my PhD journey, but as a researcher as well.

I want to thank all my colleagues in SCC, it is easy to feel isolated when working alone so often, but you have made this journey less lonely and more enjoyable. Thank you to Willy and Andrew, with whom I had the pleasure of sharing an office, our endless talks and complaints on those difficult days when everything goes wrong really helped me through. Thank you Dr. Vicent Sanz Marco for all the hours we have spent working together and attending conferences, you always made an effort to talk about topics other than our work. Thank you to Dr. Charalampos Rotsos, Dr. Angelos Marnerides, and Dr. Utz Roedig, it has been a pleasure to TA for you over the last few years; I don't think I'm ever going to forget the course material for SCC 150. Thank you to all of the PGR rep team for striving to make the PhD journey a more social experience.

A huge thanks goes to my immediate family for their unconditional support throughout whatever I choose to do. I'd also like to thank so many friends that have kept me sane over the years. Kerry, I will miss our weekly lunches. Kat and Rahel, we don't talk often but when we do I always feel better afterwards - I'm blaming you for my addiction to board games. Jed, thank you for always having an ear for me when I needed it, and all the concerts and hikes we've done together. Ben, thank you for the many nights of chatting and playing games together, they were just what I needed after a long day of work. I'd like to thank "The Engineers": Andy, Michal, Imogen, Adam, Joel, and the ever mysterious Deepak; I hope our yearly traditions never stop. Anne, Joe, Freya, Ryan, and Sophie, thank you for being there whenever I needed you. I would like to thank everyone I get to play Korfball with, you are all such wonderful people.

Finally, I would like to thank Lancaster University for providing me with an FST PhD studentship to fund my PhD.

# Publications

## Contributing Publications

- Optimizing deep learning inference on embedded systems through adaptive model selection. Marco, V. S., **Taylor, B.**, Wang, Z., and Elkhatib, Y. *ACM Transactions on Embedded Computing Systems (TECS), 19(1):1–28. (2019)*

- Adaptive deep learning model selection on embedded systems. **Taylor, B.**, Marco, V. S., Wolff, W., Elkhatib, Y., and Wang, Z. *In Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, pages 31–43. ACM. (2018*)

## Additional Publications

- Improving spark application throughput via memory aware task co-location: a mixture of experts approach. Marco, V. S., **Taylor, B**., Porter, B., and Wang, Z. *In Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, pages 95–108. ACM.(2017)*

- Adaptive optimization for OpenCL programs on embedded heterogeneous systems. **Taylor, B.**, Marco, V. S., and Wang, Z. *In Proceedings of the 18th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, pages 11–20. ACM. (2017)*

- Cracking Android pattern lock in five attempts. Ye, G., Tang, Z., Fang, D., Chen, X., Kim, K. I., **Taylor, B.**, and Wang, Z. *In Proceedings of the 2017 Network and Distributed System Security Symposium (2017)*

# Table of contents

# List of figures

# List of tables

# Nomenclature

**Glossary**

| | |
|---|---|
| *BLEU* | A standard machine translation scoring metric. |
| *F1-score* | A standard machine translation and image classification scoring metric. Calculated from ROUGE and BLEU, or Precision and Recall, for machine translation or image classification, respectively. |
| *Inference* | The action of a deep neural network making a prediction on an input. |
| *oracle* | A representation of a perfect machine learning model. That is, a model able to achieve 100% accuracy. |
| *precision* | A standard image classification scoring metric. |
| *premodel* | A statistical machine learning model created to make a prediction before a Deep Neural Network is used. |
| *recall* | A standard image classification scoring metric. |
| *ROUGE* | A standard machine translation scoring metric. |
| *top-1* | A standard image classification scoring metric. |
| *top-5* | A standard image classification scoring metric; it is more lenient than top-1. |

**Acronyms / Abbreviations**

| | |
|---|---|
| `CNN` | Convolutional Neural Network |
| `DL` | Deep Learning |
| `DNN` | Deep Neural Network |
| `DT` | Decison Tree (statistical machine learning model) |
| `GRU` | Gated Recurrent Unit |
| `K-means` | K-Means (statistical machine learning model) |
| `KNN` | K-Nearest Neighbours (statistical machine learning model) |
| `LSTM` | Long-Short Term Memory |
| `ML` | Machine Learning |
| `MLP` | Multilayer Perceptron |

| | |
|---|---|
| NB | Naive Bayes (statistical machine learning model) |
| NMT | Neural Machine Translation |
| NN | Neural Network |
| RL | Reinforced Learning |
| RNN | Recurrent Neural Network |
| SML | Statistical Machine Learning |
| SVM | Support Vector Machine (statistical machine learning model) |

# Chapter 1

# Introduction

## 1.1 Overview

Deep learning is currently one of the key current research areas in machine learning. Deep Neural Networks (DNN) have proven their ability in solving many difficult and complex problems, such as: object recognition [24, 43], detecting and recognising objects in an image; facial recognition [102, 123], recognising people from just an image containing their face; speech processing [2, 154], decoding spoken word into text; and machine translation [4, 85, 122], translating text from one language to another. As well as being operated on higher-power servers, many of these deep learning applications are also important application domains for embedded systems [68], especially for sensing and mission critical applications such as health care and video surveillance. Once deep learning technologies become ubiquitous in embedded systems, even greater benefits will be revealed, such as automatous driving, affordable robots for home, augmented reality, and more intelligent personal assistance on mobile phone.

Unfortunately, existing deep learning solutions, which utilise DNNs, are not well suited to addressing the challenges of embedded systems; they are often resource hungry tasks, demanding a considerable amount of CPU, GPU, memory, and power in order to run effectively [9]. Furthermore, deep learning solutions are becoming more complex in an effort to improve their effectiveness [143]. It is often detrimental, or infeasible, for embedded systems to offer such a large number of system resources to a single task. Without optimization, the hoped-for advances in embedded capabilities will not arrive. The disparity between the resources required and those available will lead to huge energy consumption, reducing battery life, and long inferencing times, making real-time applications infeasible on battery-powered, resource-limited embedded devices.

Numerous optimisation tactics have been proposed to enable DNN inference on embedded devices, here DNN inference refers to the process of a DNN making a prediction on some input. Prior approaches are either architecture specific [119] or come with

drawbacks, such as a reduction in accuracy or an increase in inference time. A common technique used to accelerate DNN models on embedded devices is a process known as DNN compression, able to reduce resource and computational requirements of a given model [32, 35, 36, 50], but this comes at the cost of a loss in precision. To avoid incurring this cost, alternate approaches have been developed; offload some, or all, computation to a cloud server where the resources are available for fast inference times [58, 131]. However, offloading computation into the cloud is often infeasible due to privacy concerns, high latency, or lack of a reliable connection. As such, there is a critical need to find a way to effectively execute the DNN models locally on the devices.

This thesis seeks to offer an alternative approach to executing DNN models on embedded systems. The aim is to design a *generalisable* approach to DNN inference optimisation, making on-device inference feasible without incurring a penalty to model precision, even when compared to complex DNNs such as ResNet_v2_152. It is not always clear which DNN is best for the task at hand on embedded devices, therefore the suggested approach utilises multiple DNNs. Central to the approach is the design of an adaptive scheme to determine, at runtime, which of the available DNNs is the best fit for the input and evaluation criterion. Here, the key insight is that the optimal model – the model which is able to give the correct input in the fastest time – depends on the input data and the evaluation criterion. In fact, as a by-product, by utilising multiple DNN models it is possible to increase accuracy in some cases. In essence, for a simple input – an image taken under good lighting conditions, with a contrasting background; or a short sentence with little punctuation – a simple, fast DNN model would be sufficient; a more complex input would require a more complex model. Similarly, if an accurate output with high confidence is required, a more sophisticated but slower DNN model would need to be employed – otherwise, a simple model would provide satisfactory results. Given the diverse and evolving nature of user requirements, applications workloads, and DNN models themselves, the best model selection strategy is likely to change over time. This ever-evolving nature makes automatic design of statistical machine learning models highly attractive – models can be easily updated to adapt to the changing application context – a user simply needs to supply a set of candidate features.

In order to achieve the goals laid out above, the proposed solution is split into two parts, briefly described below:

**Model Selector.** By combining classic Statistical Machine Learning (SML) algorithms, such as K-Nearest Neighbour (KNN), with DNNs, an adaptive scheme is developed to quickly select the best pre-trained DNN to use for any given input and the optimization constraint, at runtime. An *automatic* method is proposed, able to dynamically construct the optimal predictor for each DNN problem domain; the user simply needs to supply a set of candidate features. The predictor is first trained off-line. Then, using a set of

automatically tuned features of the DNN input, the predictor determines the optimum DNN for a *new, unseen* input; taking into consideration the input and evaluation criterion. Two typical and unique DNN application domains are used as case studies for evaluation: image classification and machine translation. Evaluation is reported on the NVIDIA Jetson TX2 embedded platform, a wide range of influential DNN models are considered, ranging from simple to complex. Experimental results show that a Model Selector approach delivers portable and good performance across the two DNN tasks. For image classification, inference accuracy is improved by 7.52% over the most-capable single DNN model while reducing inference time by 44.45%. For machine translation, inference time is reduced by 25.37% over the most-capable model with negligible impact on the quality of the translation.

**DNN Specialisation.** Further to a Model Selector, a method of starting with a single seed DNN and generating a pool of smaller, specialised DNNs for the Model Selector to choose between is proposed; each pool of specialised DNNs is unique to a problem domain. For example, consider image classification, each specialised DNN tailored will be tailored to a specific subset of the entire range of possible images. An *automatic* method is proposed, able to dynamically separate the DNN training data into different segments and train a number of smaller DNNs. The user simply needs to supply a set of candidate features, the training data, and a single pre-trained DNN. Once the new DNNs have been trained *off-line* a Model Selector is generated using the above method, able to determine the best DNN to use, at runtime. Evaluation is reported on the NVIDIA Jetson TX2 embedded platform, using image classification as a case study. Using this method has the potential to further reduce resource utilisation over using a Model Selector alone, while reducing inference time and increasing inference accuracy by 5.15%. During evaluation, it was not possible to generate an accurate enough Model Selector to reach the full potential of this approach; however, further work (discussed in Section 7.3) points to further areas of investigation to reach this potential.

## 1.2 Motivation

At the time of writing there are nearly 10 billion mobile and embedded devices currently in use around the world [23] - more devices than there are humans on the planet. Furthermore, deep learning technologies are becoming increasingly popular, from image classification to virtual assistants. Many mobile and embedded devices, and their applications, would potentially benefit from the new opportunities enabled by such deep learning technologies. However, DNNs are inherently computationally and memory intensive. As a result, it is very challenging to deploy state-of-the-art DNN models in resource-constrained environments such as mobile and embedded devices.

As a result, there has been a recent push for more deep learning computation to be executed on device [58, 105]. Some work has investigated off-loading computation, however this is not always an effective solution (discussed further in Section 1.3). The advancement of mobile and embedded systems computational power and architectural diversity has made on-device computation feasible for less expensive - and less accurate - DNN models. Typically, newer devices now contain 8 or more CPU cores of different levels of energy efficiency, alongside a GPU that can be used for DNN processing [124]. Furthermore, mobile operating systems now have in-built support for DNNs; CoreML for devices running iOS [133], and TensorFlow Lite for Android devices [81]. Furthermore, recent research has investigated how to build the most effective DNN architectures for embedded devices [127]. Such recent advances indicate the demand and popularity for on-device computation. Understandably, it is now common for applications to utilise DNNs, mobile and embedded devices are a significant source of information and host of computations in modern technology.

Moreover, there are a number of benefits to on-device computation, including:

- **Lower Communication Requirements.** If all computation is done on device, there is no need to communicate with a cloud server, therefore less communication bandwidth will be used by the device.
- **Less Cloud Computing Costs.** It can be expensive for application developers to maintain, or even rent, a cloud server ready to receive requests for deep learning processing. This cost grows as an application becomes more popular too. If deep learning computation is done on device, such costs can be reduced, or even removed entirely.
- **Faster Response Times.** Further to reducing communication costs, on device computation allows for faster response times. Applications no longer depend on the quality and reliability of cloud servers or mobile network connections, the latter being notorious for unreliability. This benefit is key for mission critical applications such as health care and video surveillance.
- **Privacy Preservation.** By not communicating with a cloud server, no data needs to leave the device, allowing user privacy to be preserved. Google researchers have taken this one step further, investigating methods of DNN training on device in an effort to preserve user privacy [93].

## 1.3 Limitations of Current Work

Due to the popularity and demand for on-device computation, combined with the huge interest in DNNs in general, a number of different avenues have been explored in DNN

inference optimisation [21]; typically general purpose optimisation and rarely embedded devices specific. Relevant research into DNN optimisation can be summarised into 4 general categories: (i) reducing computational demands, by optimising the underlying operations in a DNN; (ii) efficient DNNs for hardware, building more efficient DNN architectures better suited to mobile architectures; (iii) offloading computation to a server, exploring methods to still offload computation in a smarter way; and (iv) ensemble learning, utilising more than one DNN in order to achieve higher accuracy. The benefits and drawbacks of each of the categories is discussed, in turn, below:

**Reducing Computational Demands.** There is a wide range of pre-trained DNNs available. Unfortunately, these networks are often designed to increase accuracy, without much concern for inference times. As a consequence, a number of software-based approaches have been proposed to accelerate DNNs on embedded devices. They aim to accelerate inference time using methods such as: exploiting parameter tuning [70], computational kernel optimization [7, 35], task parallelism [96, 105], and trading precision for time [53]. Work that trades precision for time often yields the greatest optimisation potential, however lower accuracy is undesirable. Since a single model is unlikely to meet all the constraints of accuracy, inference time and energy consumption across inputs [34], it is attractive to have a strategy to dynamically select the appropriate model to use. The work in this thesis presents such a capability and is therefore complementary to these prior approaches.

**Efficient DNNs for Hardware.** Methods have been proposed to reduce the computational demands of a deep learning model by: trading prediction accuracy for runtime, compressing a pre-trained network [12, 37, 106], training small networks directly [32, 107], or a combination of both [50]. Using these approaches, a user now needs to decide when to use a specific model. Making such a crucial decision is a non-trivial task as the application context (e.g. the model input) is often unpredictable and constantly evolving. The work in this thesis alleviates this user burden by automatically selecting an appropriate model to use.

**Offloading Computation.** Off-loading computation to the cloud can accelerate DNN model inference [58, 131], but this is not always applicable due to privacy, latency or connectivity issues. Recent work attempts to address the issue privacy by obscuring private data before offloading [99], or moving some computation to the device [93, 116, 138]. The work in this thesis aims to advance the effort to perform more computation on-device, making it a feasible choice when cloud offloading is prohibitive.

**Ensemble Learning.** By combining multiple DNNs together, a higher overall accuracy can be achieved. A number of works have investigated how best to do this [104, 144, 86]. The main drawback of such an approach is the huge amount of system resources it

requires; mobile and embedded systems often struggle to execute a single DNN, never mind multiple in sequence. The work in this thesis is able to utilise the benefits of ensembles without the drawbacks, by generating DNNs off-line and then adaptively selecting the best DNN to use at runtime.

## 1.4 Research Questions and Goals

It is clear that there is a demand for on-device deep learning inference for mobile and embedded systems. However, despite the efforts of the research community, there is no clear winner on a single best approach to optimise DNNs for embedded inference. This thesis presents an approach able to combine multiple DNNs into a single model, allowing a gain in inference time without a loss in accuracy. Furthermore, work that aims to improve the efficiency of DNN models can be used in conjunction with the work in this thesis, applying their techniques to the individual DNNs that the Model Selector chooses from. More specifically, this thesis posits the following hypothesis:

*By utilising statistical machine learning methods (SML), recent research efforts can be combined to create an adaptive and efficient ensemble-like approach to deep learning inference, without the added costs of conventional ensembles. Furthermore, large deep learning models can be broken down into a set of smaller models capable of achieving the same accuracy for a lower cost.*

In order to validate this hypothesis, it is broken down into a set of more specific research questions that will be easier to evaluated. The research questions will be revisited throughout this thesis in order to evaluate the progress of the work. The research questions are formalised below:

**[RQ 1]** By combining multiple DNNs, is it possible to reduce the average inference time and computational cost across a dataset without causing a reduction in accuracy? Moreover, how much can inference time be reduced by?

**[RQ 2]** Is it possible to train a statistical machine learning model to choose the optimal DNN, at runtime, depending on the input and precision requirement?

**[RQ 3]** Can orthogonal DNN optimisation techniques such as model compression be used in conjunction with a statistical machine learning model to further reduce inference time without a cost in accuracy?

**[RQ 4]** Can a set of DNNs be generated that are optimised to work together, when combined with a statistical machine learning model, that achieve even further reductions in computational costs and inference times?

## 1.5 Research Methodology

This thesis adopts an *experimental* research methodology, using an *iterative* approach based on a *quantitative* analysis of *primary* data.

**Experimental.** In order to investigate the potential effectiveness of a SML model in conjunction with deep learning models, the research questions are broken down into a set of basic experiments, such as those in Section 4.1.1. All experiments are designed to reveal interesting insights, resulting in *primary* data that is quantitatively analysed in order to inform future experiments, and update the suggested solutions. Furthermore, experimentation is used to direct the research and further explore areas of an idea that were not originally considered.

**Iterative.** By adopting an experimental research methodology, new insights are revealed during further experimentation. Therefore, exploration via experimentation is used to inform an iterative approach to repeatedly improve the suggested approaches into more refined and accurate models. An iterative approach works best for this research due to the number of interacting components that all have an impact on one another. For example, it is not immediately clear what, if any, SML model is able to effectively utilise multiple DNNs. Therefore, the design of this SML model is initially based on experimentation choosing the best models based on previous research. Furthermore, once a model is chosen, a decision needs to be made on the best features, and the best DNNs etc.; The quantitative analysis of the data from, and between, experiments allows incremental improvements to be made. Intuitively, an iterative approach allows the exploration and analysis of a diverse number of methods in order to achieve the best possible implementation. A detailed experimental evaluation is reported in order to check the feasibility of the final proposed solution.

**Quantitative.** Finally, a quantitative assessment using real-world data and deep learning models in conjunction with the suggested solutions is used for evaluation. An NVIDIA Jetson TX2 embedded deep learning platform is used for evaluation. It is a single board computer module designed for embedded applications that require high performance computing [30]. Two popular DNN application domains are considered: image classification, and machine translation; using the ImageNet ILSVRC 2012 dataset, and WMT09-WMT14 English-German newstest dataset [1], respectively.

Adopting such a research methodology allows for rapid experimentation and analysis during the exploration phase. Furthermore, a quantitative analysis means that the suggested approaches can be analysed using techniques used in state-of-the art literature, allowing for clear comparisons.

[1] http://www.statmt.org/wmt15/

# Chapter 2

# Background

This chapter presents the main concepts of Deep Neural Networks (DNNs) and Statistical Machine Learning (SML) utilised in this thesis. For clarity, Machine Learning (ML) is used to mean the broad term that includes SML and DNNs. SML specifically refers to the ML models that have a basis in statistics, such as Support Vector Machines (SVMs), whereas DNNs refers to the ML models based on Neural Networks. This Chapter begins by introducing different types of learning, as this terminology is common across DNNs and SML. For clarification and conciseness, DNNs and SML will be collectively referred to as *learning models* in this section. In addition, an in-depth background for both SML and DNNs is presented in their own subsections, before ending with a background in feature selection and engineering.

## 2.1 Types of Learning

There are a number of general categories of learning models which are defined by the learning algorithm they use. This Section introduces four of the most prominent learning algorithms: supervised, unsupervised, reinforcement, and online. Below a brief overview of each learning algorithm is given, alongside some examples of common learning models for each. It is worth noting that some learning models can fit into more than one of these categories depending on its use case.

**Supervised.** In supervised learning the learning model is provided with example inputs that are labelled with their correct outputs. Figure 2.1 presents an overview of the entire supervised learning process, from the original input data to the final predictive model. Supervised learning begins by distilling the input data into a set of *labels* and *features*. Labels are the desired output that the final predictive model should predict when given this input. The features are designed to describe the original input in as few values as possible, *e.g.* if the original input is an image, the features could be the average brightness, or the number of edges. The same features will be extracted for every input, however their

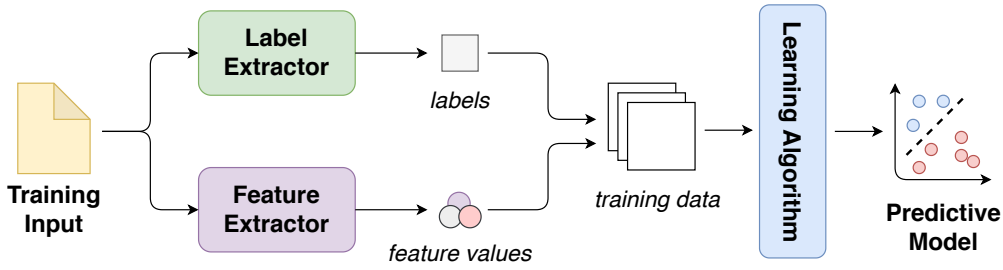



Fig. 2.1 An overview of the supervised learning process.

values will vary. The combination of features and labels is called the *training data*. The learning model can then learn from the training data and adjust its internal parameters to adapt to it through *training*. If training is successful, the learning model can then predict the output label for *new, unseen* input data. Supervised learning is commonly used in optimisation problems, where it is beneficial to predict the best solution to a problem, and the model can be trained off-line [88, 129]. Two categories of supervised learning that are used in this thesis are: *classification*, and *regression*. Classification is based on discrete output labels, *e.g.* is there a cat in this picture?; whereas regression is based on continuous output labels, *e.g.* given a person's height, can their weight be predicted?.

**Unsupervised.** In unsupervised learning the learning model is only supplied with unlabelled input data – only the features. Figure 2.2 shows an overview of the unsupervised learning process; when compared to Figure 2.1 the difference between supervised and unsupervised learning becomes clear. The learning model aims to find commonalities in the input data, and extract some meaningful information, without a reference of a correct output. Often, unlabelled data is much easier to collect than labelled data, making unsupervised learning particularly useful. Commonly, unsupervised learning is used for program optimisation [142, 91], or, commercially in recommender systems. In anomaly detection, the algorithm can draw conclusions about what a "normal" input is, and therefore indicate when an input fall outside those bounds; this is useful for detecting fraud etc. Recommender systems, such as those used by retailers to suggest new purchases, use unsupervised learning to draw conclusions about past purchases to suggest future purchases. It is worth noting that there is another kind of learning *semi-supervised learning*, where the learning model is provided with some labelled and some unlabelled data. Semi-supervised learning is not used in this thesis.

**Reinforcement.** Reinforcement learning (RL) is quite different to the learning methods described so far. In RL, the algorithm is concerned with an *agent* taking actions in some *environment* to maximise the *reward* it will collect alone the way. The idea behind RL is intuitive, if we reward good behaviour and punish bad behaviour, the agent will improve its performance. RL is useful in cases such as using ML techniques to play chess, during

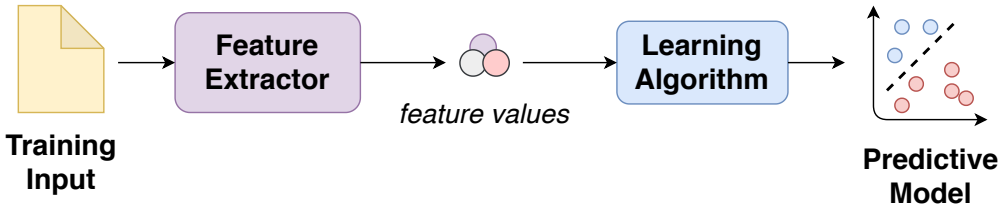



Fig. 2.2 An overview of the unsupervised learning process.

training the algorithm (agent) will learn what is a good or bad move given the current state of the board (environment). Reinforcement learning is not used in this thesis.

**Online.** Online learning is another variation of learning algorithms. In the learning algorithms mentioned above the learning model is supplied with a full training dataset and is tasked with learning its properties to make predictions in the future; the result is a *static* predictor. In online learning the learning model learns as new training data is produced, and is even able to make predictions during that time; the result is a *dynamic* predictor. Online learning is useful in environments where it is necessary for the predictor to dynamically adapt to new patterns in the data, as the patterns arise, *e.g.* stock price prediction. Online learning is not used in this thesis.

## 2.2 Statistical Machine Learning

This section introduces Statistical Machine Learning (SML), a number of common SML algorithms that are used in this thesis, and explains some common data preprocessing techniques. For clarification, Machine Learning is a general term that encompasses DNNs and SML. In this thesis the abbreviation SML is used to mean Statistical Machine Learning specifically, whereas ML is used to mean the general term Machine Learning. SML applies the principles of computer science and statistics to create statistical models which are used for future predictions (based on past data), and identifying patterns in data. Furthermore, SML has been successfully employed for various optimisation tasks, such as: task scheduling [108], cloud deployment [114], network management [135], etc.

### 2.2.1 Common Machine Learning Algorithms

This section provides an overview of some common SML algorithms, focussing on the algorithms that are used in the work within this thesis.

**K-Nearest Neighbours.** KNN is one of the most popular machine learning algorithms due to its simplicity, leading to quick inference times, and high accuracy on many tasks. In supervised learning, KNN works by plotting the input into an $N$ dimensional space, where

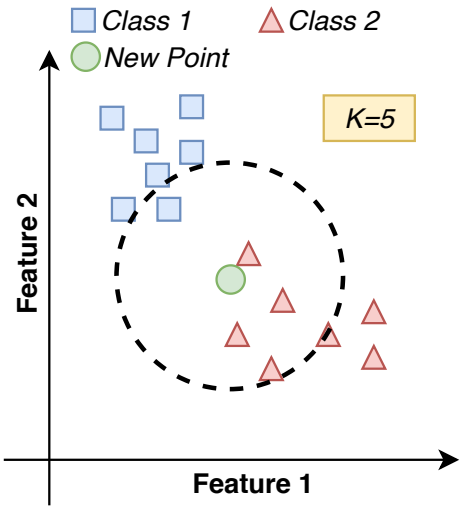



Fig. 2.3 A 2-class, 2-feature example of the K-Nearest Neighbours algorithm. The point to be classified is represented as a green circle, and its 5 nearest neighbours are enclosed in the dashed ring around it.

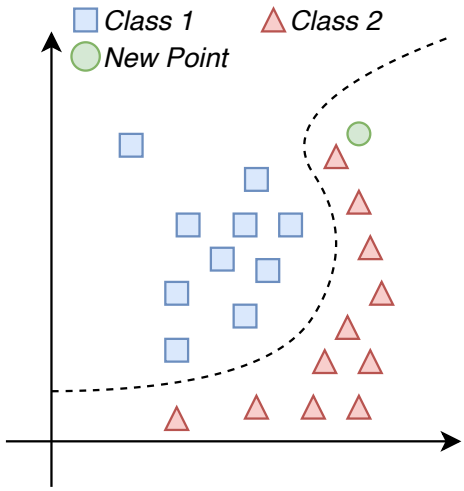
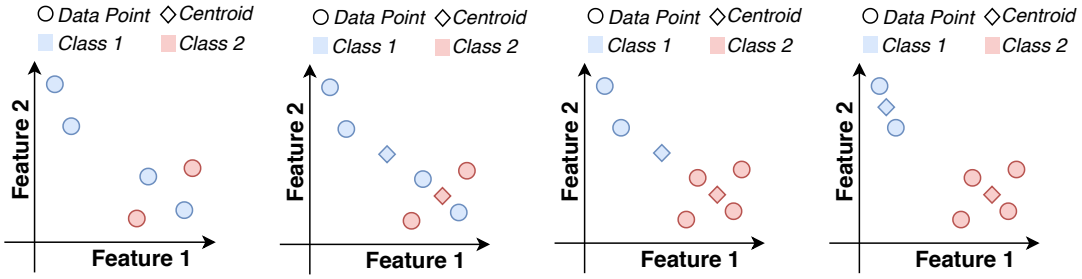



Fig. 2.4 A 2-class, non-linearly separable example of the Support Vector Machine algorithm. The hyperplane is represented as the dashed line.




(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 2.5 A 2-cluster, 2-feature example of the K-means Clustering algorithm. (a) The initial random assignment of each data point, (b) Calculate the centroid of each cluster, (c) Re-assign data points to the nearest centroid, (d) Re-calculate centroids of each cluster.

$N$ is the number of input features, and labelling each input point with its output label. During inference `KNN` takes the *new, unseen* input and finds the $k$ (a hyper-parameter which is defined by the user) closest training points, which are then used to vote for the predicted output label. An example of the `KNN` algorithm, with $k = 5$, is shown in Figure 2.3. In this example the five nearest neighbours to the new point (green circle) are: four red triangles, and one blue square; the final predicted label would be *class 2*, as it is the most frequent label. `KNN` can be used for either classification or regression; in this thesis `KNN` is used for classification.

**K-means Clustering.** `K-means` is similar to `KNN`, it is another popular algorithm due to its simplicity, however `K-means` is unsupervised. Similar to `KNN`, `K-means` works by plotting the input into an $N$ dimensional space, where $N$ is the number of input features.

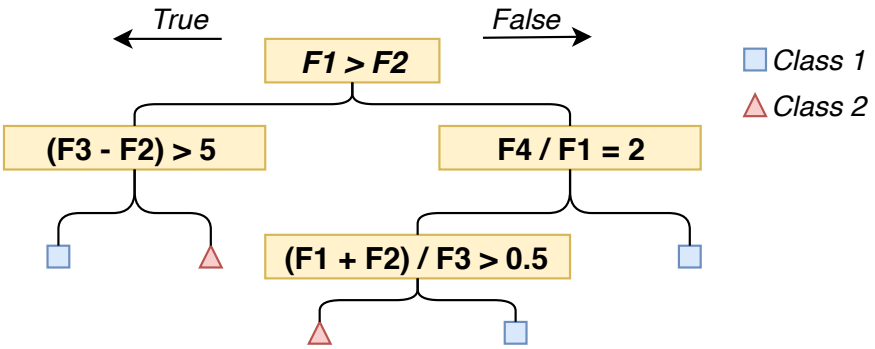



Fig. 2.6 An example of a simple decision tree. F1, F2, etc. represent different features, and each box is a different decision in the tree. Each branch of the tree ends in a leaf node representing the class that would be predicted if that branch was followed.

Figure 2.5 shows a simple example of the `K-means` algorithm. To begin, the user sets the value of $k$, in this example $k = 2$, which reflects the number of classes the data should be split into. Each data point is then randomly assigned a class (Figure 2.5a), and the *centroid* of each class is calculated, shown in Figure 2.5b. Data points are then re-assigned to the nearest centroid; the centroids are then re-calculated. The process of re-assignment and re-calculating is repeated until no more improvements can be made, that is, the mean distance between all centroids and their data points can not be reduced any more. `K-means` results in each input data point being assigned to one of the $k$ clusters.

**Decision Trees.** `DTs` are simple and computationally cheap, yet effective. An advantage of `DTs` is their ability to be easily interpretable and visualised once they are trained; Figure 2.6 presents and example of a trained `DT`. The *root* of a `DT` is the starting point of all decisions; in the example, the root is the decision '$F1 > F2$'. Every decision is either true or false, the corresponding *branch* is taken depending on the outcome of the decision. The tree ends in *leaf* nodes, which are the the class that would be predicted if that leaf is reached. During training a `DT` will try to find the best decisions that split the data most effectively. The depth and width etc. are controlled using hyper-parameters, however the deeper a tree the more complex it becomes; a balance needs to be struck between complexity and accuracy of the predictor. Some uses of `DTs` involve determining the loop unroll factor [71], and deciding the profitability of GPU acceleration [140]. `DTs` can be used for either classification or regression problems.

**Support Vector Machines.** `SVMs` are another popular `SML` algorithm that are used in supervised learning, they are more computationally complex than `KNN` and `DTs`, which allow them to solve more complex problems. During training a `SVM` will try to draw a hyperplane between training data points with different labels. In a simple case in which there is only have two classes, $A$ and $B$, the `SVM` will find the hyperplane that

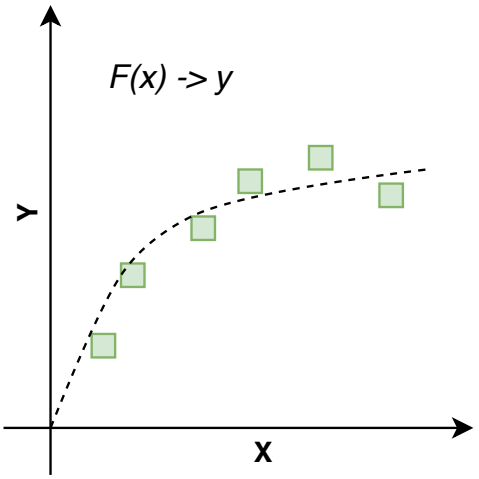



Fig. 2.7 An example of a regression-based curve fitting. Green boxes represent the 6 training examples. The dashed line shows the suggested curve, which maps values of $x$ to $y$, when plotted on the graph.

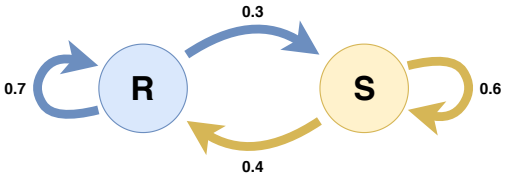



Fig. 2.8 An example of a 2 state Markov Chain. Each state can transition into every other state, including itself; transition are represented as arrows. The probability of each transition labels its arrow.

best separates the data while maximising the *margin*, that is, the space either side of the hyperplane that does not contain any data points. Figure 2.4 shows a simple example of the SVM algorithm. In this example the new point (green circle) would be classified as a red *class 2* as it is on that side of the hyperplane. Making use of *kernels* in an SVM increases the computational complexity, but allows the input of non-linearly separable data. A kernel is some functional transformation of the input data, usually to increase the number of dimensions, and allow a linear separation to be found. Once a linear separation has been found the data is mapped back into its original lower-dimensional space; this is where the curved hyperplane comes from in Figure 2.4. SVMs can be used for either classification or regression problems.

**Logistic Regression.** Logistic regression is a widely used supervised learning technique, it is very similar to linear regression, however logistic regression is designed for non-linear tasks. Contrary to SML algorithms such as KNN, regression outputs are continuous, the approach aims to map values of the input, $x$, to the output, $y$, by finding the curve, or function, $f(x)$. Simply put, logistic regression can be considered 'curve fitting'. During training the learning process for logistic regression slightly adjusts the parameters in $f(x)$ (the current learned curve) to reduce the error between the learned curve and the data points. Figure 2.7 presents an example of a trained logistic regression model. To begin, the algorithm was presented with six data points which were used to train and adjust $f(x)$ until the line shown was learned. During deployment a $y$ output value can then be predicted based on $f(x)$ for any input value of $x$. Regression has been proven

useful as a cost estimator, such as predicting execution time [83] or predicting energy consumption [109].

**Markov Chain.** Markov Chains are a common, and relatively simple, way to statistically model random processes. Conceptually, they model a number of 'states', which form a 'state space', and the probability of 'transitioning' from one state to another. Markov chains are built upon the Markov Property, which states that the transition from one state to another is only dependant on the current state and time, and is independent of the series of states that preceded it [90]. Therefore each transition in a Markov Chain has a probability of likelihood. As every state in a Markov chain can transition into every other state the complexity of the model grows exponentially as more states are added, limiting the number of states a Markov chain can effectively support. Markov chains can be used to generate data based off of a training dataset. As a simple example, a Markov Chain could be used to predict future weather patterns; whether it will be Sunny (S), or Rainy (R). To begin the user would need to supply the model with a list of past data, the Markov Chain will learn the probability of each transition from each state to create the model. Figure 2.8 shows a trained example of the Markov chain with two states: 'S', and 'R'. Markov Chains have been used in previous work for text generation [39] and financial modelling [97].

### 2.2.2 Statistical Machine Learning Feature Preprocessing

One key aspect in building a successful SML model is the features. This section introduces common feature preprocessing techniques that are used in this thesis. Depending on the problem and the domain, a different set of features will be chosen, however there are a set of standard steps a user can take to improve the model's effectiveness. These steps fall under two main categories: feature scaling and feature selection. Feature scaling is the process of scaling all features to a common range to prevent the range of any single feature being a factor in its importance. Feature selection is the process of reducing our feature count to improve the SML model's generalisability. Below, a brief overview of each of these steps is given.

**Correlation.** Checking for correlation between features reveals redundant features, *i.e.* 2 or more features which represent the same information, and drop them, reducing the computation of the SML model. To calculate the correlation between features, a matrix of correlation coefficients is constructed. This thesis uses Pearson product-moment correlation (PCC) to produce a matrix, which yields coefficient values between -1 and +1. The closer the absolute value is to 1, the stronger the linear correlation between the two features being tested. A threshold value is then chosen empirically, and any features above the threshold are removed.

**Feature Scaling.** The aim of feature scaling is to bring all features into a standard range to improve accuracy; scaling can also bring computational speed-up. Scaling does not affect the distribution or variance of feature values. Scaling can come in a number of forms, which are effective for different SML algorithms. Standardisation and normalisation will be covered here. During standardisation feature data is scaled to a mean value of 0, and a standard deviation of 1. To normalise data, it is scaled between the range 0 and 1.

**Principal Component Analysis.** (PCA) is a linear transformation technique used for feature reduction. PCA aims to reduce the feature count while maximising the feature variance; it is able to remove redundant features. The end goal of PCA is to produce a new feature-set of principal components where there is the minimum correlation (and maximum variance) between the features. Internally, PCA is more complex than simply removing the correlated features. The first principal component is chosen in such a way that it represents the most variability in the original feature-set that is possible. All subsequent principal components are chosen in a similar fashion, however they must also be orthogonal to all preceding principal components. Since the variance between the features does not depend on the SML output, PCA does not take output labels into account.

**Linear Discriminant Analysis.** (LDA), similar to PCA, is an alternate linear transformation technique used for feature reduction. However, LDA relies on output labels to reduce the dimensions of the feature-set. LDA aims to find a decision boundary around each cluster of a class. It then projects the original data points into new dimensions, such that the resulting clusters are as separate from each other as possible; the individual elements of each cluster should be as close to the cluster centroid as possible. Each new dimension is called a linear discriminant. The linear discriminants are ranked based on their ability to: minimise the distance between each centroid and its data points, and maximise the distance between each of the clusters. LDA outperforms PCA in most cases when the input data is uniformly distributed, however LDA requires labelled data.

**Feature Importance.** Often the final step in feature selection is evaluating the importance of each feature, and removing the features which have little impact on the final accuracy. A common way to evaluate feature importance is to first train and evaluate the SML model using all features, producing a score. In turn, each feature is removed and the SML model is trained and evaluated again, taking note of the reduction in score. Intuitively, the features which have little impact on the score when removed are less important, and can be removed. This process can be repeated iteratively to remove a number of features until the accuracy drop is too high.

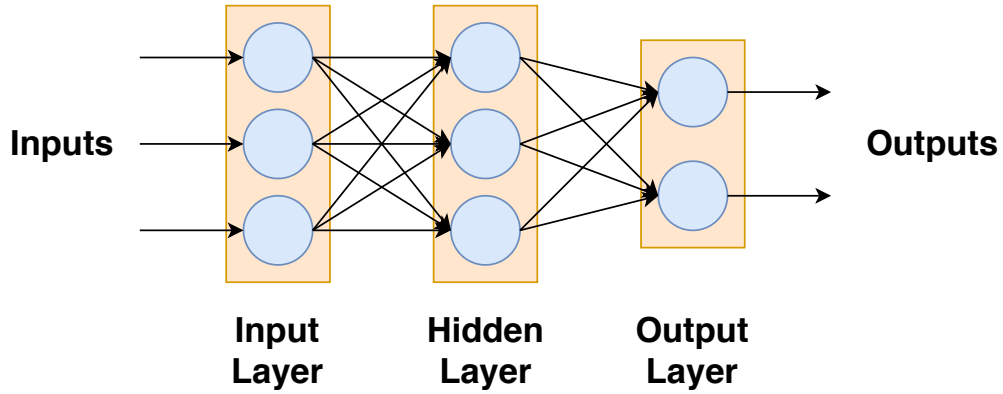



Fig. 2.9 A simple neural network consisting of 3 layers. Blue circles represent individual neurons, which are grouped into layers that are represented by the orange rectangles. Arrows represent connections between neurons.

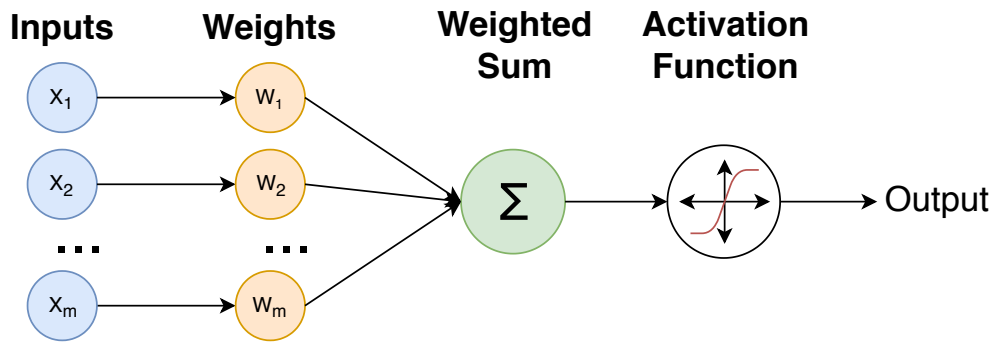



Fig. 2.10 The structure of a single neuron of a neural network. This neuron consists of *m* inputs and a single output. Each circle represents data or an operation.

## 2.3 Deep Neural Networks

SML models are relatively simple and require a user to carefully engineer a set of features. Deep Neural Networks (DNNs) are capable of solving much more complex problems, and, in some architectures, can learn the features for the user. This section introduces DNNs, their terminology, how they work, and some of their applications. In some contexts DNNs are referred to as Deep Learning (DL), which is seen as a subset of Machine Learning that refers specifically to DNNs. To begin, this Section gives a brief introduction to DNNs before moving on to introduce and describe some DNN terminology. Furthermore, this section describes the special DNN architectures that are specific to this work. Finally, some relevant applications of DNNs are described.

### 2.3.1 Structure

Neural Networks (NNs) are inspired by the human brain, which is made up of neurons, and connections; connections are formed between neurons. DNNs are a special 'Deep' variant of NNs (here 'Deep' simply means the network is large, defined in more detail

below) and can be seen as a computer simulation of a brain. DNNs are made up of two main components: *neurons* and *connections*. Below, these two components are described in more detail. They will then be built upon, introducing new components, until a simple DNN (shown in Figure 2.9) is created.

**Neuron.** Neurons are a basic unit of a NN that consist of inputs, an activation function, and an output. In basic NNs, neuron output is calculated as the weighted sum of the inputs followed by the neuron's activation function, which is usually non-linear. This can be seen in Figure 2.10, *m* inputs (blue) are multiplied by their individual weights (orange), the sum is taken (green) followed by the neuron's activation function. The weight of each input is decided by the input connection. Neurons are organised into *layers*. In more complex and specialised NNs, neuron output can be calculated differently; explained in more detail in Section 2.3.3. Unless a neuron is part of the *input layer* all of its inputs are received via connections from other neurons. Finally, unless the neuron is part of the *output layer* its output is sent via a connection to one or more neurons.

**Connection.** Connections are how neurons communicate with each other. Every connection connects one neuron to another, and has a weight. Except between special layers (described in Section 2.3.3), the output of the sending neuron is multiplied by the weight to either increase or decrease the signal given to the receiving neuron. In Figure 2.10 the inputs are the outputs of connected neurons. The weight of a connection will change during *training*.

**Layer.** A layer consists of multiple neurons which are not connected to each other. Layers are connected when the neurons in one layer are connected to the neurons in another. Figure 2.9 shows a simple NN consisting of three layers, an input, output and hidden layer. Each layer is represented by an orange rectangle; note that the connections are still formed between neurons and not layers. If a layer has no predecessor it is an *input layer*, if it has no successor it is an *output layer*. All layers between the input and output layers are *hidden layers*. If the number of hidden layers is large (in general, more than 8 [66]) then a NN is considered a DNN. Modern DNNs can have hundreds of layers [149]. There are many different types of specialised layers which are described in more detail later.

### 2.3.2 Terminology

A useful DNN is one that can take a *new, unseen* input and make an accurate prediction through a process called *inference*. In order to create an accurate DNN it will need to be *trained* on some input data (a *training dataset* in the context of this thesis) using some set of *hyper-parameters*. Below, each of these terms are described in more detail. Pre-trained networks, transfer learning, and fine-tuning are also described in detail below.

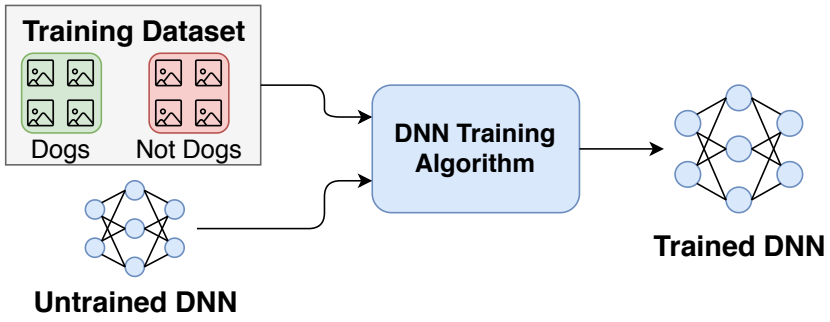



Fig. 2.11 A simple example of training a DNN to differentiate between images that either do or do not contain dogs. Each class label in the training dataset (*e.g.*'dogs') contains hundreds or even thousands of images.

**Training Datasets.** Training an accurate DNN typically requires a larger amount of data compared with traditional NNs. Generally, more and higher quality data results in a better trained DNN. In this case data means a set of inputs for the DNN along with their desired output, that is, their *class*, or *label*. As a simplified example, to train a DNN to predict whether an image contains a dog or not then it needs to be provided with two *classes* of data; Figure 2.11 shows this example. The first is a set of images of dogs, with the label of 'dog'; and the second is a set of images of anything but dogs, labelled as 'not dogs'. In real-world applications there are typically hundreds or even thousands of classes that the DNN will choose between during *inference.* Collecting enough data to train an accurate DNN then becomes a huge task. To get around this problem there are a set of standard datasets for different tasks, each containing huge amounts of data. Two different datasets are used in this thesis: ImageNet ILSVRC 2012 dataset [113] (often referred to as ImageNet), used for image classification; and WMT English-German newstest dataset [1], used for machine translation. It is worth noting that smaller 'toy' datasets are also available, such as MNIST [72] and CIFAR-10 [65]. These datasets consist of fewer classes (around 10, 100x less than ImageNet) and are often used as a proof of concept for DNN optimisation tasks, however the results do not always carry over to more complex problems.

**Training.** Training a DNN is the process of incrementally updating its weights to reduce the error of predictions, and increase overall accuracy. *Gradient descent* is commonly used to train DNNs, and is based on an algorithm called *back-propagation*; it is an iterative process of *inference*, *back-propagation*, and *weight update*. During training the DNN is given an input, and it produces an output. The inference result could be right or wrong, which is quantitatively measured by the *loss function*. Next, the gradient of the loss function is calculated using back-propagation for each neuron and weight, working from the output layer backwards. Finally, the weights are updated using *gradient descent*.

[1] http://www.statmt.org/wmt15/

Training a DNN often takes hundreds of thousands of iterations; more training data leads to more iterations. Training is often measured in *epochs* instead of iterations, where each epoch represents a pass over the entire training dataset.

**Inference.** The process of a DNN calculating its output for a given input is called inference, also known as a *feed-forward* pass. During inference the input values are given to their respective input neurons, which calculate their output. The outputs are then passed to the next layer in the network via connections. This process is then repeated for every hidden layer in the network until reaching the output layer, where its output is the output of the network.

**Hyper-Parameters.** DNN hyper-parameters are settings that can be tuned to influence the DNN's behaviour, they are parameters of a DNN that cannot be learned from the training data. Attempting to learn DNN hyper-parameters directly from the training data leads to over-fitting, creating a model unable to generalise well to all data. The number and type of hyper-parameters depend on the type of DNN. Some examples include: learning rate, which influences the learning progress of a DNN; number of hidden layers, either increasing or decreasing the model capacity; and convolutional kernel size, deciding the size of a kernel in a convolutional layer (see Section 2.3.3). To guide hyper-parameter tuning during training a validation set of the training data is usually left out. The validation set is then used to test the trained model's generalisability to new data. Hyper-parameter tuning is usually an expensive and time-consuming process, leading to automatic hyper-parameter tuning libraries being included in frameworks such as TensorFlow.

**Pre-Trained Model.** Training a DNN requires a lot of computational power and time. Bigger networks, more training data, and high quality training data typically leads to a more accurate DNN. Unfortunately, it can also lead to long training times when specialised hardware is not available. For example, training the simplest version of ResNet (`ResNet_v2_50`) on the ImageNet dataset for 90 epochs using an NVIDIA M40 GPU takes 14 days [151]. To make DNNs more accessible independent researchers have trained state of the art DNN architectures on complex and freely available datasets such as ImageNet ILSVRC dataset. They then release the weights of their trained DNN and its architecture to the public. A pre-trained DNN is a combination of the weights and architecture, which can be downloaded.

**Transfer Learning.** A pre-trained DNN can be used to bootstrap the training of a different DNN through transfer learning. The central concept of transfer learning is to use a more complex, but successful, DNN to 'transfer' its learning to a simpler problem. The key idea behind transfer learning is that the earlier layers in the DNN have learned some useful information, and its hyper-parameters have already been found to be useful.

Therefore, if the later layer(s) are replaced, they can be retrained for the new problem. For example, starting with a DNN pre-trained on the ImageNet ILSVRC dataset, transfer learning can be used to only determine the breed of dog contained in the input image. Transfer learning can take two approaches: simpler fine-tuning, or the more complex representation learning. In fine-tuning only the last couple of layers are replaced and retrained, to represent the new problem the DNN is solving. These layers are responsible for producing the DNN output values, typically label probabilities. To limit training to only the new, replaced layers, all other layers would need to be *frozen*. Representation learning replaces the last few layers of the DNN with a wholly new model, using the pre-trained model as a kind of feature extractor. Representation learning is more complex than fine-tuning as the user is required to design a new NN for their task, however, it is more adaptable.

### 2.3.3 Neural Network Architectures

This section gives an overview of the different types of neural network architectures, and the layers that make each of them unique. The following sections cover: Multi-Layer Perceptrons (MLPs), and explain *fully connected layers*; Convolutional Neural Networks (CNNs), introducing *convolutional*, and *pooling layers*; and Recurrent Neural Networks (RNNs).

#### Multi-Layer Perceptrons.

MLPs consist of at least three layers: an input layer, a hidden layer, and an output layer; although they can consist of any number of hidden layers. All neurons, except those in the input layer, use a non-linear activation function. All layers in a MLP are fully connected in sequence, that is, all neurons in $layer_i$ are connected to all neurons in $layer_{i+1}$. Figure 2.9 is an example of a small three-layer MLP. Sometimes, MLPs are referred to as "vanilla" neural networks [42]. MLPs are not directly used in this thesis, however the architectures that are used build upon them.

#### Convolutional Neural Networks

For tasks such as image processing, MLPs suffer from several drawbacks, for example spatial information is lost when flattening the image for input into an MLP. CNNs are good at capturing spatial information from the input (such as images) through the use of *convolutional layers*, which perform *convolutions*, and *pooling layers*, which shrink the output and reduce noise within the network. Typically, one or more convolutional layers are followed by a pooling layer. Convolutional and pooling layers are described in more details below.

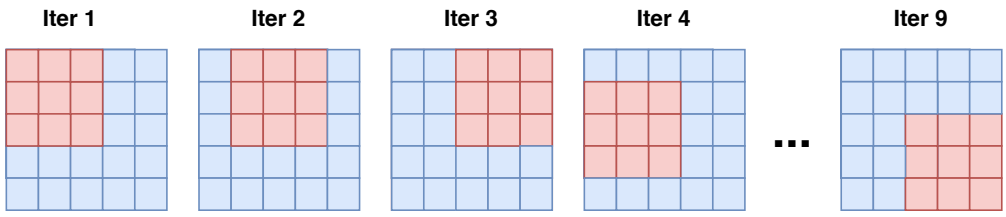



Fig. 2.12 A red convolutional filter (3x3) is sliding over the blue layer input (5x5). This convolutional layer has a stride of 1, and no padding. The filter slides from left to right, top to bottom.

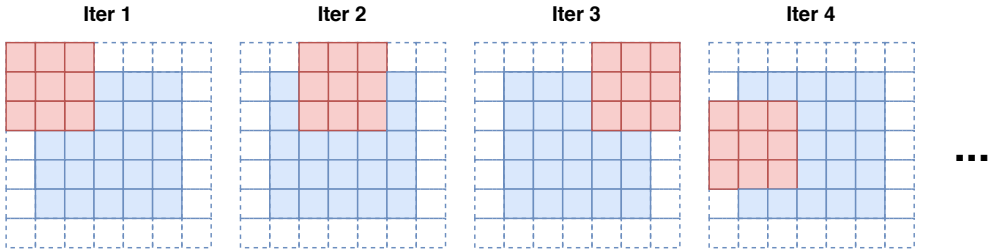



Fig. 2.13 A red convolutional filter (3x3) is sliding over the blue layer input (5x5). This convolutional layer has a stride of 2, and a padding of 1. The filter slides from left to right, top to bottom.

**Convolutional Layers.** Convolutional layers perform convolutions in place of general matrix multiplication, which is typically used in `MLPs`. Each convolutional layer is made up of a number of filters of a specified size (usually 3x3 or 5x5). The filters are 'moved' across the layer input from left to right, top to bottom, performing a matrix multiplication and producing a single output value during each iteration; the layer is convolving the input. Figure 2.12 shows a simple 2D convolutional filter passing over a layer input. As the iteration increases, the filter moves one step right until reaching the edge of the input (iter 3) where it continues on the next row (iter 4). Convolutional layers can also be tuned to have different levels of padding and stride; these values are set per convolutional layer, and are another example of hyper-parameter. Figure 2.13 shows how the convolutional filter interacts with the input when the padding and stride are set to 1 and 2, respectively. For clarification, Figures 2.12 and 2.13 show a single filter of a 2D convolutional layer, this process is repeated for each filter in the layer. In 3D convolutional layers, each filter is the same depth as the layer input, in Figure 2.14 this is 3. During training each filter will learn to recognise a specific "feature", *e.g.* eyes, or teeth. Therefore, as each filter slides across the whole input, a `CNN` is able to recognise what an image contains, rather than where an object is. The output of a filter is then passed through an activation function, typically ReLU, which decides if a feature is present at each location in the image, producing a feature map. The output size of a convolutional layer depends on a

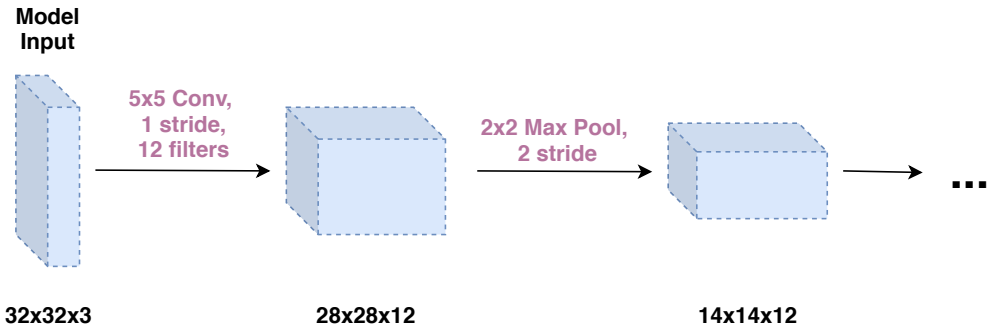



Fig. 2.14 How the data value changes shape as it passes through each layer. Equation 2.1 shows the equation used to calculate output sizes. Data volumes are represented as blue cuboids, and layers are represented as arrows, described in purple text. Input is a 32x32 image with 3 colour channels.

number of factors, and can be formalised in the following equation:

$$\frac{input\_size - filter\_size + 2 \times padding}{stride} + 1 \tag{2.1}$$

Figure 2.14 shows an example of this equation. Filter outputs are stacked depth-wise meaning the layer output depth is equal to the number of filters. Typically feature maps are then passed through *pooling layers* to shrink the output and reduce noise.

**Pooling Layers.** Pooling layers are very similar to convolutional layers, they are a special type of convolutional layer used to shrink the output and reduce noise in the network. Pooling works in a similar way to filters, although the pooling operation is specified and not learned, and is typically average pooling, or max pooling. In average pooling, the output value is the average of all the values that the pooling filter can 'see', and in max pooling the output value is the maximum. 3D pooling is performed on each 'depth slice', meaning the depth is equal before and after a pooling layer. Figure 2.14 shows how the convolutional and pooling layers change the size and shape of data as it passes through the network.

CNNs are usually computation bounded due to the number of matrix multiplications required in each layer, therefore much research has gone into reducing the amount of computation in each convolutional layer. This thesis makes heavy use of CNNs as a case study, this is due to its easily accessible nature; the approaches described in this work have been designed to be applicable to all domains of DNNs.

### Recurrent Neural Networks

RNNs are good at capturing temporal information from the input, such as speech. Unlike the neural network architectures explained above, RNNs contain self connections, where they get their name from, which serves as a kind of *memory*. Rather than individual layers having self connections, RNNs are built of recurring *modules*; different module

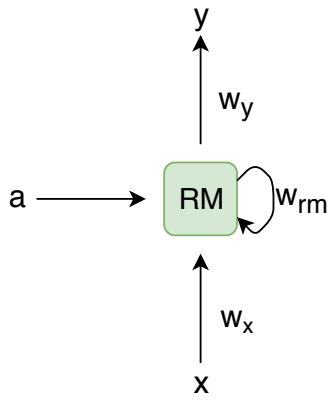



Fig. 2.15 A simple recurrent neural with its inputs, outputs and recurring module (RM). W variables are the weights for the following connections: $W_x$, for the connection from the input layer to RM; $W_y$, for the connection from RM to the output layer; and $W_{rm}$, for the connection of RM to itself. $a$ is the activation of the previous layer. Figure 2.16 shows an unrolled version of this RNN.

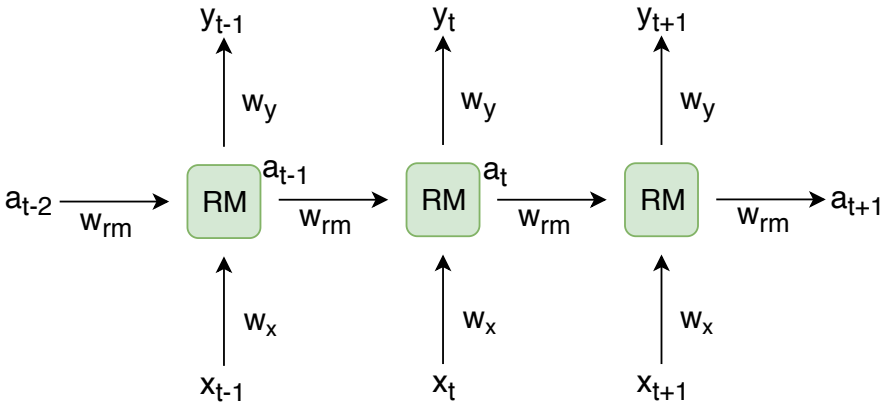



Fig. 2.16 An unrolled version of the RNN shown in Figure 2.15. The subscript of $x$, $y$ and $a$ variables represent their time step ($t$). The RM produces an $a$ value during every $t$ which is used during the next $t$ ($t+1$). Connection weights are the same for every time step.

structures separate different sub-architectures of RNNs. Therefore, when an RNN makes a prediction on an input it is not only considering the input it received at time $t$, but also the previous input at time $t-1$. Figures 2.15 and 2.16 show an example of a simple RNN, and how it interacts with itself between values of $t$. Two popular variants of RNNs are described below: GRU and LSTM networks.

**Gated Recurrent Unit.** (GRU) networks [14] are a popular variant of RNN. The key idea is the 'cell state' which allows a GRU network to retain any information without much loss. A GRU network contains two 'gates': the update gate, and the relevance gate; they help to regulate the flow of information to the cell state. The relevance gate uses a sigmoid function to decide how relevant the information from the previous time step is, while calculating the current cell state; it outputs a candidate updated cell state. The

update gate also uses a sigmoid function, it decides the portion of the previous cell state to retain based off of the candidate updated cell state (which was output by the relevance gate). During training, the gates learn which data in a sequence is important and which is not – they learn which information is worth keeping – allowing a GRU network to pass information in long sequences. GRU networks are relatively simple and require much less computational power, so can be used to form deeper networks.

**Long Short-Term Memory.** (LSTMs) networks [48] are another popular variant of RNN. LSTMs are capable of both long and short term temporal dependencies; they have shown to be effective for many tasks, particularly when the input consists of long sequences. In a GRU module, the cell state is equal to the output value, however that is not the case in an LSTM, they are different things. To facilitate this, an LSTM contains three gates: the forget gate, the update gate, and the output gate. The update and forget gates are responsible for updating the current cell state by filtering the previous cell state and candidate cell state. The output gate decides what to output based on the current cell state (after the update and forget gates) and the input. Relative to GRU networks, LSTM networks are slower and more complex, however LSTMs are more powerful.

Unlike CNNs, RNNs are typically memory bounded. This thesis makes use of RNNs as a case study of machine translation.

### 2.3.4 Applications

Many applications can benefit from DNNs, ranging from machine translation to medicine. This thesis presents work which can be applied to DNNs in general, and therefore is a benefit to all applications. In order to facilitate the evaluation of the contributions of this thesis, it has been applied to two common DNN domains: image classification, and machine translation. These domains have been chosen due to their easy accessibility and large usage. Below a brief overview of each domain is provided.

**Image classification.** The task of automatically labelling an input image to describe its contents is known as image classification. There are many useful applications of image classification such as facial recognition. With the help of the yearly ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [113] plenty of recent research has improved the accuracy of DNNs designed for this task. Currently, the most powerful networks are able to match humans for accuracy in the ILSVRC challenge, however these networks also come at a high computation cost.

**Machine Translation.** The most well known application of machine translation is Google translate, which aims to make translation between languages easily accessible. However, machine translation has proven to be a more difficult task than image classification and there is still plenty of research focussing on how to improve it [20]. Evaluation

of this work uses the WMT newstest dataset, although there are a number of others available. Machine translation typically makes use of RNNs, which are typically memory bounded.

# Chapter 3

# Related Work

Deep Learning (DL) has proven its ability in solving many difficult problems such as: object recognition [24, 43], facial recognition [102, 123], speech processing [2], and machine translation [4]. However, the DNN architectures that have been developed to solve these problems are often resource intensive tasks, consuming a considerable amount of CPU, GPU, memory, and power [9]. While running on powerful, specialised servers, their resource consumption is often an afterthought, however many of these tasks are also important application domains for resource-limited embedded systems [68]. Therefore DNNs need optimising, otherwise the gap between the resources available and the resources required will lead to long inferencing times, making real-time applications infeasible. The work in this thesis makes use of statistical machine learning (SML) techniques to optimise DNNs for embedded systems. This section briefly reviews previous relevant research into DNN optimisation, and work that utilises SML for different optimisation tasks. Although this thesis focusses on improving inferencing times, the similar work that is presented below often uses DNN parameter count (the total number of weights and biases in a model) to evaluate their approaches. A reduction in parameter count does not necessarily lead to a reduction in runtime, what matters more is sparsity and the underlying hardware; these points are explained in more detail in Sections 3.1 and 3.2, respectively. The areas of DNN optimisation that are covered are as follows: reducing computational demands, efficient DNNs for hardware, offloading computation to a server, ensemble learning, and improving DNN training. The final subsection focusses on applications of machine learning in optimisation.

## 3.1 Reducing DNN Computational Demands

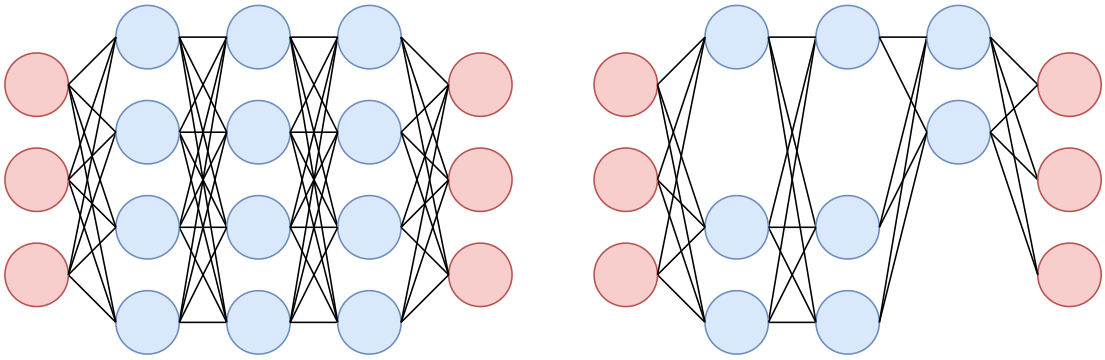
DNNs are a powerful tool. They have been able to achieve notable successes in various tasks that previously seemed difficult [66, 74, 14]. There is a wide availability of pre-trained state-of-the-art DNNs for a wide variety of domains, such as image classification.

Table 3.1 An overview of the work presented in Section 3.1. Due to the lack of agreed upon benchmark(s) a number of different datasets and base models have been used to evaluate work. This has also lead to a number of footnotes needing to be added for some work. See below the table for more information. Size is the rounded number of parameters of the model. Accuracy is the top-1 accuracy for image classification problems. NSI is an abbreviation of No Significant Impact.

| Paper | Description | Dataset | Base Model | Size (Reduction%) | Accuracy (Reduction%) |
|---|---|---|---|---|---|
| Base AlexNet Model [66] | - | ILSVRC | AlexNet | 61M | 62.5% |
| Base VGG-16 Model [118] | - | ILSVRC | VGG-16 | 138.4M | 71.3% |
| Base Inception Model [125] | - | ILSVRC | Inception | 6.8M | 68.7% |
| Han et al. [37] | Per layer pruning | MNIST | AlexNet | 6.7M(89%) | NSI |
| Han et al. [37] | Per layer pruning | MNIST | VGG-16 | 10.3M(92.5%) | NSI |
| Srinivas et al. [120] | Per layer pruning | CIFAR-10 | AlexNet | 36.6M(34.6%) | NSI |
| He et al. [45] | Per layer pruning | LVCSR | Unclear | 13.1M(40.7%) | NSI |
| Luo et al. [84] | Per layer filter pruning | ILSVRC | VGG-16 | 1.22M(99.1%) | 62.97%(-8.33%) |
| Hu et al. [51] | Per layer filter pruning | ILSVRC | VGG-16 | 65.6M(52.6%) | NSI |
| He et al. [46] | Per layer filter pruning | ILSVRC | ResNet-101 | (41.2%)[a] | NSI[e] |
| Fang et al. [29] | Per layer filter pruning | ILSVRC-100[b] | ResNet-50 | (Up to ~17.2%)[c] | NSI - 73%(~-20%) |
| Molchanov et al. [94] | Per layer filter pruning | ILSVRC | VGG-16 | (54.5%)[d] | 84.5%(-4.8%)[e] |
| Anwar et al. [3] | Per layer quantisation | MNIST | Custom NN | - (~90%) | NSI |
| Wu et al. [147] | Per layer quantisation | ILSVRC-12[f] | AlexNet | ~3.3M(94.7%) | NSI |
| Chen et al. [12] | New NN architecture | MNIST | Custom NN | - (87.5%) | NSI[g] |
| Hwang et al. [54] | New NN architecture | TIMIT | Custom NN | - | NSI[g] |
| Khoram et al. [62] | Per layer quantisation | ILSVRC | VGG-16 | ~43.7M(~68.4%) | NSI |
| Han et al. [36] | Per layer optimisation | ILSVRC | AlexNet | ~1.8M(97.1%) | NSI |
| Howard et al. [49] | New NN architecture | ILSVRC | MobileNet_V3 | 5.4M | 75.2% |
| Zoph et al. [158] | New NN architecture | ILSVRC | NasNetMobile | 7.7M | 74.4% |
| SqueezeNet [55] | New NN architecture | ILSVRC | AlexNet | ~1.2M(98%) | NSI |

[a] Only presented as a reduction in FLOPs.
[b] A subset of ImageNet is used, containing only 100 classes.
[c] Presented as a range of different models. Smaller models are less accurate.
[d] Only presented as a time speedup on NVIDIA TITAN X.
[e] Only the top-5 accuracy is given.
[f] A subset of ImageNet is used, containing only 12 classes.
[g] Compared to a comparable non-quantised NN.

Unfortunately, these networks are often designed to increase accuracy, without much concern for inference times. As a consequence, there has been much research into the optimal trade-off between accuracy and inference runtime. This section investigates current approaches that take a pre-trained model and reduce its computational complexity while having minimal impact on accuracy. Pruning, quantization, and other miscellaneous methods are discussed, finishing with a short summary. Each subsection will begin with a brief overview of the approach, followed by a discussion of recent work in the area. It is worth noting that the community is yet to agree on some benchmark(s) to measure all methods, therefore comparisons between work can be difficult. MLPerf [92] is a recently published machine learning benchmark suite that aims to become a “de facto” benchmark



(a) Before Pruning (b) After Pruning

Fig. 3.1 An example of pruning on a simple DNN, shown in (a). The input and output layers are coloured red to show that they cannot be pruned. After pruning (b), the chosen neurons (blue circles) are removed from the network along with their connections. The pruned network uses less computational resources.

suite in order to make future research more comparable. To help frame current work Table 3.1 presents an overview of work in this area.

### 3.1.1 Pruning

Neural network pruning is based on the idea that neural networks are often over-parametrized. Some neurons are redundant and have little impact on the overall output; an idea initially introduced back in 1990 by Yan Lecun [73]. Figure 3.1 shows an example of how pruning works. For this example, the removed neurons have been chosen at random; a smart pruning method would carefully pick which neurons to remove. Figure 3.1b shows that each neurons respective connections are also removed as they are no longer needed, further reducing computational demands of the network.

If every neuron in a network can be ranked according to how much they contribute, those which have minimal effect on the final output can be removed. However, removing a neuron will result in a drop in the overall network accuracy, and some fine-tuning is required for the model to recover. Therefore, pruning is usually an iterative process of prune, fine-tune, and repeat. Successful pruning will result in a faster, smaller, and more generalised network. Two different types of pruning are presented below: *standard pruning*, which focusses on pruning fully connected layers; and *filter pruning*, which focusses on pruning convolutional filters from CNNs.

#### Standard Pruning

Until recently, most pruning research focussed on the fully connected layers; specifically on how to effectively rank neurons. Early approaches to neuron ranking were based on Biased Weight Decay [38] and the Hessian of the loss function [73, 41]. Magnitude based ranking methods, such as Biased Weight Decay, are less effective than those based

on the Hessian of the loss function. However, the methods proposed in [73, 41] are more computationally expensive. In more recent work such as [45], three metrics are proposed to rank the neurons in a layer. All three of the metrics achieved similar results, able to prune the DNN to around 40% of its original size. However, in [110] the authors show that a blind pruning method that randomly selects the nodes to be removed achieves similar results. In [37], the authors show that removing the neurons that have an absolute weight value near zero can be an effective method, reducing the number of connections in AlexNet by 90% without a drop in accuracy. While this is a less computationally expensive method, it is less effective. The pruning is done on a per layer basis, and is based on the assumption that weights are always distributed around zero within a layer, which is not always true. Furthermore, the threshold for 'near zero' is based on a quality parameter which needs to be manually chosen per layer, which is infeasible for large networks. An alternate, computationally cheaper method to pruning was presented in [120]; here they avoid the expensive fine-tuning step. Their method is based on finding the difference between pairs of neurons, which they name *saliency*, and removing those with the lowest saliency. Unfortunately, this method only achieves a 35% reduction in AlexNet, much lower than other methods. When a pruning method does not utilise fine-tuning there is often a trade-off in computation and pruning effectiveness.

Uses training data to determine the pruned filters?

#### Filter Pruning

In CNNs the pruning operation needs to be slightly different. Using standard pruning methods will only prune some of the weights in each convolutional filter, resulting in sparse filters, termed non-structured sparsity. It is a non-trivial task to reach the full potential of computational speed up given by non-structured sparsity. In this scenario, even though some of the weights have been removed the user may not see any benefit as the entire convolutional filter still requires full execution to allow the model to continue. In [145, 76] the authors propose a structured sparsity method of CNN pruning, removing entire convolutional filters, therefore avoiding the sparsity problem. An added benefit of removing convolutional filters is the reduction in memory use. A number of methods have been proposed to choose which filters should be pruned. In [84] the authors propose using $layer_{i+1}$'s statistics to guide the pruning of $layer_i$, while a successful method, this is computationally heavy due to the analysis needed at each layer. Hu et al. [51] propose using the percentage of zero activations of each filter. They show that this method is mainly effective for deeper convolution layers and fully-connected layers in large CNNs. In [29] the authors propose a ranking method named Triplet Response Residual (TRR), based on the intuition that a filter is important if it is able to extract features that are useful to differentiate images belonging to different classes. TRR is able to outperform L1-Norm ranking method, achieving much higher accuracy when the same percentage

of the model is pruned. However, calculating TRR is computationally expensive and becomes infeasible with large numbers of filters and/or classes. Another ranking method was proposed in [94], based on first order Taylor expansion of the network cost function. The aim was to choose the best filters to remove while having minimal impact on the absolute network cost. In this work they were also able to carry out a brute force search for the best filters to prune for VGG-16, and show that Taylor expansion based pruning methods achieve the best performance. Alternate methods have been proposed where a pre-trained model is not required. In [46] the authors propose soft filter pruning where pruned filters are allowed to recover during training; achieving on par performance to a pruned pre-trained model. Lin et al. [79] present a new method of ranking filters for removal using a mask, yielding more pruning over typical pruning approaches. However, their approach requires the method for calculating loss to be altered, resulting in the need for an expensive retraining step, alongside extra hyper-parameter tuning. Therefore, their approach can be significantly more expensive than typical pruning techniques.

### 3.1.2 Quantization

Neural network quantization focusses on reducing the precision and bit-width of the weights and activations within a model; predominantly 32-bit floating point has been used. Quantization is an effective method of reducing the bandwidth and storage of a DNN. Furthermore, when integer representation is used in place of floating point there is a reduction in overall computation. Numerous works such as [136] have demonstrated that weights and activations can be represented using 8-bit integers without incurring a significant loss in accuracy. In order to convert the floating point representation into an integer, a *scale factor* is introduced to map the original dynamic range into the integer format range. This leads to a lower resolution of representable values, therefore a reduction in overall precision. The mapping between floating point and integer values has a big impact on the overall resulting performance.

The work presented in [3] quantized the neural network using L2 error minimization; quantizing the layers one by one. Wu et al. [147] applied k-means clustering quantization to the parameter values. HashedNets are presented in [12] which reduce bit-width though the use of hash functions to randomly group connection weights. All three of these works show promising results, as shown in Table 3.1, shrinking the NN by over 87% each. However, their results are based on simple problems and small NN, and it is unclear whether the results carry over to bigger problems and models. Alternatively, more extreme methods of bit-width reduction have been suggested. In [54] the authors proposed ternary weights and 3-bit activations, achieving only negligible performance loss. Binary networks have also been suggested, such as BinaryConnect [17] and BinaryNet [18]; here the weights and activations are limited to either $+1$ or $-1$. However,

the presented methods have not been proven on large datasets such as ILSVRC 2012, only smaller toy datasets such as MNIST and CIFAR-10. All the above methods equally reduce the precision and bit-width of the entire network at once. The work presented in [62] allows for unique quantization precisions for each parameter. However this is currently only applicable to specialised hardware.

### 3.1.3 Other Methods

This subsection covers a number of methods that have been suggested to reduce the computational complexity of DNNs that do not fit into either pruning or quantization.

Han et al. [36] present Deep Compression, a three stage pipeline of pruning, trained quantization and Huffman coding. Deep Compression was shown to reduce AlexNet by 35x and VGG-16 by 49x. However, this work still suffers from manually tuning the threshold parameter for pruning for each layer, making this work infeasible for larger DNNs. Replacing the fully connected layer with global average pooling has been suggested as another method of reducing the number of parameters of DNNs. The work in [125, 78] show success with this approach, achieving state-of-the-art results on a number of benchmarks. Unfortunately, this approach makes fine-tuning the networks for a different task more difficult.

Finally, some work suggests entirely new DNN architectures, built with the objective of using less parameters from the ground up [49, 158]. SqueezeNet [55] was able to achieve AlexNet level accuracy with 50x fewer parameters; this can be reduced further when used in combination with Deep Compression. AlexNet does not achieve state-of-the-art results, therefore SqueezeNet is mostly used in applications where model size is a priority and not accuracy. EfficientNets [128] are a new DNN architecture, designed to be a smarter way of scaling up CNNs in order to use less parameters to achieve similar or better performance. A family of DNNs are presented which trade-off accuracy for faster inferencing times; however it is still down to the user to decide which is the best model to use, a non-trivial task. Recently researchers have investigated methods to automatically generate new DNN architectures [16, 157]. This is a promising area of research, removing the need for experts or off-the-shelf DNNs for the typical user; however they are computationally expensive while searching got the most efficient DNN for big datasets. Such work could be used in conjunction to the work presented in this thesis, able to automatically design the best DNN for sub-sets of the full dataset.

### 3.1.4 Summary

Methods have been proposed to reduce the computational complexity of DNNs by trading prediction accuracy for runtime through: pruning, quantization, unique architectures, training small networks directly, or some combination of those methods. These methods

Table 3.2 An overview of the work presented in Section 3.2. Due to the lack of agreed upon benchmark(s) a number of different base models and hardware platforms have been used to evaluate work. This has also lead to a number of footnotes needing to be added for some work. See below the table for more information. Speedup is given on a mobile GPU unless otherwise stated. Accuracy is the top-1 accuracy for image classification problems. NSI is an abbreviation of No Significant Impact.

| Paper | Description | Dataset | Base Model | Speedup | Accuracy (Reduction%) |
|---|---|---|---|---|---|
| Han et al. [35] | Custom Hardware | ILSVRC | AlexNet | 92.3% | NSI |
| SparseSep [7] | Compiler and Runtime | ILSVRC | VGG-16 | 63% | NSI[a] |
| CNNDroid [70] | `NN` GPU accelerated library | ILSVRC | AlexNet | 97.6%[b] | NSI |
| AutoTVM [11] | `DNN` optimisation framework | ILSVRC | MobileNet | 22.7%[c] | NSI |
| Lane et al. [67] | Resource control inference software | ILSVRC | AlexNet | 75.5%[d] | (-4.9%)[e] |
| Huynh et al. [53] | Optimising inference software | ILSVRC | VGG-16 | 74.4% | 83.94%(-6%)[f] |
| Motamedi et al. [96] | Thread granularity optimisation | ILSVRC | AlexNet | 46.3%[g] | NSI |
| Song et al. [119] | Adaptive `NN` architecture | Unclear | Custom `NN` | Up to 44.4% | (Up to -10%) |

[a] Results given for a Snapdragon 400 SoC.
[b] Results compared to a CPU implementation.
[c] Results compared to a native GPU acceleration.
[d] Only given as a reduction in model size which is unlikley to directly translate to speedup.
[e] Unclear whether top-1 or top-5 score.
[f] Presented as top-5 score.
[g] Results given for a a Snapdragon 810 SoC.

have not been proven to achieve state-of-the-art accuracy on complex problems or large networks. To make matters worse [31] shows that across thousands of experiments, complex techniques shown to yield high compression rates on smaller datasets perform inconsistently, and that simple magnitude pruning approaches achieve comparable or better results. Further exacerbating the problem, there is evidence that pruning may not always be the best approach to address hardware constraints. Yazdani et al. [150] shows that even though pruning may give correct test accuracy, the inference confidence score reduces significantly. Therefore, pruning (and similar methods) is not an acceptable approach for applications that depend on the confidence score of a prediction. As a consequence the usefulness of many `DNN` optimisation techniques are brought into question when applied to large datasets and newer, more accurate `DNN` architectures. Often faster `DNN` inference means lower accuracy, leading to a choice between high accuracy, or a smaller, cheaper model. Furthermore, making such a crucial decision is a non-trivial task as the application context (*e.g.* the model input) is often unpredictable and constantly evolving. The work presented in this thesis aims to remove this decision by automatically choosing the best model to use at runtime.

## 3.2 Efficient DNNs for Hardware

Recently, a number of software-based approaches have been proposed to accelerate `DNNs` on embedded devices. This section focusses on software optimisations that make efficient

use of available hardware resources for DNN execution; some hardware optimisations are also mentioned in this section. The approaches presented below are not mutually exclusive to those presented in Section 3.1; they can be used in conjunction. In order to frame current work Table 3.2 presents an overview of work in this area. These approaches aim to accelerate inference time by computational kernel optimization [35, 7], exploiting parameter tuning [70, 95], task parallelism [96, 67, 105], and trading precision for time [53, 29]. Each of these approaches are discussed below.

### 3.2.1 Computational Kernel Optimsation

The works in [35, 7] present methods to speed up computational kernels to reduce the resource requirements of DNNs. Here a computational kernel is a segment of computation that has been compiled for high throughput, usually for a device such as a GPU *e.g.* a convolutional filter. In [7] the authors present SparseSep, a compiler and runtime that is able to effectively utilise DNN sparsity through the use of codebooks to achieve on average 11.3x and 13.3x less memory and running time, respectively. By controlling the sparsity that is introduced during compilation SparseSep generates a codebook that can be combined with a sparse matrix to recreate a dense matrix which closely approximates the original. Unfortunately, due to the approximation, this method also results in an accuracy loss of around 5%. Han et al. [35] take a different approach, they present a custom hardware architecture named Efficient Inference Engine (EIE), also developed to make use of kernel sparsity. EIE makes use of the sparsity by being developed specifically for the task, additional logic is built into the device to keep track of where input and output values belong in the network. When EIE is used in conjunction with Deep Compression it is able to achieve 13x speed up over a GPU. However, both of these approaches have only been shown to work on AlexNet and VGG, which do not achieve state-of-the-art performance. AlexNet and VGG are relatively small DNNs compared to better performing DNNs such as ResNet-152, therefore the approaches may not necessarily extrapolate to large models [31].

### 3.2.2 Tuneable Parameters

CNNdroid [70] is an open source library for execution of trained convolutional neural networks on Android devices. AutoTVM [11] is a tuning compiler that is implemented into TVM [1] for hardware specific optimisations. These works also aim to optimise computational kernels, however they do not make use of sparsity, instead they tune how the kernel runs (*e.g.* what order is memory accessed, what is the best batch size etc.) to achieve speed-up. As they do not change the DNN at all, only how it is executed,

[1] https://github.com/apache/incubator-tvm

they do not impact accuracy. Both libraries make use of some base code with tunable parameters, which is accessed through a front-end library, to optimise software for the desired hardware. On compilation the library will enter a optimisation loop where it will profile, tune, and recompile the DNN until a threshold is met. While this approach often achieves huge speed-ups, seeing up to 60X speedup and up to 130X energy savings (when compared to CPU performance), it has some drawbacks. Unless an effective conversion tool is available, DNNs will need to be re-written and re-trained in the native library. The DNN programmer needs to make use of the specific library's front-end, which also requires an implementation for all components of the DNN. Furthermore, tuning can be a long process depending on the size of the DNN and the number of parameters that are being optimised.

### 3.2.3 Task Parallelism

A number of researchers have presented inference software as an approach to efficiently run DNNs inference on embedded devices. Lane et al. [67] note that the high resource cost of DNNs leads to bottlenecks and slowdowns when used on embedded devices. To combat this they present a pair of resource control algorithms that decompose monolithic network architectures into unit-blocks of various types – making it more manageable. They term this DeepX. The smaller unit-blocks can then be offloaded to other devices to improve inference time. Their method aims to reduce energy consumption and model size while meeting a runtime deadline such as 500ms and having minimal impact on accuracy (<-5%). Using DeepX on AlexNet they were able to achieve an average of 92.4% and 75.5% reduction in energy and memory, respectively, while losing 4.9% accuracy. Huynh et al. [53] present DeepMon, which is similar to DeepX. DeepMon presents a suite of optimisation techniques for offloading computation to a GPU, which can be tuned more specifically for the DNN and hardware constraints than DeepX. If all methods presented in DeepMon are utilised it can achieve a runtime of 269ms at a cost of 6% accuracy drop on AlexNet, nearly halving the runtime of DeepX. Furthermore, with careful tuning DeepMon can still outperform DeepX while only reducing accuracy by 1.6%. In a similar vein Rallapalli et al. [105] reveal that by offloading the DNN layers that require the most memory to the CPU they can reduce bottlenecks and increase throughput. In [96] the authors show that launching the maximum number of logical threads is often not the best approach for embedded systems, due to their unique architectures and resource limits. They implemented a regression model which allows an accurate prediction of the correct number of threads for each DNN with minimal profiling leading to a 57.8% reduction in application runtime, and up to 47.4% reduction in its energy efficiency. The methods presented above show that runtime speed-up can be achieved with careful optimisation

and computation mapping onto devices. However, the best gains are when accuracy is sacrificed which might not be an option, as discussed in section 3.1.

### 3.2.4 Accuracy-Runtime Trade-off

Fang et al. [29] present a framework which can adapt to the resources available at runtime; they name this NestDNN. They achieve this through the implementation of a Multi-Capacity Model that can grow and shrink depending on the resources currently available. While an effective method, it requires tuning for each new DNN and hardware. Furthermore, the filter ranking method they propose grows exponentially as the number of classes increases, therefore their approach has only been evaluated on a subset of Imagenet ILSVRC 2012. In a similar vein, Pervasive CNN [119] (P-CNN) generates multiple computation kernels for each layer of a CNN during compilation. At runtime P-CNN uses performance modelling to select the best kernels to use to meet the users accuracy or runtime requirements. While these methods are adaptable at runtime to a changing environment, the methods mean that the user has to choose between accuracy and speed. Furthermore, the initial implementation of these methods can be very time and computationally expensive, especially for larger more accurate models such as ResNet-152.

### 3.2.5 Summary

Since a single model is unlikely to meet all the constraints of accuracy, inference time and energy consumption across inputs [9, 34], it is attractive to have a strategy to dynamically select the appropriate model to use. The work in this thesis provides exactly such a capability and is thus complementary to these prior approaches.

## 3.3 Offloading DNN Computation to a Server

In some embedded environments a powerful external server is available for computation, however the use of an external server is not always feasible due to privacy, latency, or connectivity issues. This section focusses on work which attempts to combat these problems through choosing which work should be offloaded, or novel methods of privacy protection. Work that chooses when to offload work to a server will be discussed first, followed by work that combats privacy concerns.

When an external server is available the simplest solution is to always offload DNN computation to it. However, it is not always the best solution, communication across a network is inherently unreliable unless tightly managed – which is often not the case. Due to the network, the latency of sending data to a server and receiving a result could

be longer than doing the computation on the device [28]. Neurosurgeon [58] presents a solution to overcome this problem by identifying when it is suitable to offload layers to a server. Neurosurgeon successfully identifies the best layers to offload with the use of regression models to predict layer runtime and energy consumption. The approach is able to reduce end-to-end latency (runtime) by 67.74%,and energy consumption by 59.5%. In [131] the authors present Distributed Deep Neural Networks (DDNN). DDNN is designed to be a framework that can spread DNN computation across cloud, fog, and end devices, allowing a combination of fast and localized inference on edge devices and complex inference in cloud servers. Each device that the DNN is spread across is able to generate an intermediate prediction that can be used in two ways. If the intermediate prediction is of high confidence, then that prediction is the output and no further computation is needed. If the intermediate prediction is not of high confidence, it is fed into the next device's DNN partition and computation continues. This method reduces communication as data is only sent when it is needed, and when data is sent across the network it is in a much smaller format – the intermediate output is much smaller than the original DNN input. They show that through this method the communication overhead could be reduced by 20x due to the changing data as it passes through a DNN. Furthermore, they claim to help with privacy issues as the raw data is partially processed on-device before transmission. Both works presented above use layer level partitioning; an interesting direction that the community has yet to explore may be to use finer-grained partitioning. For example, partitioning a DNN at the computational kernel level could lead to even more obscurity in communication, and improvements in computational efficiency by only offloading the kernels the edge device struggles to execute quickly.

The work presented by Ossia et al. [99] partially addresses the issue of privacy-preserving when offloading DNN inference to the cloud. Similar to DDNN the authors present an approach which performs part of the inference on-device before transmitting the partially processed data. They note that some information could be extracted from the partially processed data and therefore present methods to obscure it. Through influencing the training of the model with Siamese fine-tuning and adding some noise before transmission they successfully obscure the data at the cost of accuracy (<5%). Alternatively, [116] suggests training two separate models, a shared, and a personal. The shared model is trained on a server on non-private data. The personal model is a fine-tuned version of the shared model using the personal data on the device, all training is carried out on-device. Their approach shows an increase in accuracy after fine-tuning without transmitting private data. However, the approach requires training, a resource intensive task, to be carried out on an embedded device, a resource limited device; the cost of this is not given in the paper.

The work presented in this thesis aims to make on-device inference feasible for embedded devices, and therefore avoiding the issues around latency, connectivity, and privacy.

## 3.4 Ensemble Learning

This section investigates recent approaches to DNN ensemble models. Previous sections have focussed on techniques designed to improve performance of DNNs, potentially at the cost of accuracy. DNN ensemble models aim to improve accuracy without caring about performance. A DNN ensemble model is when multiple DNNs are used to make a prediction on a single input, the predictions from each model are then combined by some algorithm to produce a single output. In this section the collection of DNNs into one model will be referred to as an ensemble model, and the DNNs which make up the ensemble model are referred to as component models or component DNNs. Ensemble models can be thought of in two different ways: horizontally stacked, where the output of one component model feeds into the next; or vertically stacked, where multiple component models exist 'side by side' and do not interact until their outputs are combined. Below a brief overview of some recent work in the area is presented.

Kontschieder et al. [64] present a horizontally stacked ensemble model they term Deep Neural Decision Forests. They combine a CNN with a decision forest, feeding the output from the CNN's fully connected layer into the decision forest which produces the final output. To facilitate end-to-end training the authors introduce decision trees (which form the decision forest) capable of stochastic back-propagation. Their results show a slight improvement over Inception on the ILSVRC 2012 dataset, improving top-5 accuracy by 4% depending on the number of crops used. Zhou et al. [156] take this approach further, they replace the neurons in a DNN with random forest models; they call this gcForest. This approach achieves competitive performance with DNNs, while requiring less training and has less hyper-parameters to tune. However, the results of gcForest are only presented on smaller datasets such as MNIST. Furthermore, while they claim their approach is highly parallel, their approach is almost 4x slower than a DNN running on a GPU in their example.

Work surrounding vertically stacked DNN ensembles often investigate how to effectively combine the component model outputs to produce the most accurate final ensemble output. Using this approach some guarantee can be given about the confidence of a prediction, for example if all models in the ensemble predict the same output then there is a high confidence that is correct. Two examples of vertically stacked DNNs are shown in [104, 144] Wen et al. [144] present an ensemble of CNNs for facial recognition. Their approach trains a number of different CNNs, all with the same architecture, on the same dataset to produce a number of candidate component models. A simple selection method

is then used to select the *n* best performing component models, without taking diversity into account – which is shown to be a useful consideration in ensembles [86] The output of each component model is combined using probability-based fusion. Their results show an improvement in accuracy by up to 5% on a number of different datasets. Similarly Qui et al. [104] presented an ensemble of DNNs for regression and time series forecasting. In this work, the authors use a single DNN frozen at different stages of training to produce 20 different component models; again not taking diversity into account. The output of each component model is then fed into a Support Vector Regressor (SVR) which produces the final output. When evaluating the approach on a number of different datasets it is show to outperform other methods in the area. Stahlberg et al. [122] present an approach based on minimum bayes risk to combine two models just before the decoding step where the sequence of words is generated. This work was then taken further in [121] to extend the approach to combine any number of models. The authors compare their ensemble model against each component model working alone and show that their approach improves the BLEU score by around 1 or 2 ( 5% increase) in a number of different WMT newstest datasets.

The work presented in this thesis aims to make use of the concept behind ensemble models – different DNNs are good at different tasks, even in the same dataset. This thesis aims to enhance inference speed without losing accuracy by using a predictive ensemble approach. Ensemble approaches often increase inference time as they need to execute multiple DNNs, the work in this thesis pre-learns the best component DNN to use on a particular input, therefore only executing one component DNN. This approach effectively combines multiple DNNs, bringing the strengths of all approaches, without the weaknesses.

## 3.5 Improving DNN Training

There are two main phases for DNNs: training, and inference. Previous sections have focusses on approaches that aim to improve inference accuracy, or inference runtime; this section investigates approaches to improve DNN training. The largest costs when creating a DNN are usually the data collection phase, as huge amounts of data are needed to create an effective model, and the training phase, due to iterating over the training data multiple times. This section focusses on recent research which attempts to improve the training of a DNN through speeding up training or developing methods that require less data to train effectively. As training is not a focus of the work in this thesis only a brief overview of a few works in this area are given below. This section discusses the training of sparse networks, followed by transfer learning.

Some recent work [22, 6] tries to improve training by directly training a sparse model. These approaches aim to achieve the same end goal as those discussed in section 3.1

without the need to start with a pre-trained DNN. Bellec et al. [6] introduce Deep-R, a method of training a sparse network while achieving the same performance as a dense DNN. Their approach involves 'rewiring' the network during training so the most needed connections are available when they are needed, meanwhile restricting the total number of active connections. Deep-R chooses the connections that should be active using a Bayesian approach that probabilistically chooses which connections will be needed during the upcoming iteration. The approach is compared to training a dense network from scratch and they show that even with a restriction of 1.3% of connections active they approach achieves very similar accuracy. Unfortunately, the method is only shown to work for MNIST and CIFAR-10 using a simple CNN. Deep-R is computationally expensive and challenging to apply to large networks and datasets. A less computationally expensive approach is presented in [22]. To achieve similar performance to a dense DNN the authors present an algorithm they term Sparse Momentum. Sparse Momentum uses exponentially smoothed gradients (momentum) to identify layers and weights which reduce the error. The authors show that their approach can achieve on par performance with other comparative methods, such as Deep-R, on a subset of the ImageNet ILSVRC dataset. The actual speed-up of this approach is not given and is estimated by proxy of the reduction of FLOPS which can obscure the actual speed-up value. Furthermore, the FLOPS reduction on the subset of ImageNet is not given.

An alternate approach to improving DNN training is transfer learning, which aims to distil some information from a larger teacher model to help a smaller student model. This teacher-student dynamic is designed so that the student model requires less training data to generate an effective DNN. Zagoruyko et al. [152] introduce a method they term Attention Transfer (AT) to facilitate transfer learning. The attention that they develop is designed to summarise the convolutional filter activations into a state that allows transfer between models, even when they are different sizes. They investigate using the filter activations directly, or their gradients to create and effect transfer method. For Imagenet they determine that activation-based attention is the best method, and they are able to achieve 1.1% top-1 and 0.8% top-5 better validation accuracy over training from scratch. However, to evaluate AT they use small versions of ResNet (ResNet-34 as a teacher, and ResNet-18 as a student), and a subset of ImageNet; it's unclear whether the same benefits carry over to larger models and datasets.

The work in this thesis does not effect training times of the DNNs, however these methods could be used to enhance the methods that are presented. One approach could be to train a number of different models from scratch using the methods presented above instead of relying only on pre-trained DNNs.

## 3.6 Applications of Machine Learning

Machine learning has been employed for various optimization tasks. It has proven its ability to be adaptable to evolving environments, making it particularly useful. This thesis uses machine learning techniques to optimise DNNs, both statically and dynamically. This section looks at the different ways that machine learning has been employed to solve a wide array of problems. First, techniques which utilise ML for static optimisation (optimisation carried out before runtime *e.g.* compilation) are discussed, followed by dynamic optimisation (optimisation carried out at runtime, *e.g.* resource management).

Using machine learning in compiler optimisation has become commmonplace [141], it provides the utility for dynamic and adaptable compilers. Wang et al. [139] present a profile-driven approach for parallelism detection for compilers. Their approach is based on the Intermediate Representation (IR) of the input program, which they instrument and execute to generate a Control and Data Flow Graph (CDFG). The CDFG is then used to inform an SVM classifier that chooses whether or not to parallelise a loop candidate and how it should be scheduled. This approach achieves, on average, 96% of the performance of the hand-tuned benchmarks. However, this approach relies on the user for final approval of the prediction due to the uncertainty of the approach. Furthermore, this approach relies on dynamic profiling of the code, introducing further overheads (up to up to 100x slowdown) during compilation. In [129] the authors present an adaptive method for OpenCL kernel mapping. An OpenCL kernel is a segment of computation that has been compiled for high throughput and can easily be mapped to a number of different processing devices such as a CPU or GPU. Similar to above, this approach extracts features directly from the IR, however this approach does not use dynamic profiling. The extracted features are fed into an SVM classifier which chooses which device to run the kernel, and the frequency the device should run at. Using this approach the authors achieve 92.6%, 91.4% of the Oracle (here, the Oracle is the best possible achievable results) performance and energy consumption, respectively. While this approach avoids the overheads introduced by dynamic profiling, it is yet to be seen how adaptable this approach would be to more complex loops, and varying loop sizes. These approaches show that ML techniques are an effective tool for creating compilers (static optimisation) that can adapt to new software and hardware combinations, similar to work presented in section 3.2.

ML techniques have also been proven to work in dynamic environments, namely cloud resource management. Such an approach is presented by Delimitro et al. [19] where they introduce Quasar. Quasar is a scheduling framework that estimates the scale-out and scale-up factor for jobs using collaborative filtering based on profiling information from the first few tasks. Using this method Quasar improves resource utilization by 47% in a 200-server EC2 cluster while meeting performance constraints. Quasar is

Table 3.3 A summary of how the work in this thesis fits in with some of the key works already discussed.

| | Adaptive | Automatic Network Tuning | Large Dataset | On device | Conserves Accuracy |
|---|---|---|---|---|---|
| **NestDNN** [29] | ✔ | ✔ | | ✔ | |
| **Deep Compression** [36] | | | ✔ | ✔ | ✔ |
| **ThiNet** [84] | | ✔ | ✔ | ✔ | |
| **Molchanov et al.** [94] | | ✔ | ✔ | ✔ | |
| **Wu et al.** [147] | | ✔ | | ✔ | ✔ |
| **Neurosurgeon** [58] | | ✔ | ✔ | | ✔ |
| **This Thesis** | ✔ | ✔ | ✔ | ✔ | ✔ |

one of the first works that present a performance-centric approach to cloud processing jobs. In a similar vein, [137] presents Ernest, a performance prediction framework for cloud servers. Ernest is designed to predict the computation and communication of tasks as they run, this information is then used to effectively allocate resources. As opposed to Quasar which makes a prediction at the beginning of task execution, Ernest is designed with a low overhead to always be running and adapting to the task workload as it runs. Furthermore, to reduce overhead Ernest employs an optimal experiment design strategy to create a predictor able to make accurate predictions quickly. Ernest is able to reduce resource consumption by 75%, however this number is based on virtual machine instances.

The work presented above shows that ML can be an effective tool for dynamically choosing the best hardware or software configurations to apply in a changing environment. The work in this thesis takes the same principles and applies it to DNN optimisation; this is not commonplace in DNN optimisation techniques. Instead of choosing the best hardware or software configuration to use, this thesis focusses on choosing the best DNN for the task at hand.

## 3.7 Discussion and Conclusion

This chapter has introduced the basic concepts of DNNs and SML that are used in this thesis. Furthermore this chapter discusses the recent work in the area and shows how the work in this thesis fits into the current state-of-the-art. This section provides further discussion of the presented works, and how they relate to the work in this thesis at a higher level.

Table 3.3 provides a summary of how this thesis relates to similar work in the area. The works presented in Table 3.3 have been chosen due to their similarities to the work in this thesis; each work was chosen to represent different aspects of the related work presented above. For clarity, a brief description of each heading is given:

- **Adaptive.** Is the work able to adapt to a changing environment? Often the best DNN changes depending on the data or resources available.
- **Automatic Network Tuning.** Does the work automatically tune and optimise the network? When this process is not automatic it relies on human input to choose the best strategy during DNN compression.
- **Large Dataset.** Has the work been proven effective on a large dataset such as ImageNet? Some work has been proven on toy datasets, or subsets of ImageNet. However, the results do not necessarily carry over to larger models and datasets [31]. Furthermore, this can indicate that the method is not scalable, such as NestDNN.
- **On Device.** Is all of the computation carried out on-device? Offloading computation to an external server is not always a viable option.
- **Conserves Accuracy.** Does the approach conserve the accuracy of the base DNN before compression? The most effective compression methods often come at the cost of accuracy.

Table 3.3 shows that this thesis is unique. Most work in the area is not adaptive, which leaves a huge amount of potential efficiency untapped. NestDNN is arguable the closest competitor to this thesis, however the methods presented are incredibly computationally expensive and are unable to scale to large datasets and large models. Furthermore, NestDNN does not try to conserve accuracy. This thesis takes a novel approach to efficient DNN inference, surpassing similar work in the area. Some work such as that in Section 3.2 is not included in this table as it is not comparable; it would be better utilised in conjunction to the work in this thesis, leading to further improvements DNN efficiency. A deeper discussion is presented on those works below.

**Performance-Accuracy Trade-off.** Sections 3.1 and 3.2 discuss a number of methods for improving the inference speed of DNNs, however these methods are often at the cost of accuracy. To make matters worse, Yazdani et al. [150] show that even though these methods often have no significant impact on scoring metrics such as top-1 and top-5, the inference confidence score reduces significantly. Therefore, these are not acceptable approaches for applications that depend on the confidence score of a prediction. The work in this thesis aims to circumvent this issue by allowing more flexibility in the choice of DNN at runtime, furthermore the presented approaches can be adapted to a different optimisation metric such as prediction confidence.

**Computation Offloading.** Section 3.2 investigates some recent methods that decide when it is beneficial to offload work to other processors on device. Section 3.3 investigates the similar problem of offloading computation to a server; this problem has an

additional caveat of network communication and latency. An interesting direction that the community has yet to explore may be to combine these two research areas into one. This could result in a framework which is able to identify when it is beneficial to offload computation, given the current state of the environment (which processors are free, or best for the task at hand). The work in this thesis takes a different approach to inference speed-up. Instead of focussing on the device to run the DNN, this work presents methods to choose the best DNN to use when running in an embedded environment. Moreover, computational offloading could be used in conjunction with this work to provide even more gains in inference speed.

**Ensemble Models.** The work in this thesis is closely related to ensemble learning (section 3.4) which is shown to be useful for scheduling parallel tasks [26] and optimising application memory usage [88]. This work applies a similar technique to optimise DNN inference on embedded devices. However, this work differs from conventional ensemble approaches; it only executes a single component model during inference instead of them all. To facilitate this, the work uses a pre-classifier to choose the best DNN, turning the conventional approach of ensemble models on its head. By doing this, the work is able to see the accuracy gains of ensemble models without the added computational overheads. An avenue of research that could be of interest, but is not explored in this thesis could be a combination of the work in sections 3.4 and 3.5, resulting in fast training, diverse ensemble models.

# Chapter 4

# Approach

This chapter presents the methodology and justification for the design choices of the work in this thesis. First, the central thesis is presented: that SML can be effectively utilised to reduce DNN inference costs, at little or no impact to accuracy; this section will focus on the overarching idea of the whole thesis. The central thesis is then broken down into an additional two sections, each focussing on individual, yet complimentary components of the central thesis.

## 4.1 Overview

This thesis presents a novel approach to DNN inference optimisation. Typical approaches to DNN optimisation focus solely on how to improve inference times or reduce model sizes. Whether that is through tuning the network directly (Section 3.1), tuning the underlying code for the hardware (Section 3.2), or choosing when to offload computation (Section 3.3). However, the downfall of these approaches lie in their assumption that one size fits all, that is, that one DNN is suitable for all inputs; this leaves much optimisation potential – for both accuracy improvement and inference time reduction – on the table. Some work, such as that in Section 3.4, focus on improving accuracy; however, it comes at a computational cost. This thesis presents an alternate approach to DNN optimisation, reducing inference time and improving accuracy, without the added computational cost. Instead, the presented approach utilises more memory space than typical approaches in order to achieve its benefits The next sections describe the overall motivation for this thesis, beginning with the initial motivation. A natural progression of the initial motivation is then described, exploiting the opportunities of specialising DNNs; this is explorative research, showing the potential for future research. To finish, the key points and ideas are summarised.

(a) Image 1 (b) Image 2 (c) Image 3

Fig. 4.1 The three images used on four CNN-based image recognition models in Section 4.1.1. The target object is highlighted in each image.

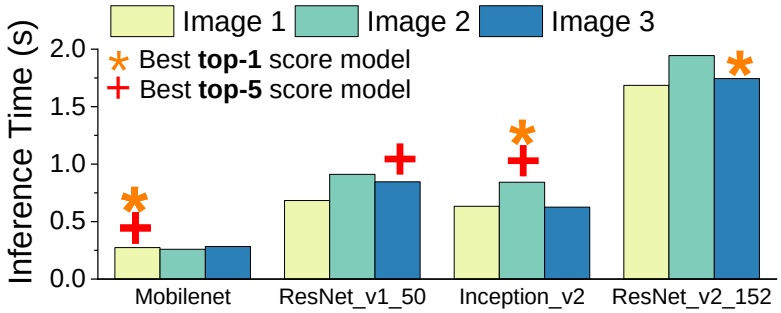



Fig. 4.2 The inference time of four CNN-based image recognition models when processing the images in Figure 4.1. This example (combined with Table 4.1) shows that the optimal model (*i.e.* the fastest one that gives an accurate output) depends on the success criterion and the input.

### 4.1.1 Initial Motivation

As a motivating example, consider performing an image classification task on an NVIDIA Jetson TX2 platform (described in detail in Section 5.1.1).

**Setup.** This experiment compares the performance of three influential CNN architectures: Inception [125], ResNet [43], and MobileNet [50]; described further in Section 5.1.2. Specifically, the following CNNs were used:

- `Inception_v2`. The second version of the Inception architecture.
- `ResNet_v1_50`. The first version of the ResNet architecture, with 50 layers.
- `ResNet_v2_152`. The second version of the ResNet architecture, with 152 layers.
- `MobileNet_v1_025`. The first version of the MobileNet architecture with a width multiplier of 0.25.

All models are built upon TensorFlow [1] and have been pre-trained by independent researchers using the ImageNet ILSVRC 2012 *training* dataset [113]. The GPU on the Jetson TX2 was used for inference.

Table 4.1 A list of CNNs that give the correct prediction per image under the top-1 and the top-5 scores. The optimal CNN for each image under each scoring metric is marked in italics.

| Scoring Metric | Image 1 | Image 2 | Image 3 |
|---|---|---|---|
| top-1 score | *MobileNet_v1_025*, ResNet_v1_50, Inception_v2, ResNet_v2_152 | *Inception_v2*, ResNet_v2_152 | *ResNet_v2_152* |
| top-5 score | *MobileNet_v1_025*, ResNet_v1_50, Inception_v2, ResNet_v2_152 | *Inception_v2*, ResNet_v1_50, ResNet_v2_152 | *ResNet_v1_50*, ResNet_v2_152 |

**Evaluation Criteria.** As input, each CNN takes an image, in return a list of label confidence values are given as output. Each value indicates the confidence that a particular object is in the image. The resulting list of object values are sorted in descending order regarding their prediction confidence; the label with the highest confidence appears at the top of the list. In this example, the accuracy of a model is evaluated using the top-1 and top-5 scores defined by the ImageNet Challenge. Specifically, the top-1 score, checks if the top output label matches the ground truth label of the image; the top-5 score checks if the ground truth label of the image is in the top 5 of the output labels. The scoring is done on a per-model, per-image basis.

**Results.** Figure 4.2 shows the inference time per CNN of each image in Figure 4.1; the images are from the ImageNet ILSVRC *validation* dataset. Recognising the main object (a cottontail rabbit) in the image shown in Figure 4.1a is a straightforward task; the main object is central within the image, and contrasts with the background. Table 4.1 shows that all CNNs were able to provide a correct answer under the top-1 and top-5 scoring criterion. For this image, MobileNet_v1_025 is the best model to use under both scoring criterion, it has the fastest inference time – 6.13x faster than ResNet_v2_152. Clearly, for this image, MobileNet_v1_025 is good enough. There is no need to use a more complex (and computationally expensive) model for inference. Now consider a slightly more complex image classification task using the image shown in Figure 4.1b. MobileNet_v1_025 is unable to give a correct answer regardless of the success criterion. For this image, Inception_v2 should be used, although it is 3.24x slower than MobileNet_v1_025. Finally, consider the image shown in Figure 4.1c, intuitively it is a more difficult task. The main object is a similar colour to the background, and rotated at an usual angle. In this case the optimal model changes depending on our success criterion. ResNet_v1_50 is the best model to use under top-5 scoring, executing inference 2.06x faster than ResNet_v2_152. However, if top-1 is used for evaluation then ResNet_v2_152 must be used, it is the only model capable of obtaining the correct answer despite being

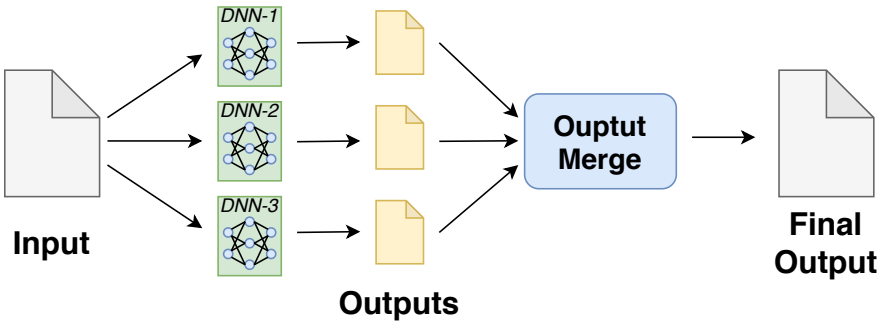



Fig. 4.3 An example of how an ensemble of DNNs works, using model stacking. The input is passed to each component DNN, which will produce its own output. The outputs are then combined to produce the final output. Section 3.4 explains ensembles in depth.

the most expensive. Inference time for this image is 6.14x and 2.98x slower than MobileNet_v1_025 for top-1 and top-5 scoring respectively. The results are similar if we use different images of similar complexity levels.

**Conclusion.** This example shows that the best model depends on the input and the evaluation criterion. Furthermore, determining which model to use is non-trivial. A 'Model Selector' is needed, a technique that is capable of choosing the best DNN to use at runtime, depending on the individual input, and the available DNNs. It is worth noting that the proposed solution is significantly different to an ensemble of stacked DNNs, and is significantly more efficient. Figures 4.3 and 4.4 show how the Model Selector is implemented differently to a stacked ensemble. Figure 4.3 shows an ensemble of stacked DNNs, and Figure 4.4 shows a comparative setup with a Model Selector; using an approach known as a *bucket of models*, also known a *mixture of experts*. Comparing the two Figures it becomes clear that if an implemented Model Selector requires less resources than the two DNNs it prevents running, then compute resources can be saved overall. Furthermore, if each input is considered individually, and only one DNN is run per input, the benefits are twofold: (i) sometimes a cheaper DNN can be selected to improve accuracy, and (ii) the cheapest accurate DNN can be selected to reduce inference time. Section 4.2 describes a Model Selector in detail.

### 4.1.2 A Natural Progression

The previous section clearly shows that different DNN models are capable of handling different inputs; if the optimal DNN can be chosen for each input at runtime then inference times can be reduced, and accuracy increased. It is well established that different parts of a DNN learn and utilise different features of the input which are used to larger or smaller degrees in classifying particular inputs [98, 103]. This can explain why less complex DNNs are able to classify inputs that more complex DNNs cannot, they are learning different features, *i.e.* MobileNet_v1_025 has learned some features that ResNet_v2_152 has not,

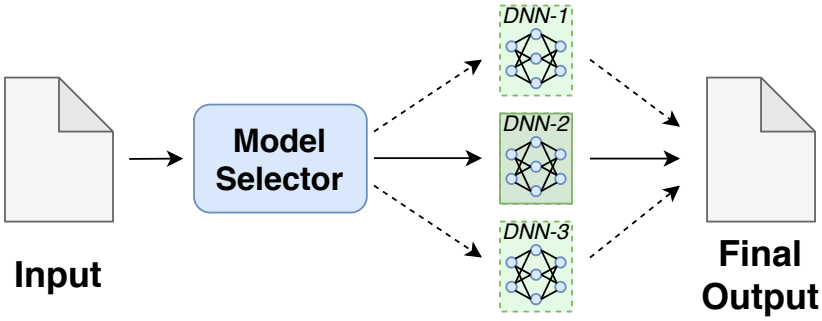



Fig. 4.4 A look at how a Model Selector would work in place of an ensemble of stacked DNNs; using a *bucket of models* approach. The dashed lines and boxes represent possible paths that the input could have taken; solid lines and boxes represent the actual path taken. In this example, the Model Selector chose to select DNN-2 for the given input. DNN-1, and DNN-3 will not run in this example.

meaning it is able to classify different images correctly. Furthermore, as ResNet_v2_152 is a larger and more complex DNN it learns more features, which contributes to its higher overall accuracy; it has more opportunities to learn a diverse set of features. Therefore, the natural progression to a Model Selector would mean breaking down a DNN such as ResNet_v2_152 into sub-DNNs that are each specialised for its 'data segment' (subset of the input data), while removing the features of the network that are important to other data segments; specialising each sub-DNN. In this scenario, each sub-DNN would be faster than the original DNN, as the problem it is solving is much simpler so more pruning will be possible. This work exploits the opportunities of specialising DNNs, it is explorative work showing the potential for future research. As a motivating example, consider performing an image classification using ResNet_v2_152.

**Setup.** This experiment compares the convolutional filter activations caused by different images when using ResNet_v2_50 for inference. A pre-trained version of ResNet_v2_50 was used, built upon TensorFlow [1], and trained by independent researchers using the ImageNet ILSVRC 2012 *training* dataset [113]. All 50k images from the ImageNet ILSVRC 2012 *validation* dataset were inferred on ResNet_v2_50, and the activation of every convolutional filter for each image was recorded; the L2-norm of each filter is calculated to produce a single value per filter, per image. The L2-norm is recorded as this is a common method used for ranking neurons and filters for removal [45, 110]. Using this data, two averages are calculated:

- **Overall Layer Average Activation.** (OLAA) Calculated per layer in the model, it is the average L2-norm of all image activations across all filters in each layer. This provides a baseline average activation value for each layer in the DNN.

- **Overall Filter Average Activation.** (OFAA) Calculated per filter, it is the average L2-norm of all image activations for each filter. This provides a baseline average activation value for each filter in the DNN.

**Evaluation Criteria.** Next, the importance of every filter for an image needs to be determined. A simple approach would be marking a filter as important if an image causes an activation higher than the average activation. However, this proved to be unreliable during pilot experiments; typically an image would determine all filters to be important or none, indicating that some images produce higher activations on average. Furthermore, some layers in the DNN appear to naturally have higher activations on average than others, therefore the filter activations need to be brought into the same range. In order to achieve this, a *layer penalty* was used, calculated using the following metrics:

- **Image Layer Average Activation.** (ILAA) Similar to OLAA, however calculated on a single image. Used to normalise filter values within each layer.
- **Image Filter Activation.** (IFA) Similar to OLFA, however this is simply the L2-norm of an image's activation for each filter. If this is higher than the model average for the same filter (OFAA), then this filter is important for an image.
- **Image Layer Penalty.** (ILP) Calculated per layer, per image. It is calculated as $OLAA - ILAA$. Used to normalise an image's filter activations within each layer, therefore preventing images that have high activations on average from determining that every filter as important.

Finally, each filter was determined to be important if the following held true:

$$IFA - ILP > OFAA \tag{4.1}$$

In essence, this equation checks whether an image causes a higher than average activation for every single filter, while accounting for the layer variation per image (ILP). If an image causes a higher than average activation for a filter, then it is considered important for this image. Using this measure of filter importance, each image determined that approximately 40% of all filters are important.

**Results.** Tables 4.2, and 4.3 analyse the important filters for each of the three images shown in Figure 4.2; the images are from the ImageNet ILSVRC *validation* dataset. Here an overlapping filter is one that an image shares with one of the other two, whereas a unique filter is not shared at all. Table 4.2 compares the important filters between each of the chosen images. Between the images there is a high overlap of important filters, this is to be expected. Early layers of a DNN learn generic features that are applicable

Table 4.2 The percentage of filters that are important for each image (shown in Figure 4.2) when using `ResNet_v2_50` for inference. The method for determining importance is defined in Section 4.1.2.

| Image | Total | Unique | Overlapping |
|---|---|---|---|
| Image 1 | 40.64% | 5.90% | 34.74% |
| Image 2 | 40.45% | 5.22% | 35.23% |
| Image 3 | 40.82% | 5.96% | 34.86% |

Table 4.3 The percentage of filters that are not important for each image (shown in Figure 4.2) when using `ResNet_v2_50` for inference. Here, and unimportant filter is every filter that is not determined to be important.

| Image | Total | Unique | Overlapping |
|---|---|---|---|
| Image 1 | 59.36% | 4.96% | 54.40% |
| Image 2 | 59.55% | 4.47% | 55.08% |
| Image 3 | 59.18% | 4.84% | 54.34% |

to most inputs, the later layers then build upon the generic features in order to create more specialised features [98, 103]. Furthermore, each image determines that 5-6% of all filters are important to only this image and not the other two. Therefore, if a `DNN` was no longer required to classify image 1 an extra 5-6% of filters could be removed from the `DNN` to compress it, reducing end-to-end inference time. A compounding effect with diminishing returns would be expected as more images are removed from the classification pool; the level of the compounding effect is one aspect that is evaluated in this thesis. Table 4.3 compares the unimportant filters between each of the chosen images. Here, a high overlap is to be expected, and is the reason `DNN` pruning works - most inputs agree that many of the filters are unimportant and therefore can be removed.

**Conclusion.** This example shows that the most efficient `DNN` depends on the input image - not every input uses all of the `DNN` the same amount. If inputs could be grouped so that all inputs within a group agree on which filters are important, and all groups disagree on which filters are important then each group could have a specialised `DNN` created for it. Each specialised `DNN` would be capable of classifying all images it has been trained for while needing less filters, and therefore performing inference faster. For this method to work, it would rely on a Model Selector to be trained which can determine the best specialised `DNN` to use, at runtime, for each input. Furthermore, if only one pre-trained `DNN` is available, it could be cloned and pruned numerous times (once for each group, creating numerous sub-`DNN`s) to reduce the overall inference time more than typical `DNN` pruning is capable of, without impacting accuracy. In order to achieve this a method needs to be developed to split the `DNN` training data into closely related subsets that agree on important filters - a non-trivial task. If this idea was taken to an extreme, *e.g.* each

data segment would being a single image, the Model Selector would be doing the actual classification, replacing the DNN entirely. Clearly, this would result in low accuracy as SML is not as effective at complex classification problems as DNNs are. Therefore, an integral part of splitting the data into data segments is finding the right balance of data segments to make. More data segments will lead to more specialised sub-DNNs, and allow for further pruning; however, this will also lead to a more complex classification problem for the SML based Model Selector, a task that may be too complex for the SML model to solve.

### 4.1.3 Summary

Figure 4.5 shows how the two concepts explained above interact, while also being two individual problems. The following paragraphs will describe each component in more detail.

**Model Selector.** On its own the Model Selector, termed premodel, aims to overcome some of the downsides to using an ensemble approach to DNN inference optimisation. Given a pool of available DNNs and incoming data, the premodel is tasked with choosing the 'optimal' DNN for each input, a non-trivial task. In this context, the optimal DNN is the one that is able to achieve the highest accuracy in the lowest time. Initially the premodel is designed to select between off-the-shelf DNNs; when integrated into the next work it is choosing between bespoke, specialised DNNs. Inspiration for the model selector came from classical work in image classification, before DNNs were an effective solution to the problem [40, 134]. The older work shows how simple SML models, such as SVMs, can be effective at more coarse grained image classification problems. This component is described in more detail in section 4.2.

**DNN Model Specialisation.** Some level of model specialisation has already been shown to work in recent literature [37, 94]. Pruning and quantisation are good examples, they aim to remove unnecessary computation from a DNN, making it smaller without reducing accuracy. However, in general, these approaches are applied to a single model and a whole dataset. The aim of this work is to find the best way to separate a dataset into 'data segments' and tailor a DNN to each segment of the data. Once successful, it can be combined with a model selector to improve accuracy and efficiency. This component is described in more detail in section 4.3.

This thesis aims to be an alternate approach to DNN inference optimisation, complimentary to optimisation techniques that do not utilise pruning, *e.g.* work discussed in Sections 3.2 and 3.5. The following two sections describe the design and implementation of each of the components introduced above in more detail.

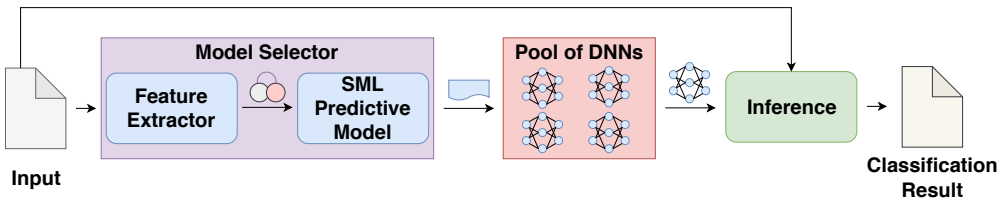



Fig. 4.5 An overview of how the two main pieces of work in this thesis interact. Each component has work directly relating to it. Section 4.2 focusses on the Model Selector, and Section 4.3 focusses on tailoring DNNs for the DNN Pool.

## 4.2 Model Selector - Design and Implementation

This section focusses on the work that has been published in LCTES '18 [130]; showing an effective implementation of a Model Selector. The following subsections provide an overview of the design and implementation of this work.

### 4.2.1 Overview

Figure 4.5 provides an overview of how the model selector (termed premodel) interacts with the rest of the inference process. Note that in this Section, the DNN pool will be made out of off-the-shelf DNNs. There are two paths an input will take, the first is the same as a normal inference process, and the second chooses the best DNN based on the input. The premodel is based on a set of quantifiable properties – or *features*, such as the number of edges and brightness of an image – of the input; extracted by the *feature extractor*. A set of candidate features need to be supplied that are narrowed down through an automatic feature selection process, described in detail in Section 4.2.4. Next, the premodel will carry out *model selection* via its internal algorithm (described in more detail in section 4.2.2) resulting in a DNN choice. Finally, the input is passed to the chosen DNN which will provide the classification result of the entire process. The premodel has been designed to work in exactly the same way as a normal, single model inference process *i.e.* the input and output (classification result) will be in the same format. Accurate model selection will result in the correct classification result being given while using the cheapest possible DNN, improving overall efficiency.

The premodel is automatically generated and trained depending on the problem domain, however requires some human input to begin. The user needs to supply: a set of candidate features, as mentioned in Section 4.2.4, these could be automatically generated [71, 13]; a set of pre-trained candidate DNNs, usually easily accessible, but depends on the problem domain; and an evaluation method for the DNNs, allowing the technique to quantify the accuracy of each DNN. By making the process automatic the best premodel architecture and SML classifier(s) can be chosen to create a fast and accurate premodel. Making this process automatic allows the premodel to adapt

to different scoring methods, *e.g.*top-1 accuracy or F1-score; a different DNN may be selected depending on the scoring method. As input, the premodel takes the same input as the problem domain DNNs, from which it will extract features and make a prediction, outputting a label referring to the best DNN to use for the current task. While the overall accuracy and runtime is improved by using this method, there may be some individual cases when the accuracy is lower, or the runtime is higher than optimal for a given input. This would be due to an incorrect prediction by the premodel, or the DNN selection algorithm (described in Section 4.2.3) choosing not to include a DNN which is vital to accurate predictions for that input.

**Limitations.** The success of this approach lies in the assumption that the average execution time of the chosen DNN, plus the cost of the premodel overhead is less than the cost of the highest costing individual DNN; formalised in the following equation:

$$p + \sum_{i=1}^{n} o_i m_i \; < \; max(m_1 + m_2 + ... + m_n) \tag{4.2}$$

where $p$ is the cost of running the premodel, $m_i$ is the cost of running DNN $i$, and $o_i$ is the proportion of the input data that considers model $i$ optimal. The sum of all $o_i$ is equal to 1. Note that this equation only takes cost into account, and not accuracy.

**Limitations Example.** First, consider the scenario laid out in Table 4.4. In order to produce this data the three selected DNNs ran inference on every image in the ImageNet ILSVRC 2012 *validation* dataset. The average inference time of each DNN was recorded, and the optimal DNN for each image was calculated; the column marked 'Proportion of Input' represents the proportion of the entire dataset that considers that DNN optimal under top-1 scoring. For this example, the worse case (with respect to inference time) DNN was used for the images that no DNN could correctly classify; in this case that is ResNet_v2_152. In this example, a premodel would be required to choose a DNN quicker than 1129ms; anything quicker than that would result in inference time savings on average. Now consider a second scenario using the same data, but with MobileNet_v1_100 removed, shown in Table 4.5. In this case, a much higher proportion of the dataset considers Inception_v4 to be optimal, a much slower model. A premodel is required to execute faster than 154ms in order to break even, leaving little room for improvement. The two given examples indicate that the most effective approach should involve both quick DNNs, and slow but accurate DNNs.

The following sections provide an in depth description of each component of the model selection process. The sections will cover the following parts of premodel creation: design, selecting the best DNNs to use, feature selection, premodel training, and premodel deployment.

Table 4.4 An example use case of the Model Selector. Using equation 4.2: $p + 742.86 < 1872.03$. Therefore the `premodel` must be quicker than 1129.17ms in order to be effective in this case.

| Model Name | i | Proportion of Input(o) | DNN runtime (m)(ms) | o*m |
|---|---|---|---|---|
| MobileNet_v1_100 | 1 | 0.72 | 336.71 | 242.43 |
| Inception_v4 | 2 | 0.12 | 1674.27 | 200.91 |
| ResNet_v2_152 | 3 | 0.16 | 1872.03 | 299.52 |
| Totals | – | 1.0 | – | 742.86 |

Table 4.5 An example use case of the Model Selector. Using equation 4.2: $p + 1717.78 < 1872.03$. Therefore the `premodel` must be quicker than 154.25ms in order to be effective in this case.

| Model Name | i | Proportion of Input(o) | DNN runtime (m)(ms) | o*m |
|---|---|---|---|---|
| Inception_v4 | 1 | 0.78 | 1674.27 | 1305.93 |
| ResNet_v2_152 | 2 | 0.22 | 1872.03 | 411.85 |
| Totals | – | 1.0 | – | 1717.78 |

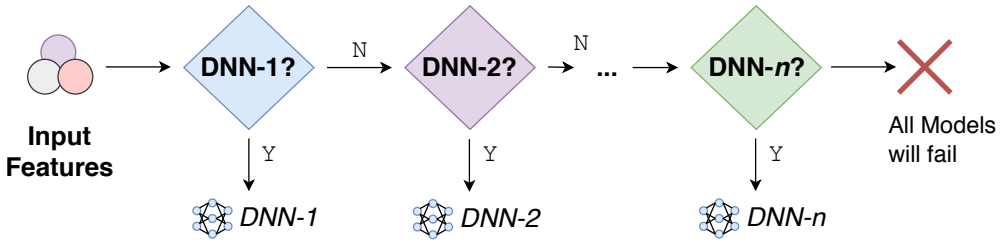



Fig. 4.6 An example of a multi-classifier architecture of `premodel`. Each diamond represents a separate SML classifier which decides whether to choose a specific DNN or not. The process for selecting which DNNs are included and their order is described in section 4.2.3.

### 4.2.2 Premodel Design

Due to the nature of the environment of the model selection process, there are two main requirements to consider during design: (i) fast execution time, and (ii) high accuracy. A model selector which takes longer than the DNNs it is choosing between would outweigh its benefits. Furthermore, an inaccurate `premodel` would often choose a DNN which is not optimal for the given input leading to one of two cases: (i) the chosen DNN can correctly classify the input, but is slower than the optimal DNN; or (ii) the chosen DNN is unable to correctly classify in input, leading to an incorrect classification. Therefore, a highly accurate `premodel` is imperative to achieve a reduced overall cost, and a better overall score across a dataset. Using a standard SML classifier, such as a SVM, can yield acceptable results across DNN applications. However, during experimentation it was discovered that performance could be maximised by using multiple SML classifiers in sequence. Therefore, an automatic approach to `premodel` generation was created,

**Algorithm 1** DNN Selection Algorithm

**Require:** *training_data, θ, selection_method*

```
current_DNNs = []
current_DNNs.add(most_optimum_DNN(training_data))
current_accuracy = calculate_accuracy(current_DNNs)
accuracy_difference = 100
while accuracy_difference > θ do
    scoring_metric = next_scoring_metric(selection_method)
    next_DNN =
        ↪ greatest_improvement_DNN(training_data, current_DNNs, scoring_metric)
    current_DNNs.add(next_DNN)
    new_accuracy = calculate_accuracy(current_DNNs)
    accuracy_difference = new_accuracy − current_accuracy
    current_accuracy = new_accuracy
end while
```

improving performance across DNN applications, such as image classification, or machine translation – also called *domains*. Below, the architectures and SML classifiers that were considered are introduced.

**SML Classifiers.** A number of fast predicting, well known SML classifiers were considered. Only SML classifiers were considered due to their simplicity and speed. A neural network could achieve a higher accuracy than a SML classifier, however their added cost does not warrant the potential accuracy gain. The SML classifiers that are considered are: K-Nearest Neighbour (KNN), a simple clustering based classifier; Decision Tree (DT), a tree based classifier; Naive Bayes (NB), a probabilistic classifier; and Support Vector Machines (SVM), a more complex, but well performing classification algorithm.

**Single-Classifier Architecture.** A single-classifier architecture premodel works in the same way as a 'vanilla' SML model. The features of an input are given to the model, and the predicted label is returned. In the context of a model selector, the SML is usually tasked with a multi-class classification problem, which can make predictions less accurate as the number of classes increases.

**Multi-Classifier Architecture.** A multi-classifier architecture premodel is made up of multiple SML classifiers organised in sequence, where each SML is tasked with deciding whether to use a specific DNN or not. Figure 4.6 shows an example of a multi-classifier architecture premodel. The choice and order of DNNs included in this architecture is described in more detail in section 4.2.3. Using this architecture can lead to increased accuracy in some problem domains at little added cost to the inference time. Furthermore, if each SML in the premodel is the same, *e.g.* a KNN classifier, further optimisations can be made to curtail the overhead introduced by using multiple classifiers.

**Algorithm 2** Most Optimum DNN

**Require:** *training_data*
1: $counter\_map = initialise\_counter\_map()$
2: **for** *input* **in** *training_data* **do**
3: $DNN\_O = optimal\_DNN(input)$
4: $counter\_map[DNN\_O] += 1$
5: **end for**
6: $(most\_optimal\_DNN, most\_optimal\_count) = max\_val\_pair(counter\_map)$
7: **return** *most_optimal_DNN*

### 4.2.3 DNN Selection Algorithm

A key problem to solve for a model selector is choosing which DNNs to choose from, *i.e.* the pool of available DNNs. A simple solution would mean training the premodel to select between every DNN the user supplies, however, this can lead to low premodel accuracy, long selection times, and high memory usage. Furthermore, experimentation shows that simply increasing the DNN count results in diminishing returns. The solution presented in this thesis attempts to find a balance between DNN count and the overall score of the chosen evaluation metric, *e.g.*top-1 or F1-score; it is termed the *DNN Selection Algorithm*.

Algorithm 1 describes the DNN selection algorithm. Note that the algorithm uses the term optimum/optimal, in this context the most optimum DNN for a single image is that which gives the correct output for the lowest cost, *e.g.* if two DNNs produce the correct output then the faster one is the optimal DNN; here we use $\texttt{DNN}_O$ to mean the optimal DNN. Algorithm 1 also makes use of the function *most_optimum_DNN* which is described in detail in Algorithm 2. Put simply, Algorithm 2 counts the number of times each DNN is $\texttt{DNN}_O$ for every input in the training dataset, and return the DNN that has the highest count. For example, considering the following counts of $\texttt{DNN}_O$ across 25 inputs:

- MobileNet_v1_100 - 12
- Inception_v4 - 8
- ResNet_v2_152 - 5

in this case, algorithm 2 would return MobileNet_v1_100.

Algorithm 1 requires three input parameters: *training_data*, the training data for the premodel (described in more detail in section 4.2.5); $\theta$, a threshold parameter used to decide when to stop; and *selection_method*, one of a choice of methods that define the evaluation method (*scoring_metric*) to use when selecting the next DNN during each iteration. Algorithm 1 line 2 shows that the first DNN selected is always the most optimum, that is, the DNN that is optimal for most of the data in *training_data*. Further DNN selection is decided by the *selection_method* which gives a *scoring_metric* in each iteration (see

line 6). Different selection methods have been implemented to make the algorithm more adaptable to the user's needs. The *selection_method* can be a choice of:

- **Accuracy.** Each iteration of the loop chooses the DNN that gives the highest increase overall to the joint DNN accuracy. Including the first DNN selection criteria the selected DNNs follow the pattern: most optimal, most accurate, most accurate, etc.
- **Optimal.** Each iteration of the loop chooses the DNN that is optimal for the most of the training data that currently cannot be predicted correctly; therefore optimising the choice for any images that cannot be predicted with the current selection. Including the first DNN selection criteria the selected DNNs follow the pattern: most optimal, most optimal, most optimal, etc.
- **Alternate.** A hybrid of the first two approaches. This method alternates between choosing the most optimal and the most accurate DNN in each iteration. Including the first DNN selection criteria the selected DNNs follow the pattern: most optimal, most accurate, most optimal, most accurate, etc.

In general, algorithm 1 walks through the following steps: (i) First, select the most optimal DNN, that is, the DNN that is optimal for most of the training data; (ii) Determine the scoring metric for this iteration, based on the selection method; (iii) Consider, in turn each of the remaining unselected DNNs and select the one which bring the greatest improvement to the scoring method; (iv) Repeat step (iii) until the accuracy improvement of the step is less than $\theta$; (v) Terminate, the best models have been selected. The DNN selection algorithm has been designed to select the DNNs that are best able to compliment one another when working together, maximising accuracy while minimising runtime.

### 4.2.4 Feature Selection

As mentioned in section 4.2.1 the premodel requires a set of features to work effectively. Selecting the right features to characterise the input is key to building a successful SML classifier. However, the right features will change depending on the problem domain. To avoid requiring the user to filter and choose the best features to use depending on the problem domain an automatic feature filtering and selecting process was created; the user is simply required to provide a set of candidate features. Automatic feature generation could be used to provide candidate features, however this is out of the scope of this work [71, 13]. This section will describe how the automatic feature selection and scaling works.

Due to the speed of the premodel, feature extraction is the biggest overhead of this approach. Therefore, by reducing the feature count the overall overhead of this

approach can be reduced. Furthermore, reducing the feature count also improves the generalisability of the `premodel`, *i.e.* reducing the likelihood of over-fitting on the training data.

**Feature Correlation.** The first step is correlation-based feature selection, described in detail in section 2.2.2. Pearson product-moment correlation (PCC) is used, which returns a value between $-1$ and 1 to indicate the similarity between two features. The closer the absolute value is to 1, the more similar the two features are, meaning they represent similar information. It was empirically decided that a threshold value of 0.75 is effective at this stage. If the absolute value of PCC between two features is greater than 0.75, then one of the features is removed, retaining the other.

**Feature Importance.** Next, the importance of each feature is evaluated. In order to evaluate the importance of each feature a baseline is needed. To get a baseline the `premodel` is first trained and evaluated using K-fold cross validation (see Section 4.2.5) using all of the current features, recording the accuracy. Each feature is then removed in turn and the `premodel` is retrained and re-evaluated, noting the change in accuracy from the baseline. Intuitively, if there is a large drop in accuracy when a feature is removed then the feature must be important for the task at hand. Alternatively, if the accuracy drop is very small, or accuracy increases the feature must be unimportant. It was empirically determined that if the accuracy drop is greater than 1% then a feature is deemed important. This step is performed iteratively, performing a greedy search, removing the least important features one by one. At the end of this stage all remaining features are deemed important.

**Feature Scaling.** The final step in feature selection is feature scaling, or *normalising*, bringing all features into a common range in order to prevent the range of any single feature being a factor in its importance. Feature scaling also reduces the computation time of the `premodel`. All features are normalised, bringing them into a range between 0 and 1 using the following equation:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{4.3}$$

where $x$ is all values for this feature, $x_i$ is the value to normalise, and $z_i$ is the normalised value.

To facilitate feature scaling during deployment, the minimum and and maximum value of each feature in the training set is recorded; they are used to scale the corresponding features of new data. If a feature value falls outside of the minimum/maximum range during deployment then it is still scaled accordingly, however its normalised value will be capped at 1.0. For example, with a minimum value of 0, and a maximum value of 10, the feature value 15 would be scaled and capped to 1.0.

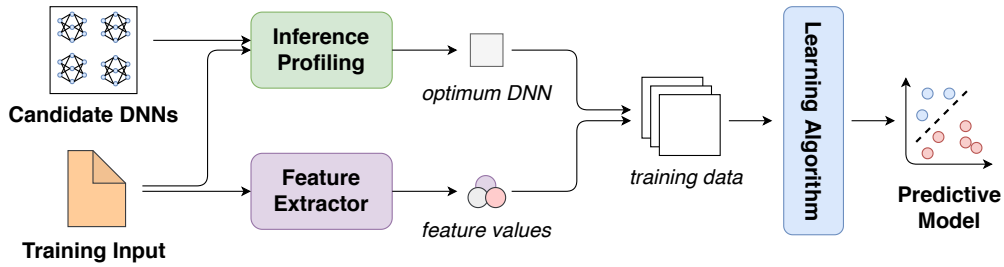



Fig. 4.7 The `premodel` training process. The same procedure is used to train each individual `SML` in a multi-classifier architecture `premodel`.

### 4.2.5 Premodel Training

Training the `premodel` follows the standard `SML` training procedure, summarised in Figure 4.7. In general, the optimal `DNN` for each training image needs to be discovered through inference profiling, it is then combined with the input features to create a training dataset. The training dataset will summarise the best `DNN` to use based on a set of features. Finally, the `premodel` is trained on the training data so it is able to predict the optimal `DNN` for any new inputs at runtime. This section covers different aspects of training the `premodel`, covering the generation of training data, building the `premodel` and the overall cost of training.

#### Generate Training Data

As mentioned above, the training data consists of a set of feature values for each input combined with the optimal `DNN` for that input. The `premodel` can be trained for different evaluation criteria, *e.g.*top-1 or top-5 for image classification, as the optimal model changes depending on the evaluation criteria. To accurately evaluate the performance of each candidate `DNN` they must be applied to unseen training inputs, that is, input that has not been used to train the `DNN`. Inference profiling (see Figure 4.7) involves exhaustively executing inference of every candidate `DNN` with every training input, measuring the prediction output and execution time. Inference profiling should be carried out on an unloaded machine to reduce noise and speed up data generation; it is a one-off cost, *i.e.* it only needs to be completed once. During development it was discovered that the relative runtime of `DNNs` is stable, *i.e.* the relative time difference between `DNNs` stays the same whether running on an embedded system or a high-performance server; if `MobileNet_v1_100` is faster than `ResNet_v2_152` on a server, it will be on an embedded system too. This means that inference profiling can be carried out on a high-performance server to speed-up data generation. It is worth noting that adding a new candidate `DNN` at a later date simply requires executing inference of the new `DNN` with all inputs while taking the same measurements described above.

Once inference profiling and feature extraction (explained in section 4.2.4) are complete their respective outputs can be combined to produce the premodel training data. As with any machine learning training problem, it is difficult to quantify the exact amount of data required in order to produce accurate predictions. The data presented in Chapter 6 indicate that 10k training inputs is not sufficient, whereas 50k inputs produced accurate predictions. Using the DNN outputs, evaluation criteria, and inference execution times, the optimum DNN for each input can be calculated. The optimum DNN is that which achieves the accuracy goal (top-1 or top-5) in the least amount of time. The features (premodel input) and optimum DNN (ground truth label) for each input are combined to create the full training data for the premodel.

#### Automatically Building the Premodel

Once the training data is available the best premodel architecture for the problem domain can be ascertained. First, the best DNNs to include in the premodel needs to be decided, explained in section 4.2.3. Next, the best classifier and SML classifier to use is searched for, initially through a random search of all possibilities, and then a fine-grained search based on the most accurate combinations. For example, the random search could reveal that SVM and KNN perform effectively at the task at hand, therefore the fine-grained search will investigate if a multi or single-classifier architecture would work best. During the fine-grained investigation the best hyper-parameters for each classifier is also decided.

#### Training Cost

The overall generation and training cost of the premodel is dominated by the generation of the training data. As the candidate SML classifiers are computationally simple, their training and evaluation costs during premodel generation are inexpensive. Furthermore, as more candidate DNNs and training inputs are considered, the more training time will increase. As an example, the total training cost of an image classification premodel– considering 12 candidate DNNs, and 50k training inputs on a NVIDIA P40 GPU – took around 30 hours; 24-25 of those hours were spent on training data generation.

### 4.2.6 Deployment

Deployment of the model selection process has been designed to be simple and easy to use, similar to current DNN usage techniques. All inner workings of the process have been encapsulated, such as the need to extract features from an input and pass them to the premodel. A user would interact with this approach in the same way as they do a typical DNN, passing in an input and receiving a prediction as output. For example, in the case of image classification, the user would pass in an image and be given a prediction of what

that image contains, along with its confidence levels, in return. The following paragraphs briefly describe a deployment process for this technique; a more detailed example is in Section 6.1.1.

First, a premodel needs to be generated and trained; an automatic process, however requires some human input to begin. The user needs to supply: a set of candidate features, as mentioned in Section 4.2.4, these could be automatically generated; a set of pre-trained candidate DNNs, usually easily accessible, but depends on the problem domain; and an evaluation method for the DNNs, allowing the technique to quantify the accuracy of each DNN. Once all data has been supplied the training data is generated, the premodel architecture selected, and the selected premodel is trained. Considering 12 candidate DNNs, and 29 candidate feature values, the whole process took around 1.5 days.

The premodel can now be deployed. To make use of the premodel the user will first need to load it into memory, which will automatically load the dependant DNNs into memory at the same time. Inference now follows the same process as using any single DNN, the user will supply the premodel with an input – which, internally, will follow the process in Figure 4.5 – and be returned an output in the same format as the candidate DNNs.

## 4.3 DNN Specialisation - Design and Implementation

This section focusses on work that is not yet published. It builds upon the work published in ACM Transactions on Embedded Computing Systems [89]. Based on the work discussed in Section 3.4, the diversity of the component models within an ensemble is an important metric to take into consideration when using multiple models together. The DNN selection algorithm in Section 4.2.3 attempts to utilise the DNNs with the highest diversity by only including those which significantly increase accuracy; a necessary step when using off-the-shelf DNNs. The problem with using off-the-shelf DNNs is their lack of diversity. They have all be trained to generalise across the entire dataset, and therefore, have little diversity. More could be done to increase diversity, if each DNN was trained and optimised for a subset of the training data then model diversity would increase.

This work aims to build a pool of bespoke DNNs that are designed to work together, specialising each bespoke DNN for a specific segment of the training data. By adapting the DNNs specifically for this purpose, they can collectively achieve a higher accuracy, lower overall runtime, and lower memory consumption than using off-the-shelf DNNs. Furthermore, if the bespoke DNNs are all generated based on the same seed DNN there is the opportunity for weight sharing, further reducing memory requirements. The following subsections provide an overview of the design and implementation of this work.

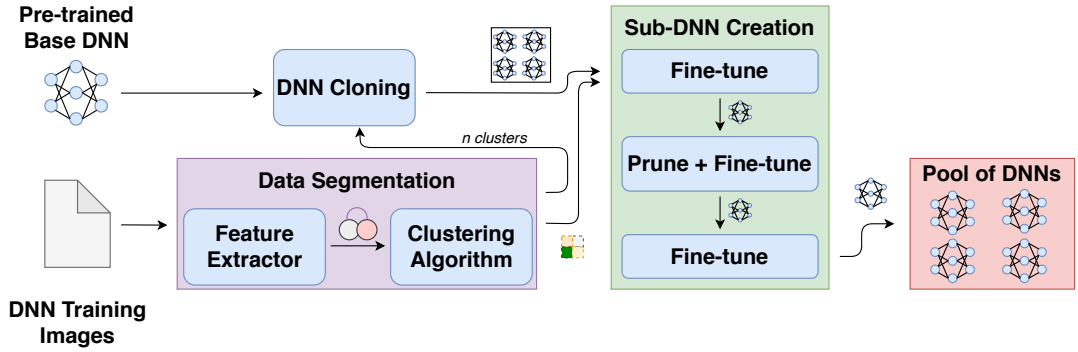



Fig. 4.8 An overview of how the DNN specialisation process works. First, the training data is segmented using a clustering based method. The data segments are then used to create *N* sub-DNNs during DNN specialisation, where *N* is the number of data segments.

### 4.3.1 Overview

The first step in building a pool of bespoke DNNs is deciding how many to build, and what data each DNN will specialise in. In general, the more diverse the set of component models in an ensemble, the better the overall accuracy [86]. Therefore, by generating a more diverse set of bespoke DNNs, the overall accuracy of the approach described in Section 4.2 can be improved. The question now becomes "What is the best way to split the data for the task at hand in order to increase diversity in each split?"; a question that can be tackled through the use of clustering. Clustering, or *Data Segmentation* as it is termed in this thesis, now has a number of problems to solve: (i) How many clusters to use, and therefore the number of bespoke DNNs to generate; (ii) Which features to use to represent the data for clustering; and (iii) What value to use for each of the clustering hyper-parameters. They are discussed in more detail in Section 4.3.2.

Next, the pool of bespoke DNNs needs to be generated based on the Data Segmentation output. To reduce the overall runtime, and memory consumption, of this approach the generated DNNs need to be as small as possible. Furthermore, if the bespoke DNNs can share some parts of their networks, memory can further be reduced with the implementation of weight sharing. Generation of the bespoke DNNs is based on the work discussed in Section 3.1.1; designed to prune a DNN down to its smallest possible size, reducing runtime and memory consumption, without affecting accuracy. During this stage the problems to consider are: (i) How much to prune a bespoke DNN without affecting accuracy on its specific task, and (ii) How much to fine-tune a bespoke DNN to avoid over-fitting. They are discussed in more detail in Section 4.3.3.

Figure 4.8 provides an overview of how the DNN specialisation process generates sub-DNNs which will populate the DNN pool shown in Figure 4.5. The first stage of generating the pool of sub-DNNs is termed *Data Segmentation*; it is responsible for splitting the DNN training data into *data segments*. Similar to the premodel in Section 4.2, Data Segmentation is based on a set of features, such as the number of edges and brightness of

an image, which are extracted by the *feature extractor*. A set of candidate features need to be supplied that are narrowed down through an automatic feature selection process, described in detail in Section 4.3.2. The Data Segmentation process will decide how many segments to split the training data into, and the best way to split it. Next, the data segments are passed into the *sub-DNN Creation* process, which will make a copy of the base DNN for each data segment. To begin Sub-DNN Creation it is passed multiple data-segment-base-DNN pairs. The Sub-DNN Creation process is responsible for creating a specialised DNN for each data segment through a process of fine-tuning and pruning; explained in more detail in Section 4.3.3. Finally, the generated sub-DNNs are added to a pool, and form the pool of DNNs shown in Figure 4.5. The model selection and inference process then follows the same algorithm proposed in Section 4.2, resulting in the most useful and diverse set of sub-DNNs being utilised. After sub-DNN generation, inference will work in exactly the same way as a normal, single model inference process *i.e.* the input and output (classification result) will be in the same format. Successful sub-DNN generation will result in a number of faster, more specialised DNNs capable of faster overall inference without a loss in accuracy.

By combining the ability to build a pool of diverse DNNs to select from, with a premodel (described in Section 4.2), more accurate, faster running and more memory efficient DNN inference can be achieved. The following sections provide an in depth description of each component of the DNN specialisation process. The sections will cover Data Segmentation, Sub-DNN Creation, premodel generation, and deployment.

### 4.3.2 Data Segmentation

The Data Segmentation process aims to find the best way to split the DNN training data; how many splits should be made, and which data forms each data segment. The full DNN specialisation process described in Figure 4.8 is a time consuming process from end to end (dominated by the Sub-DNN Creation process), therefore, it is important that good data segments are found before generating the sub-DNNs. Good Data Segmentation will result in a pool of smaller, faster, and more specialised DNNs with the same collective inference capability as the original base DNN. Furthermore, bad Data Segmentation would impact the premodel accuracy during inference as the premodel would find it difficult to differentiate the optimal DNN for each input; leading to incorrect DNN choice, and incorrect classification outputs.

Data Segmentation is based on a set of quantifiable properties – or *features*, such as the number of edges and brightness of an image – of the input; extracted by the *feature extractor*. A set of candidate features need to be supplied that are narrowed down through an automatic feature selection process, described in detail in an upcoming subsection. The feature extraction and selection process is similar to that described for the premodel

in Section 4.2.4. The main difference between the two extraction processes is how the importance of each candidate feature is determined; the `premodel` is a supervised learning algorithm, whereas Data Segmentation is based on an unsupervised learning algorithm. The similarities and differences are clarified in an upcoming subsection. Once the features are extracted they are passed into an unsupervised clustering algorithm that is responsible for deciding how to segment the data. The K-Means clustering algorithm will look for similarities in the input data to group the inputs into $k$ segments. Similar to the `premodel` described in Section 4.2, the data segmentation process has been designed to adapt to different `DNN` domains. A set of candidate features need to be supplied by the user, then Data Segmentation will choose the best value of $k$ to use, and the best features. Unfortunately, as the set of candidate features grows in size, the possible combinations of features grows exponentially. To make the search for the best parameters feasible the clustering algorithm needs to be quickly evaluated.

The following sections describe: how the clustering output is evaluated, to infer whether it is a good clustering or not; how feature selection is carried out, specifically, how it differs from the work in Section 4.2.4; and how the number of data segments is decided. Finally, this section ends with a summary containing a worked example, showing the outcome of each step.

#### Clustering Evaluation

A key problem to solve when segmenting the training data is choosing the best features and number of clusters ($k$) to use; $k$ directly relates to the number of data segments created. Here, the search space can be massive, and grows exponentially as more candidate features are added, due to the number of feature combinations growing exponentially. Furthermore, a full evaluation from end-to-end can take days to run, dominated by compute intensive and time consuming `DNN` fine-tuning. Exacerbating the problem further, a poor choice of features or $k$ value will lead to overlapping clusters, resulting in poor `premodel` accuracy during inference. Therefore, it is imperative to evaluate the data segmentation of each combination of features and $k$ before any fine-tuning takes place; the best set of features and value of $k$ can then be used in end-to-end evaluation.

To evaluate each data segmentation, two evaluation metrics are used: Mean Squared Error (MSE), and Mean Silhouette Coefficient (MSC). The chosen metrics are common clustering evaluation metrics used in previous work [8, 100, 10, 132]. Each evaluation metric is described below:

**MSE.** MSE is the mean of the squared differences between each cluster element and its centroid [75]. It is a measure of how distinct each cluster is from all other clusters; MSE is also able to value the variability of the data assigned to a cluster. The equation for

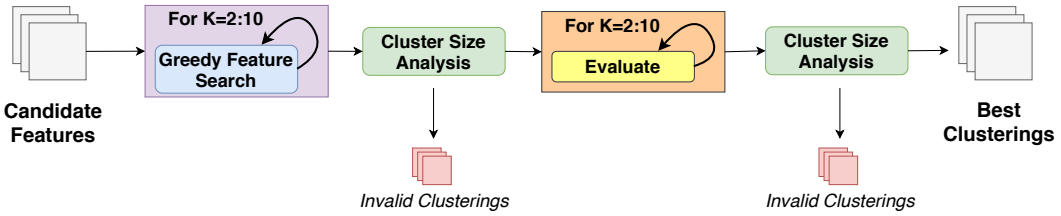



Fig. 4.9 An overview of how the data segmentation process works. First, a greedy search is performed for every value of K, in this case from 2 to 10, and scores recorded. The size of the clusters are then analysed, removing any invalid feature-sets. Next, the remaining feature-sets are evaluated for every value of K, and new scores recorded. Finally, the new clustering cluster sizes are analysed, removing any invalid clusterings. The best feature-set-k combination is the chosen from the remaining clusterings. A full walk-through of this process, including an example, is given in Section 4.3.2.

calculating MSE is:

$$MSE = \frac{\sum_{i=1}^{n} x_i - \bar{x}_i}{n} \tag{4.4}$$

where $x_i$ is sample $i$, and $\bar{x}_i$ is the centroid for $x_i$. When calculating SSE *lower is better*.

**MSC.** MSC is the mean of the Silhouette Coefficient [111] for each sample in the dataset. It is a measure of how well formed the predicted clusters are. MSC returns a value between 1 and $-1$, ranging from best to worst, respectively. Values close to 0 indicate overlapping clusters, and negative values generally indicate that a sample has been assigned to the wrong cluster, *i.e.* a different cluster is closer and more similar. The equation for calculating MSC is:

$$MSC = \frac{\sum_{i=1}^{n} \frac{b_i - a_i}{max(b_i, a_i)}}{n} \tag{4.5}$$

where $a_i$ is the difference between sample $i$ and its centroid (intra-cluster distance), and $b_i$ is the difference between sample $i$ and the nearest centroid that it is not a part of (nearest-cluster distance). When calculating MSC *higher is better*.

### Data Segmentation Feature Selection

Selecting the right features to characterise the input is key to building a successful clustering algorithm. However, the right features will change depending on the problem domain. To avoid requiring the user to filter and choose the best features to use depending on the problem domain an automatic feature filtering and selecting process was created; the user is simply required to provide a set of candidate features. Feature correlation and feature scaling follow the same process for feature selection as described in Section 4.2.4, it will not be repeated here.

**Algorithm 3** Initial Data Segmentation Algorithm

```
Require: candidate_features
for k_valinrange(2,10) do
    remaining_features = candidate_features
    while length(remaining_features) >= 2 do
        feature_scores = {}
        for featureinremaining_features do
            temp_features = remove_feature(feature, remaining_features)
            score = evaluate_features(temp_features)
            feature_scores[feature] = score
        end for
        write_to_disk(feature_scores)
        least_imp_feature = get_least_important(feature_scores)
        remaining_features = remove_feature(least_imp_feature, remaining_features)
    end while
end for
```

The key problem to solve here is choosing how many ($n$), and which, features to use. A simple solution would be an exhaustive search. However, the number of iterations ($I$) grows exponentially as $n$ increases due to the exploding number of possible combinations of features. The total number of combinations $I$ for any value of $n$ can be calculated by:

$$I = 2^n - 1 \tag{4.6}$$

To reduce the search space a simple greedy search is implemented instead. The greedy search works by determining the importance of each of the $n$ features. Feature importance can be determined by the change in score between the original dataset, and the dataset with that feature removed. The least important feature is then removed, and the process repeated with $n-1$ features, terminating when $n = 1$. Using Big O Notation, the number of iterations required is significantly reduced from $O(C^n)$ to $O(n^2)$. Using a greedy search, the number of iterations $I$ comes to:

$$I = \frac{n(n+1)}{2} \tag{4.7}$$

### Number of Data Segments

Choosing the number of data segments, and therefore the value of $k$ clusters, requires careful consideration. Too few clusters can lead to a lot of overlap between data segments, meaning a difficult classification problem for the `premodel`, and less chance to specialise each sub-`DNN`. Too many clusters can lead to long sub-`DNN` training times, and a complex classification problem for the `premodel`. Furthermore, the best number of data segments will change depending on the problem domain. The possible values of $k$ that were considered ranges from 2 to 10; 10 is taken as the upper limit as 10 or more sub-`DNNs`

would reduce the training data for each sub-DNN significantly, therefore reducing the effectiveness of training during fine-tuning [47, 155]; Section 4.3.3 provides more detail. Figure 4.9, and Algorithm 3 present an overview of the process described in this section.

In its simplest form, the search for the best value of $k$ forms an outer loop around feature selection (4.3.2), that is, the best features will be found for each value of $k$. As each value of $k$ presents a unique feature optimisation problem, the optimal features are likely to vary between $k$ values. Therefore it is important that each optimal feature-set from each value of $k$ is analysed individually at first, to avoid any single feature-set dominating. During initial experimentation it was found that, in some cases, a greedy feature search can lead to very unbalanced clusters. In some cases, when $k = 2$ one cluster would contain 90% of the data or more; this is an unfortunate side effect of a more time efficient search method. To avoid such bad feature-sets being chosen for deployment a cluster size analysis phase was included in data segment selection. A *minimum cluster size threshold* ($\theta$) value of 10 (representing 10%) was chosen as a sub-DNN trained on less than 10% of the data would have reduced effectiveness [47, 155].

Initially, the three best performing feature-sets, regarding their MSC scores, for each value of $k$ are extracted. Next, the cluster sizes are analysed to make sure they pass an initial minimum cluster size threshold $\frac{\theta}{2}$; the value is divided by 2 in this case to allow extra leeway during initial analysis. The remaining feature-sets are then considered the 'best' feature-sets for this data. Each of the best feature-sets are then evaluated against every value of $k$, recording their MSE and MSC scores. Again, the cluster sizes are analysed, any feature-set-$k$ combination that produces a cluster containing less than $\theta$% of the data is removed. Note that the full value of $\theta$ is considered in this case, the clusters generated here directly relate to the training data used for each sub-DNN. This stage results in each best feature-set being scored against every considered value of $k$, and the infeasible combinations have been removed. The next stage focusses on choosing the best feature-set-$k$ combination.

A common method for choosing the best number of clusters is known as the *elbow method*, which is based on the MSE scores. However, the elbow method is often unreliable and ambiguous [60]; this was the case during pilot experiments. A better method involves choosing the feature-set-$k$ combination that achieves the highest MSC score [59], indicating well formed, non-overlapping clusters. Based on pilot experiments, a hybrid of the two mentioned approaches was adopted; fully evaluated in Section 6.2.2. The hybrid approach considers a tolerance ($\delta$) below the maximum value of MSC score, the chosen feature-set-$k$ combination is the chosen based on the lowest MSE score within the threshold.

Table 4.6 All candidate features considered during Data Segmentation.

| Feature | Description |
|---|---|
| *n_keypoints* | # of keypoints |
| *avg_brightness* | Average brightness |
| *brightness_rms* | Root mean square of brightness |
| *avg_perc_brightness* | Average of perceived brightness |
| *perc_brightness_rms* | Root mean square of perceived brightness |
| *contrast* | The level of contrast |
| *edge_length{1-7}* | A 7-bin histogram of edge lengths |
| *edge_angle{1-7}* | A 7-bin histogram of edge angles |
| *area_by_perim* | Area / perimeter of the main object |
| *aspect_ratio* | The aspect ratio of the main object |
| *hue{1-7}* | A 7-bin histogram of the different hues |

### Summary

Once Data Segmentation is complete, it will output a number of data segments, based on the clusters, that should be easily separable. To clarify, Data Segmentation works on the DNN domain *training data*, as it is used during Sub-DNN Creation to specialise each sub-DNN. To end this section, a worked example is provided to help demonstrate Data Segmentation First a setup is described, followed by an example of the greedy feature search is given. Next the example shows how the best feature-sets are chosen, finishing with choosing the best feature-set-$k$ combination.

**Setup.** In this example 29 candidate features are supplied the Data Segmentation process, they are listed in Table 4.6. Features are extracted from 500k randomly selected images in the ImageNet *training* dataset; there is an equal number of images selected from each of the 1k ImageNet classes. OpenCV and SimpleCV were used to extract features from each image. For this example $\theta = 10$, and $\delta = 0.08$; during pilot experiments these values produced good results. Both values are analysed in the Section 6.2.2.

**Greedy Feature Search.** The goal of data segmentation is to decide the best number of clusters, and the best features to use. This paragraph focusses on choosing the best features. This example walks through the first two iterations of a greedy feature search where Table 4.6 shows all candidate features, and $k = 2$. Note that this process will be repeated for every considered value of $k$. First, feature correlation and feature scaling is carried out, following the same process as described in Section 4.2.4; the remaining features are shown in Table 4.7 in the left column. Next, the importance of each feature is calculated (shown in Table 4.7). As a reminder, feature importance can be determined by the change in MSC score between using all features, and the same clustering with that feature removed. In this instance, the *contrast* feature is removed as removing it actually

Table 4.7 The importance of every feature during the first iteration of a greedy search for feature selection using $k = 2$. Features are sorted by increasing importance, the least important feature is at the top. *higher is better*.

| Feature | Importance |
|---|---|
| *contrast* | -0.068841 |
| *avg_brightness* | -0.028825 |
| *edge_length_1* | -0.019215 |
| *edge_length_2* | -0.009389 |
| *edge_length_3* | -0.006084 |
| *edge_angle_6* | -0.004618 |
| *area_by_perim* | -0.002911 |
| *edge_angle_2* | -0.002905 |
| *n_keypoints* | -0.000784 |
| *hue7* | 0.000155 |
| *hue1* | 0.000349 |
| *aspect_ratio* | 0.000729 |
| *edge_angle_4* | 0.002021 |
| *edge_angle_5* | 0.004675 |
| *edge_angle_3* | 0.00743 |

Table 4.8 The importance of every feature during the second iteration of a greedy search for feature selection using $k = 2$. Features are sorted by increasing importance, the least important feature is at the top. *higher is better*.

| Feature | Importance |
|---|---|
| *avg_brightness* | -0.054427 |
| *edge_length_1* | -0.034823 |
| *edge_length_2* | -0.017825 |
| *edge_length_3* | -0.014427 |
| *edge_angle_6* | -0.010785 |
| *edge_angle_2* | -0.009257 |
| *area_by_perim* | -0.00531 |
| *n_keypoints* | -0.003187 |
| *hue7* | -0.002292 |
| *aspect_ratio* | -0.001494 |
| *hue1* | -0.000552 |
| *edge_angle_3* | 0.001246 |
| *edge_angle_5* | 0.00188 |
| *edge_angle_4* | 0.188589 |

Table 4.9 A list of feature-set-$k$ combinations that produce the best MSC scores after a greedy search across all values of $k$. The combinations that produce invalid cluster sizes have been removed. MSC and MSE scores achieved by each feature-set are presented in Figures 4.10 and 4.11, respectively. A full list of the features contained in each feature-set is provided in Appendix A.

| | | | |
|---|---|---|---|
| KMeans_2_193 | KMeans_2_187 | KMeans_3_228 | KMeans_3_233 |
| KMeans_3_171 | KMeans_4_320 | KMeans_4_190 | KMeans_5_373 |
| KMeans_6_199 | KMeans_6_376 | | |

increases the overall MSC score. The importance of each feature is then re-calculated based on the new feature-set, giving the values presented in Table 4.8. In this instance, the *avg_brightness* feature is removed, completing the second iteration of the feature search. This process is repeated until 3 features remain.

**Choosing the Best Feature-Sets.** During the greedy feature search every feature-set is given a unique *feature-set ID*, integers of increasing value starting at 1. As an example, the feature-set with ID 1 contains all features listed in Table 4.7, whereas the feature-set with ID 199 only contains the following features: *edge_angle_4*, *area_by_perim*, *aspect_ratio*, *hue1*, and *hue7*. A full list of the features contained in each relevant feature is provided in Appendix A. At this stage, each feature-set-$k$ combination is named

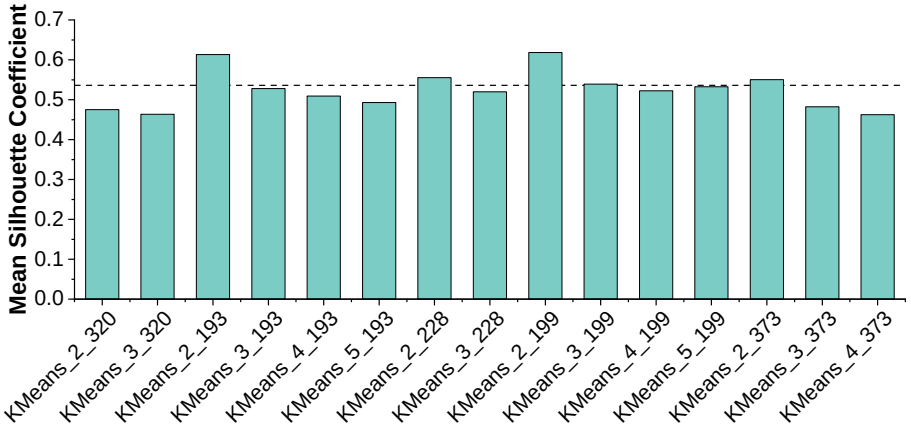



Fig. 4.10 Mean Silhouette Coefficient (MSC) of the best feature-sets (Table 4.9) across every value of $k$. The $x$ axis of this figure matches Figure 4.11. Lower MSC threshold, determined as $max(MSC) - \delta$ is shown as a dashed line. For MSC, *higher is better*.

with the following convention: KMeans_$k$_$i$, where $k$ is the number of clusters (or data segments) considered, and $i$ is the feature-set ID; KMeans_6_199 is 6 data segments created using the feature set described earlier. Table 4.9 shows the feature-sets from each value of $k$ that achieved the best MSC scores, the feature-sets that produce cluster sizes less than $\frac{\theta}{2}$ have been removed. In this case 10 best feature-sets have been chosen, each one is now scored against every value of $k$ being considered.

**Choosing the Best Feature-Set-$k$ Combination.** The final stage of Data Segmentation is choosing the best feature-set-$k$ combination overall that produces the best Data Segmentation; Figure 4.9 presents an overview of this process. Figures 4.10 and 4.11 show the MSC and MSE scores of all feature-set-$k$ combinations, respectively; the combinations that produce cluster sizes less than $\theta$ have been removed. First, MSC scores are analysed and the best scorer is found, in this case that is KMeans_2_199. Next, the value of $\delta$ is taken away from the MSC score of the top scorer, all feature-set-$k$ combinations that achieve a MSC score greater than the result are now considered. In this example, KMeans_3_199 achieves an MSC score of 0.618, and $\delta = 0.08$, therefore the MSC score lower bound becomes 0.538 (as $0.618 - 0.08 = 0.538$), marked with a dashed line in Figure 4.10. Of the 15 feature-set-$k$ combinations, only 5 are within the threshold: KMeans_2_193, KMeans_2_228, KMeans_2_199, KMeans_3_199, KMeans_2_373. Figure 4.11 shows the MSE scores of all combinations, the combinations within the threshold are marked with a red tilde. In this case there is a clear winner: KMeans_3_199. Therefore the best parameters to use for Data Segmentation in this example are: feature-set ID 199, and $k = 3$.

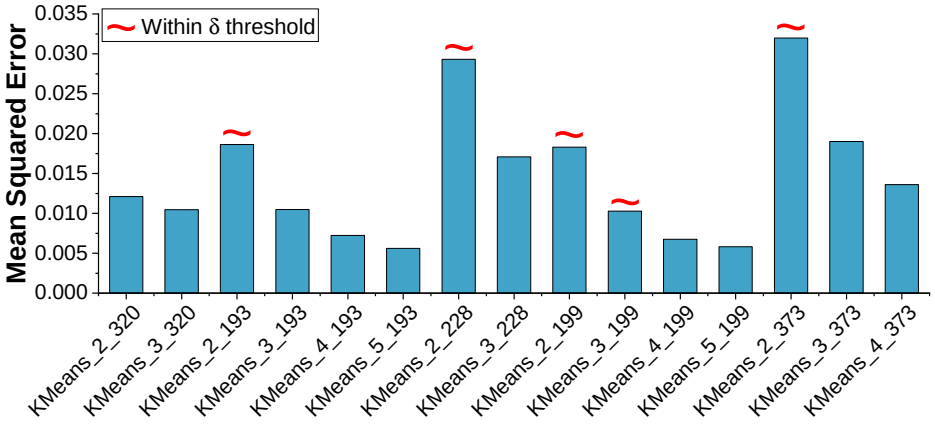



Fig. 4.11 Mean Squared Error (MSE) of the best feature-sets (Table 4.9) across every value of $k$. The $x$ axis of this figure matches Figure 4.10. Red tilde marks show the feature-set-$k$ combinations that are greater than the lower MSC threshold. For MSE, *lower is better*.

### 4.3.3 Sub-DNN Creation

To begin, the sub-DNN creation process is given two inputs: (i) a set of training data segments, each data segment correlating to a cluster; and (ii) clones of the pre-trained base DNN (*e.g.* `ResNet_v2_50`), one for each data segment. Each DNN clone is assigned a data segment. In general, more complex problems require more complex DNNs. After data segmentation each data segment represents a simpler problem to solve, therefore, each sub-DNN will have a simpler problem to solve. Simpler sub-DNNs will lead to cheaper and faster overall inference. The following process is carried out, in turn, for each combination of data segment and DNN to create each sub-DNN.

#### Initial Fine-Tuning

Initially, the DNN clone is fine-tuned on its data segment with a learning rate of $1 \times 10^{-5}$ for 10 epochs, or until the validation loss begins to increase instead of decrease, indicating over-fitting. Such a low learning rate and so few epochs are used as the DNN has already been pre-trained and is expected to be well fitted to the data segment. To clarify, validation loss is the loss score based on the training validation set, that is, 20% of the training data used to validate training; it is not the loss score based on the ImageNet Validation set. Initial fine-tuning aims to begin specialising this DNN clone for its data segment by slightly adjusting the weights, preventing the first iteration of pruning from removing exactly the same neurons for each DNN clone.

### Pruning and Fine-Tuning

Pruning and fine-tuning makes up the majority of the sub-DNN creation process; it is computationally expensive due to iterative fine-tuning. Previous work in DNN pruning indicates that the first layers of a DNN learn general features [153, 115], the later layers building on those general features to form more specialised features and classify more complex problems. Therefore, during this approach pruning begins part way through the model to preserve the general features. Pilot experiments indicated that not pruning or fine-tuning the first 35% of a DNN produced good results, full evaluation in Section 6.2.3. Beneficially, all sub-DNNs will be exactly the same up until the point that pruning begins, therefore, they can share weights.

Initially, each convolutional layer in a sub-DNN is pruned by 40%, that is, 40% of the convolutional filters in each layer are removed; convolutional layers make up the majority of resource consumption in a CNN. During pilot experimentation, 40% was shown to be an effective value for shrinking a DNN without negatively affecting its accuracy. Keras-Surgeon [146] was used to prune the DNNs. Specific sub-DNNs may be pruned more later on after more analysis. Typically, pruning and fine-tuning is an iterative, per-layer process, however, it is also time consuming. In this work, fine-tuning is only carried out every 3-4 layers of pruning, helping to reduce over-fitting and reduces overall runtime of Sub-DNN Creation. Similar to the initial fine-tuning, a low learning rate is used for 5 epochs, or until the validation loss begins to increase instead of decrease, indicating over-fitting. Unfortunately, this also increases the likelihood of a sub-DNN under-fitting on its data segment. Therefore, some final fine-tuning is carried out to ensure a well performing sub-DNN, explained in more detail in the next Section.

An alternate method for Sub-DNN Creation involves first pruning and fine-tuning the base DNN as much as possible on all of the training data (similar to the work in [37]) before cloning, pruning, and fine-tuning to create each sub-DNN. The intuition behind this idea is that there are some common sections of the base DNN that will be removed by all sub-DNNs, therefore this method would save on repeated computation during pruning and fine-tuning. However, this method would restrict the divergence of the individual sub-DNNs and potentially damage the effectiveness of this overall approach.

### Final Fine-Tuning

In order to avoid over-fitting, sub-DNNs are not fine-tuned after each layer is pruned. Final fine-tuning aims to make up for any lost accuracy during pruning and fine-tuning and helps to fully specialise a sub-DNN to its data segment. Similar to initial fine-tuning, final fine-tuning is carried out with a low learning rate on its data segment for 20 epochs, or until the validation loss begins to increase instead of decrease, indicating over-fitting. More training epochs are carried out here as the DNN is expected to need to learn some

new features after pruning has taken place. Fine-tuning after pruning is an important step in order to recover from any lost accuracy due to pruning [29, 46]. Once final fine-tuning is finished this sub-DNN has been created. The process will repeat on the next sub-DNN if more are available to be created.

### 4.3.4 Premodel Generation and Training

Generation and training of a premodel is very similar to that in the previous work (see Section 4.2.5). The features are automatically selected, the premodel is automatically generated, and the premodel is trained via a similar process. This Section briefly covers the areas where this premodel is different to the one described in Section 4.2.

#### Feature Selection

Similar to the premodel in Section 4.2, and clustering during Data Segmentation (Section 4.3.2), this premodel requires the right features to characterise the input in order to build a successful SML classifier. The same automatic feature filtering and selecting process was used to choose the best features based on a set of supplied candidate features; the process is described in more detail in Section 4.2.4. Automatic feature generation could be used to provide candidate features, however this is out of the scope of this work [71, 13].

#### Training

Training the premodel follows the standard SML training procedure. In general, the optimal sub-DNN for each image in the ImageNet *validation* dataset needs to be discovered through inference profiling, it is then combined with the input features to create a premodel training dataset. A different premodel training dataset will be generated depending on the optimisation metric, and problem domain. Finally, the premodel is trained on the premodel training data so it is able to predict the optimal sub-DNN for any new inputs at runtime. Overall, premodel training is very similar to that described in Section 4.2.5, therefore, the same information will not be repeated here. In this work, the premodel does not take the DNN Selection Algorithm (Section 4.2.3) into account, unlike in the previous work in Section 4.2. The Data Segmentation process, if successful, will accurately and effectively segment the data, meaning there will be no need to select the best sub-DNNs; evaluated fully in Section 6.2.2.

### 4.3.5 Deployment

Deployment of DNN specialisation is closely integrated with Model Selection, described previously. It has been designed to be simple and easy to use, encapsulating the inner

workings, and requiring little input from a user. During deployment, a user would interact with this approach in the same way as they do a typical DNN, passing in an input and receiving a prediction as output. DNN specialisation requires some extra pre-deployment computation, however, this is a one-off cost and does not need to be repeated unless the problem domain changes. The following paragraphs briefly describe a deployment process for this technique.

First, the user to supply some data to begin the DNN specialisation process:

1. Some DNN training data for the problem domain.
2. A set of candidate features for the DNN training data, these could be automatically generated.
3. A pre-trained base, or *seed*, DNN, used to generate the sub-DNNs.
4. A set of candidate features for the premodel training data, these could be the same or different to the DNN training data features.
5. An evaluation method for the DNNs, allowing the technique to quantify the accuracy of each DNN.

Once all of the above has been supplied, Data Segmentation can begin, making use of the DNN training data candidate features, and generating the data segments. Data segments, DNN training data, and the seed DNN are then used to generate the individual sub-DNNs, populating the premodel pool of DNNs. Finally, premodel training data is generated, the premodel architecture selected, and the selected premodel is trained. Considering half a million DNN training data points, and 29 candidate feature values for DNN and premodel training data, the whole process took around one week, running on the GPU-Server described in Section 5.1.1.

The premodel can now be deployed. To make use of the premodel the user will first need to load it into memory, which will automatically load the sub-DNNs into memory at the same time. Inference now follows the same process as using any single DNN, the user will supply the premodel with an input – which, internally, will follow the process in Figure 4.5 – and be returned an output in the same format as the seed DNN.

# Chapter 5

# Experimental Setup

This chapter presents the experimental setup used to evaluate the work in this thesis - the results of which are reported in the next chapter. First, the setup of the systems used to train and evaluate models is presented, covering the hardware and software used. Next, the methodologies used to evaluate the different models (`DNN` and `SML`) and different problem domains is presented. Finally, the methods to record and evaluate the results of an end-to-end evaluation are described in detail.

## 5.1 Systems Setup

This Section describes the systems used to train models and evaluate the approaches described in the previous Chapter. First, the hardware and setups used are described in detail, including a brief description of how the system has been used. To end this Section, the deep learning frameworks and `DNN` architectures used are described in detail, along with their uses.

### 5.1.1 Hardware and Software

Two distinct systems were used to evaluate this thesis: a high end GPU-based server, termed *GPU-server*; and an embedded deep learning platform, NVIDIA Jetson TX2. All experimental results are reported based on an implementation on a Jetson TX2; that is, the resource utilisation and model runtime are extracted directly from the Jetson TX2 during runtime. The GPU-server was primarily used for the computationally intensive tasks that are not directly involved in deployment, *e.g.* model training. The hardware and setup of the GPU-server is described first, followed by the NVIDIA Jetson TX2.

### GPU-Server

**Brief Description.** The GPU-server is a high-end system used that has been used for compute intensive tasks such as feature extraction and DNN training. All compute intensive tasks that are not directly involved in the final deployment were executed on this device to save time. The following tasks were executed on the GPU-server: premodel training data generation, premodel training, data segmentation, DNN training, and DNN fine-tuning.

**Hardware.** The server has an Intel Core i7-3770 CPU, with 4 cores and hyper-threading, running at 3.4Ghz (and turbo boost up to 3.9Ghz). Alongside the CPU, there is a 2688 core NVIDIA GeForce GTX TITAN (GK110) running at 876Mhz, with 6GB of GDDR5 internal memory. The system also has 32GB of DDR3 memory, and a further 32GB of virtual memory. Finally, the GPU-server has 1TB of SSD storage.

**Software.** In terms of software, the server is running Debian 10.2 with Linux kernel v4.19.0. Keras v2.2.4 was installed, alongside keras_applications v1.0.7 for access to pre-trained DNN models. Keras used a TensorFlow v.1.14.0 back-end, running with cuDNN (v7.0) and CUDA (v10.0.130); used for DNN model training (see Section 5.1.2). The premodel is implemented using the Python scikit-learn package. The feature extractor is built upon OpenCV and SimpleCV. All testing was run using Python 3.5.2.

### NVIDIA Jetson TX2

**Brief Description.** NVIDIA Jetson TX2 is an embedded deep learning platform. It is a single board computer module designed for embedded applications that require high performance computing [30]. The TX2 has been specifically designed to run GPU workloads, such as DNN inference, on device in a fast and energy efficient manner [112]; making it a perfect evaluation device for this thesis. The TX2 device has been used to evaluate the runtime and energy consumption of individual models, as well as a full end-to-end evaluation of the proposed solution.

**Hardware.** The TX2 system has a 64-bit ARMv8 dual-core Denver2 and a 64-bit ARMv8 quad-core Cortex-A57 running at 2.0 Ghz, and a 256-core NVIDIA Pascal GPU running at 1.3 Ghz. The board has 8 GB of LPDDR4 RAM and 96 GB of storage (32 GB eMMC plus 64 GB SD card).

**Software.** The evaluation platform runs Ubuntu 16.04 with Linux kernel v4.4.15. Keras v2.2.4 was installed, using a TensorFlow v.1.14.0 back-end, running with cuDNN (v7.0) and CUDA (v9.0.252); used for DNN evaluation (see Section 5.1.2). Similar to the GPU-

server, Python 3.5.2 was used with scikit-learn for the premodel, and OpenCV and SimpleCV were used for the feature extractor.

### 5.1.2 Deep Learning Frameworks and Model Architectures

The work in this thesis has been designed to be generally applicable to varying DNN problem domains. Therefore, the work has been implemented in platform and problem domain agnostic deep Learning frameworks, and considers multiple different off-the-shelf pre-trained DNNs. First, the deep learning frameworks that were used are described, followed by a brief description of each DNN architecture.

#### Deep Learning Frameworks

Deep Learning Frameworks (also known as DNN frameworks) offer building blocks for designing, training and validating DNNs through a high level programming interface. They are designed to have easy-to-use interfaces that aim to simplify the implementation of complex and large-scale deep learning models. Furthermore, deep learning frameworks rely on GPU-accelerated libraries such as cuDNN and NCCL to deliver high-performance multi-GPU accelerated training. Widely used deep learning frameworks include: MXNet, PyTorch, TensorFlow and Keras. They can generally be separated into higher and lower level frameworks. As a general rule, lower level frameworks implement the underlying mathematical operations of NNs whereas higher level frameworks abstract away from the underlying operations. Lower level frameworks implement operations such as matrix multiplication, convolutional operations, and activation functions. The user builds upon the mathematical operations to create a neural network, giving the user fine-grained control over the network. Higher level frameworks build upon lower level frameworks and abstract away from the underlying operations in order to release a more readable API in terms of layers, weights, and model saving. This thesis makes use of a lower and higher level framework: TensorFlow and Keras, respectively. They are the most popularly used frameworks [57]; a brief description of both is given below.

**TensorFlow.** TensorFlow [1] was developed by researchers and engineers from the Google Brain team, it is an open source project for – at a generic level – numerical computation using data flow graphs. Each node in the graph represents a mathematical operation, while the edges represent multi-dimensional data arrays, or tensors, that flow between them. The flexible architecture of TensorFlow allows the user to deploy models to one or more CPUs or GPUs in a desktop, server, or mobile device without re-writing code. For visualizing TensorFlow results, TensorFlow offers TensorBoard, a suite of visualisation tools. In this evaluation TensorFlow was used as a back-end for Keras, and

some lower level operations, such as quantisation, were implemented directly in it for optimisation.

**Keras.** Keras [15] is a higher level deep learning framework that builds upon the operations in lower level frameworks. TensorFlow, Theano, or CNTK (all lower level frameworks) can be used as a 'back-end' for Keras, that is, the framework that Keras is abstracting over. Keras provides a high-level API, alongside multiple pre-trained DNNs of popular architectures, that make both prototyping and deployment simple and effective. Unfortunately, Keras' less configurable environment means TensorFlow needed to be used directly for some of the evaluation. In this evaluation, Keras was used where possible to allow fast implementation, testing, and evaluation of the proposed approaches.

### DNN Architectures

Throughout the evaluation, this thesis considers a wide range of influential DNN models, ranging from simple to complex. The DNNs were chosen due to their popularity and usability, as well as their pre-trained availability. The following paragraphs provide brief descriptions of the DNN architectures evaluated, and the naming conventions each one uses.

**Image Classification.** Below is a list, and brief description, of the image classification DNNs used for evaluation in this thesis; all models were trained on the ImageNet ILSVRC 2012 training set. A range of CNNs were considered for evaluation, from low-end, simpler, and less accurate models, to high-end, complex and accurate models. The 14 CNNs used for evaluation are as follows:

- **ResNet.** ResNet [43] NNs are a family of very deep DNNs; they tend to have hundreds of layers. For context, alternate DNN architectures, such as Inception and MobileNet, tend to have tens of layers. In short, ResNet networks introduced the 'residual block' which helped to solve the degradation problem in larger NNs, a problem which limited previous NN architectures in size. Two versions of the ResNet architecture [44], and three different sizes of each version (50, 101, and 152 layers), were considered in the evaluation of this thesis for a total of 6 unique ResNet models. They are named as ResNet_v$i$_$j$, where $i$ is the version number, and $j$ is the number of layers in the model. The ResNet architecture provides a range of high-end, computationally expensive, and accurate DNNs.

- **Inception.** Briefly, the Inception architecture was first introduced by researchers at Google in 2014 [125]. The key idea behind it is the 'Inception Cell' which performs a series of convolutions at different scales and aggregates the results, as opposed to using layers with filters of all the same size - a typical approach in CNN architectures. Four different versions of Inception [126] were considered during

evaluation, they are named as `Inception_v`*i*, where *i* is the version number. The Inception architecture provides a range of mid-end `DNNs`. Their complexity and accuracy sits between ResNet and MobileNet architectures.

- **MobileNet.** MobileNet is a group of `CNNs` that use "depth-wise separable convolutions to build light weight deep neural networks" [50]; primarily designed for fast and efficient mobile inference, although the models are useful on any embedded system. A hyper-parameter, determining the width of the model, was introduced to decide the width of the model, making it adaptable to different applications. Four different widths of MobileNet were considered during the evaluation of this thesis. They are named as `MobileNet_v`*i*`_`*j*, where *i* is the version number, and *j* is a width multiplier out of 100, with 100 being the full uncompressed model. The MobileNet architecture provides a range of low-end, computationally cheap, but less accurate `DNNs`.

A number of popular `CNNs` were not used for evaluation in this thesis. Reasons for an architecture not being used include: A architecture of similar complexity is already being used, the model cannot run on the Jetson TX2, or the architecture has been superseded by a newer architecture already used. The following `CNN` architectures were considered, but not used for evaluation:

- **AlexNet.** AlexNet [66] was one of the first `CNNs` to achieve considerable accuracy in the 2012 ImageNet challenge, achieving a top-5 score of 84.7%. AlexNet is a relatively simple architecture consisting of just 5 convolutional layers, and 3 fully connected layers; there are no alternate versions of AlexNet. Due to its simplicity, and relatively low accuracy when compared to MobileNet and ResNet, it was not included in the evaluation of this work.

- **VGGNet.** VGGNet [118] was the first runner up in the ImageNet classification task challenge in 2014. Although it did not win, it was a significant improvement over the winner of the challenge in the last two years. VGGNet introduced the concept of stacking 3x3 convolutional filters to simulate the effects of using larger filters, such as 11x11 or 5x5 that were used in AlexNet. By stacking convolutional filters VGGNet reduced the overall number of parameters in the model, allowing for deeper `CNNs` to be designed. There are many versions of VGGNet, however the most popular, and widely available as a pre-trained model, are VGG-16 and VGG-19; 16 and 19 refer to the number of layers in the model. VGGNet was not used for the evaluation of this thesis as the number of parameters in the model means that it is unable to run on the Jetson TX2. VGG-16 and VGG-19 contain 138.4M and 143.7M parameters, respectively. For context, the largest

version of ResNet (`ResNet_v2_152`) contains just 60.4M parameters; ResNet also out-performs VGGNet in accuracy.

- **DenseNet.** DenseNet [52] is an architecture of `CNNs` that focusses on a dense set of connections between the layers. Traditional `CNNs` with L layers have L connections - one connection between each layer and the next. DenseNets have $L(L+1)/2$ connections, the feature maps of all preceding layers are used as inputs to a layer, in turn, the feature maps of each layer are used as input to all subsequent layers. By increasing the number of connections DenseNets are able to substantially reduce the number of parameters. The largest version of DenseNet (`DenseNet_201`, 201 is the number of layers) and ResNet (`ResNet_v2_152`) achieve similar top-1 and top-5 accuracy, however DenseNet requires a third of the parameters. ResNet was chosen in place of DenseNet due to their comparable runtimes and accuracies; ResNet is also more widely used for evaluation and therefore, it is easier to obtain a pre-trained model.

**Machine Translation.** Two unique architectures of machine translation `DNNs` were used for evaluation in this thesis. Models were trained using Tensorflow-NMT, a Neural Machine Translation library provided by TensorFlow [85], and the WMT09-WMT14 English-German newstest dataset [1]. Pre-trained `RNNs` of varying sizes and architectures proved difficult to come by, therefore the `RNNs` used in this thesis were trained using the hardware available, limiting the choice. Two different architectures of `RNN` were considered: the standard TensorFlow-NMT model architecture, and the Google Neural Machine Translation Attention (GNMT) [148] architecture. Multiple sizes of each architecture were considered. The `RNN` models are named as `gnmt_N_layer`, prefixing the name with `gnmt_` where the model uses the GNMT architecture, and N is the number of layers in the model. For example, `4_layer` is a default TensorFlow-NMT model made up of 4 layers. In total 15 `RNNs` were considered.

## 5.2 Evaluation Methodology

The evaluation of this work can be split into two general categories: the evaluation of the `DNNs`, and the evaluation of the `premodel`. The evaluation of each category will be carried out in different ways, as each is trying to solve a different problem. The following sections outline the methodology used to evaluate the `premodel`, then each individual `DNN` used in this thesis.

[1] http://www.statmt.org/wmt15/

### 5.2.1 Premodel Evaluation

The proposed `premodel` is evaluated using standard `SML` evaluation techniques. The `premodel` training dataset and *k-fold cross validation* is used, with *k* equal to 10. Specifically, the training dataset is split into 10 sets (folds) which each equally represent the full dataset. One fold is retained for `premodel` testing, and the remaining 9 are used as training data. This process is then repeated 10 times, one for each fold, with each fold being used exactly once for testing.

For example, if the training dataset consists of 50k images, it will be split into 10 sets (folds) of 5k images each. Each fold will equally represent each `premodel` class as much as possible. Next, 9 folds (45k images) are used to train the `premodel`, and the remaining fold (5k images) is used for testing; this process is repeated 10 times.

The accuracy, inference time, and energy consumption of every data point in the dataset is recorded, and an average is taken as the `premodel` score. Section 5.3 gives a detailed explanation on how the inference time and energy consumption are recorded. This standard methodology evaluates the generalisation ability of a `SML` model. For end-to-end evaluation, that is, from input (such as an image) to `DNN` prediction output, the `premodel` outputs (choice of `DNN`) from k-fold cross validation are used to choose the `DNN` for each input.

### 5.2.2 DNN Evaluation

`DNNs` are evaluated in different ways depending on the problem they are trying to solve; different problem domains have conventional ways of evaluating success. In this thesis, two typical problem domains are covered: image classification and machine translation. The same evaluation methodology is used for individual `DNNs` as well as evaluating end-to-end performance, this is possible as end-to-end input and output is similar to individual `DNNs`. The next two Sections cover how each of these domains are evaluated.

#### Image Classification

Image classification is typically evaluated using different forms of accuracy (top-1 and top-5), and calculating metrics based on its confusion matrix (precision, recall, and f1-score). For context, Figure 5.1 shows how to calculate the values in a confusion matrix based on ground truth labels, and predicted labels. The following 5 evaluation methods for image classification are used in this thesis:

- **Top-1** *(higher is better)*. Top-1 scoring is commonly used in the ImageNet Challenge, a popular yearly image classification competition. As input, each `DNN` takes an image, and returns a list of label confidence values as output. Each value indicates the confidence that a particular object is in the image. The resulting list

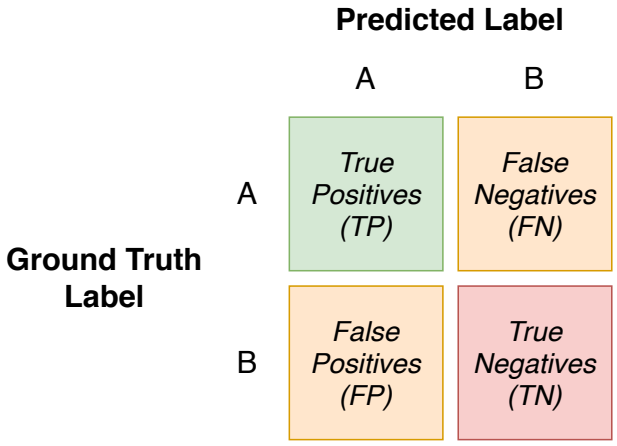



Fig. 5.1 A confusion matrix showing how different values are calculated based on a models predictions. Values are true or false depending on whether the ground truth and predicted label match or not, respectively. Positive and negative is determined based on the predicted label.

of object values are sorted in descending order with regards to their prediction confidence; the label with the highest confidence appears at the top of the list. For top-1 scoring, if the ground truth label is at the top of the list for a particular image, it counts as a match. The top-1 score is then calculated as the average number of matches over the whole test dataset. For example, if 25 of 50 images had a match under the top-1 criteria described above, the DNNs overall top-1 score for the given dataset would be 50%.

- **Top-5** *(higher is better)*. Top-5 scoring is very similar to top-1 scoring, it is also commonly used in the ImageNet Challenge. In this case, once the resulting list of object values have been sorted in descending order, the top five most confident labels are checked for the ground truth label, if it is found it counts as a match. The top-5 score is then calculated as the average number of matches over the whole test dataset.

- **Precision** *(higher is better)*. Precision is a measure of the correctly predicted images to the total number of images that are predicted to have a specific object. Note that when calculating precision, only the top most confident prediction is considered. Mathematically, using the confusion matrix, precision can be calculated using Equation 5.1 below. Colloquially, precision can be thought of as an answer to the question "*Of all the images that are labelled to have a cat, how many actually have a cat?*".

$$Precision = \frac{TP}{TP+FP} \tag{5.1}$$

- **Recall** *(higher is better).* Recall is a measure of correctly predicted images to the total number of test images that belong to an object class. Note that when calculating recall, only the top most confident prediction is considered. Mathematically, using the confusion matrix, recall can be calculated using Equation 5.2 below. Colloquially, recall can be thought of as an answer to the question "*Of all the test images that have a cat, how many are actually labelled to have a cat?*".

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

- **F1-Score** *(higher is better).* F1-score is the weighted harmonic mean of precision and recall; it is used to summarise the precision and recall of a model in a single value. It is useful when the test datasets have an uneven distribution of classes. Mathematically, F1-score can be calculated using Equation 5.3 below.

$$F1\text{-}Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.3}$$

**Machine Translation**

Machine translation is inherently harder to evaluate than image classification. It involves degrees of correctness rather than being either correct or incorrect. Therefore, a different set of metrics are used to evaluate the performance of machine translation `DNNs`. The following 4 evaluation methods for machine translation are described in detail below:

- **BLEU** *(higher is better).* Bilingual Evaluation Understudy Evaluation (BLEU) was first suggested in 2002 [101], since then it has become a widely used evaluation metric for machine translation; it is similar to precision in image classification. BLEU uses n-grams for its calculation, where *n* can be any number. A 1-gram, or unigram, would be each individual word, whereas a 2-gram, or bigram, would be word pairs. BLEU works by counting the matching n-grams in the candidate (`DNN` model output) and reference text (ground truth text), the comparison is made regardless of the order. A value between 1 and 0 is returned by BLEU, 0 being a completely different to the reference text, and 1 being a perfect match to the reference text; a value of 1 is very rarely achieved.

- **BLEU-PS** *(higher is better).* BLEU per second. As BLEU is only able to represent a degree of correctness, BLEU-PS is also used; it is designed to evaluate the trade-off between BLEU and inference time. BLEU-PS is similar to Energy Delay Product (EDP) [69], which is used to evaluate the trade-off between energy consumption and runtime. BLEU-PS is calculated using the formula below.

$$BLEU\text{-}PS = \frac{BLEU \times BLEU}{Inference\text{-}Time} \tag{5.4}$$

- **Rouge** *(higher is better).* Recall-Oriented Understudy for Gisting Evaluation [77] is another commonly used evaluation metric for machine translation. Similar to BLEU, ROUGE measures the degree of correctness of a candidate text; however, unlike BLEU, ROUGE is more similar to recall in image classification. Whereas BLEU measures how often the words (and/or n-grams) in the candidate text appear in the reference text(s), ROUGE measures how often the words (and/or n-grams) in the reference text(s) appear in the candidate text; a subtle difference. A value between 1 and 0 is returned by ROUGE, 1 being a perfect match; a value of 1 is very rarely achieved.

- **F1-Measure** *(higher is better).* Similar to F1-score, F1-measure is the weighted harmonic mean of BLEU and ROUGE scores; it is used to summarise both scores into a single value. For clarity, F1-measure can be calculated using the equation below.

$$F1\text{-}Measure = 2 \times \frac{BLEU \times ROUGE}{BLEU + ROUGE} \tag{5.5}$$

## 5.3 Overall Performance Report

The overall performance of this work is completely evaluated *end-to-end*, that is, from the point that an input value is given, *e.g.* an image, to when the prediction is returned. In Figure 4.5, it is the full traversal from the left to the right. The following sections describe the metrics used for end-to-end evaluation, followed by the implemented strategy to ensure reliable and accurate results.

### 5.3.1 End-to-End Evaluation Metrics

Inference time and energy consumption are important to consider when running `DNNs` on embedded systems. If a network consumes a lot of energy it will become a hindrance in battery powered devices such as mobile phones. Furthermore, `DNNs` are often very computationally intensive and therefore take a long time to run, especially on embedded devices. Both metrics are considered during the evaluation of this work, and are measured at 3 different stages: (i) `premodel` evaluation, (ii) `DNN` evaluation, and (iii) end-to-end evaluation. The following list describes how each metric is recorded for each evaluation stage.

- **Inference Time** *(lower is better).* Inference time is the wall clock time between the start and end point of the evaluation. For `premodel` evaluation, the start point is when the feature extractor starts to extract features, and the end point is once the `premodel` outputs its predicted `DNN`. For `DNN` evaluation, the start point is when the `DNN` receives an input, and the end point is when the `DNN` returns an output.

End-to-end evaluation includes both of the steps above, starting when the feature extractor begins, and ending when the DNN returns an output.

- **Energy Consumption** *(lower is better)*. Energy consumption measures the energy consumed by the running model, over its whole runtime from a start point to an end point. To measure energy consumption, a lightweight runtime was developed to take readings from the on-board energy sensors of the NVIDIA Jetson TX2 at a frequency of 1,000 samples per second. Static power consumption, used by the system when idle, is deducted from the energy readings. The start and end points for premodel, DNNs, and end-to-end evaluation are the same as inference timings.

### 5.3.2 Evaluation Strategy

Recording inference time and energy consumption is inherently noisy. To ensure reliability and reduce noise in the collected values each input is run on each DNN multiple times until a 95% confidence bound smaller than 5% per DNN, per input is achieved. Meaning that, for each input on each DNN there is a 95% chance that the actual mean value falls within 5% of the mean of the repeated tests. The same confidence strategy is not required for all other metrics used in this thesis; each DNN returns a deterministic output when given an input. Furthermore, during experimentation the results exclude the loading time of the DNN models as they only need to be loaded once in practice. Finally, the reported results will present the *geometric mean* of the aforementioned evaluation metrics across cross validation folds.

# Chapter 6

# Experimental Results

This chapter presents a full evaluation of the work described in Chapter 4, using the methods described in Chapter 5. Similar to Chapter 4, this chapter is split into two main sections, each evaluating a different component of the overall thesis. First, the idea of a 'Model Selector' is fully evaluated in-depth, from end-to-end; the evaluation is carried out using off-the-shelf DNN models, without considering DNN specialisation. To end this chapter, DNN specialisation and the generation of a pool of DNNs, designed to work in conjunction with a Model Selector, is evaluated end-to-end.

## 6.1 Model Selector - Evaluation

The results presented in this section are based on two papers: a conference paper [130], and a journal paper [89]. Evaluation of this work is split into three main sections, two case studies (covering image classification and machine translation) and an in-depth evaluation of different components of this approach. The two presented case studies will cover an overview of the effectiveness of a Model Selector approach on each of the DNN application domains. The in-depth evaluation will then evaluate each individual component of the Model Selector approach in isolation, analysing any individual successes or failures. For clarification, the end-to-end approach utilising a premodel is termed a Model Selector approach in this thesis. First, the two case studies are presented, starting with image classification; this section ends with the in-depth evaluation.

### 6.1.1 Case Study: Image Classification

To evaluate the Model Selector within the DNN application domain of image classification, 14 pre-trained off-the-shelf CNN models were considered (see Section 5.1.2). All considered CNN models are built and trained using TensorFlow and the ImageNet ILSVRC 2012 *training* dataset. The ImageNet ILSVRC 2012 *validation* dataset is used to generate the training data for the Model Selector, and evaluate the overall approach using k-fold

Table 6.1 All candidate features considered for the image classification case study. *Edge_length*, *edge_angle*, and *hue* features account for 7 individual features each, one feature for each bin in the histogram. This is a copy of Table 4.6, to help clarify reading.

| Feature | Description |
|---|---|
| *n_keypoints* | # of keypoints |
| *avg_brightness* | Average brightness |
| *brightness_rms* | Root mean square of brightness |
| *avg_perc_brightness* | Average of perceived brightness |
| *perc_brightness_rms* | Root mean square of perceived brightness |
| *contrast* | The level of contrast |
| *edge_length{1-7}* | A 7-bin histogram of edge lengths |
| *edge_angle{1-7}* | A 7-bin histogram of edge angles |
| *area_by_perim* | Area / perimeter of the main object |
| *aspect_ratio* | The aspect ratio of the main object |
| *hue{1-7}* | A 7-bin histogram of the different hues |

cross-validation. First the Model Selector and feature selection process is described, followed by an analysis of the overall end-to-end performance of the generated Model Selector for image classification.

### Premodel Generation

The `premodel` and feature selection process is presented in this section. In order to carry out the following steps the `premodel` training dataset was generated by exhaustively predicting every image on every candidate `CNN`; the optimal `CNN` for each image was calculated resulting in the training labels for `premodel` training. First, all candidate features are described, followed by an analysis of the feature selection process, describing which features are removed, and why. Next, the feature analysis process is presented, describing which features are further removed, and explaining why. Finally, the automatic `premodel` generation process is presented, describing the final `premodel` architecture that is used for end-to-end evaluation.

**Feature Selection.** For image classification a total of 29 candidate features were considered, shown in Table 6.1; *edge_length*, *edge_angle*, and *hue* features account for 7 individual features each, one feature for each bin in the histogram. Features were chosen based on previous image classification work [40], such as edge-based features (more edges lead to a more complex image), as well as intuition based on the motivation presented in Section 4.1.1, such as contrast (lower contrast makes it harder to see image content). Table 6.2 summarises the features removed using correlation-based feature selection, leaving just 17 features. As a reminder, the correlation-based feature selection method calculates the Pearson product-moment correlation (PCC) between each feature,

Table 6.2 Correlation values (absolute) of removed features to the features that were kept. Higher values mean the features are more correlated, up to a maximum value of 1.

| **Kept Feature** | **Removed Feature** | **Correl.** |
|---|---|---|
| *avg_perc_brightness* | *perc_brightness_rms* | 0.98 |
| | *avg_brightness* | 0.91 |
| | *brightness_rms* | 0.88 |
| *edge_length1* | *edge_length {4-7}* | 0.78 - 0.85 |
| *hue1* | *hue {2-6}* | 0.99 |

Table 6.3 The remaining image classification features after feature selection and importance analysis.

| | | | |
|---|---|---|---|
| *n_keypoints* | *avg_perc_brightness* | *hue1* | *aspect_ratio* |
| *contrast* | *area_by_perim* | *edge_length1* | |

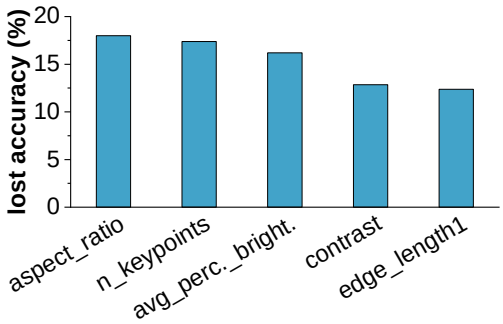
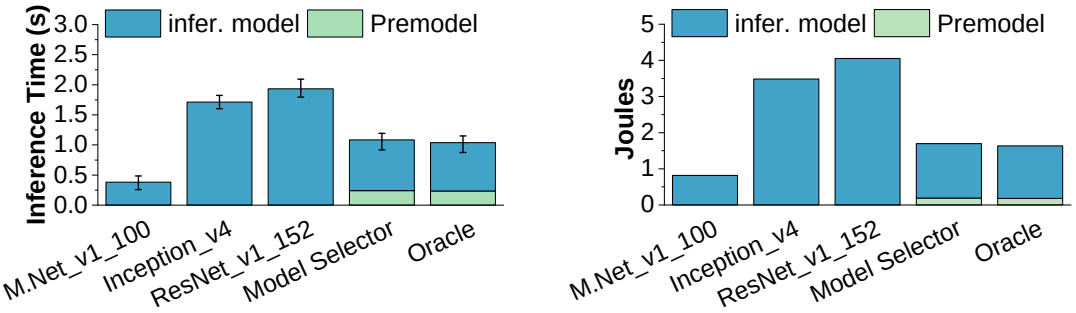



Fig. 6.1 The importance of the top 5 most important features for an image classification `premodel`. Values denote the loss in accuracy when each feature is not used in the `premodel`. Higher values show more important features.

and every other feature, yielding a value between $-1$ and 1. The closer the absolute value is to 1, the stronger the linear correlation, and therefore similarity, between the two features being tested. Next, the importance of each remaining feature is evaluated, and a greedy search performed to further reduce the feature count, resulting in 7 features remaining (Table 6.3); described in the next paragraph.

**Feature Analysis.** The importance of each remaining feature, after feature selection, is analysed next. Feature importance is calculated by first training a `premodel` using all features ($n$) and taking note of the `premodel` accuracy. Each feature is then removed, in turn, retraining and evaluating the `premodel` on the remaining $n-1$ features, noting the drop in accuracy. Figure 6.1 shows the top 5 dominant features based on their impact on their importance. It is clear that the remaining features all hold a similar level of importance, ranging between 18% and 11%, for the most and least important feature, respectively. The similarity of importance in the remaining features is an indication that




(a) Inference Time

(b) Energy Consumption

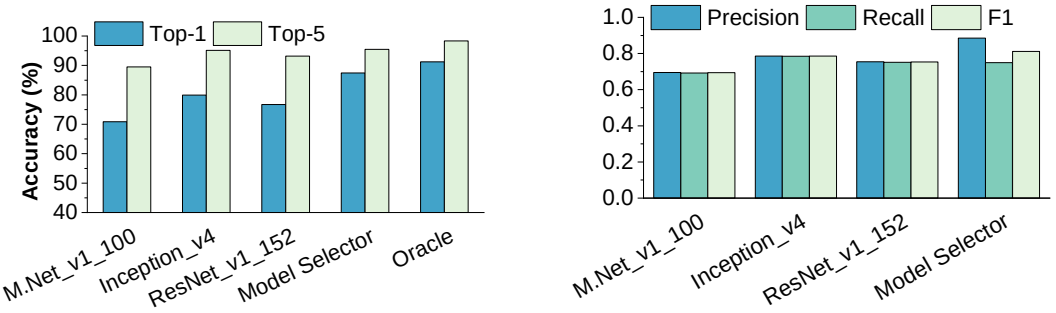
Fig. 6.2 The overall performance of individual CNN models compared to an approach using a Model Selector. Inference time (a) and energy consumption (b) of the premodel is shown when using a Model Selector. Results are presented as the average per image across all ImageNet ILSVRC *validation* set images. *lower is better*.

each feature is able to represent distinct information about an image. All remaining features, presented in Table 6.3, are important for the prediction task at hand.

**Premodel Generation.** Finally, the premodel is automatically generated using the automatic premodel generation method, described in Section 4.2.2; resulting in a premodel using a multiple classifier architecture consisting of a series of simple KNN models. Analysis of the choice found that the chosen architecture achieves a quick prediction time (<1ms) and high accuracy for this problem; meeting the two goals laid out in Section 4.2.2. Furthermore, the DNN Selection Algorithm ( 4.2.3) was applied to determine the best CNNs to include in the premodel selection choice. DNN selection algorithm parameters were set to: *selection_method* was set to 'Accuracy', and $\theta = 0.5$; see Section 6.1.3 for a sensitivity analysis of these parameters. As a result, three models were chosen: MobileNet_v1_100 was chosen for *Model-1*, Inception_v4 for *Model-2*, and ResNet_v1_152 for *Model-3*. Now the premodel architecture has been fully described, using the data generated at the start of this section a new premodel training dataset is calculated using only the three selected CNNs by the DNN Selection Algorithm. The generated premodel architecture can now be evaluated using k-fold cross-validation.

### Overall Performance

Figures 6.2 and 6.3 present the overall performance of a Model Selector on the ImageNet ILSVRC *validation* dataset. This section analyses the end-to-end performance of a Model Selector approach, using the generated premodel, in comparison with its component CNNs. The component CNNs are used for comparison as they present a range of CNN complexities, from low (MobileNet_v1_100), to mid (Inception_v4), to high (ResNet_v1_152); CNN models of similar complexities produced similar results. The following paragraphs analyse the achieved inference time, energy consumption, accuracy




(a) Accuracy Scores

(b) Precision, Recall, and F1

Fig. 6.3 The overall performance of individual CNN models compared to an approach using a Model Selector. An Oracle Model Selector is shown in (a), showing the highest achievable top-1 and top-5 scores if the premodel is 100% accurate. Results are presented as the geometric mean of 10-fold cross-validation across all ImageNet ILSVRC *validation* set images. *higher is better*.

scores, and Precision, Recall, and F1 scores of each of the approaches, before ending with a short summary.

**Inference Time.** Figure 6.2a compares the inference time of the selected CNNs against a Model Selector approach; the cost of the premodel is also included and clearly shown. The fastest CNN for inferencing is MobileNet_v1_100, running 2.8x and 2x faster than Inception_v4 and Resnet_v1_152, respectively. However, MobileNet_v1_100 is the least accuracy CNN (see Figure 6.3a). The average inference time of the Model Selector approach is around 1-second, slightly longer than the 0.4-second average of MobileNet_v1_100. A Model Selector is expected to have a larger inference time than the fastest component CNN as the cost of the premodel needs to be accounted for, alongside the cost of choosing Inception_v4 or ResNet_v1_152 on occasion. Most premodel overhead is due to feature extraction. When compared to Inception_v4, the most accurate CNN in the selected models, a Model Selector is 1.8x faster. Given that a Model Selector can significantly improve prediction accuracy in comparison to MobileNet_v1_100, the modest cost of this approach is acceptable.

**Energy Consumption.** Figure 6.2b compares the energy consumption of the selected CNNs against a Model Selector approach; the cost of the premodel is also included and clearly shown; energy consumption was captured using the method described in Section 5.3.1. On the evaluation platform chosen (NVIDIA Jetson TX2) the energy consumption is generally proportional to the model inference time. As the Model Selector approach reduces overall inference time it also reduces overall energy consumption by more than 2x when compared to Inception_v4 and ResNet_v1_152. The energy footprint of the premodel is small, it is 4x and 24x lower than MobileNet_v1_100 and ResNet_v1_152, respectively. As such, a Model Selector approach is suitable for

power-constrained devices, and can be used to improve the overall accuracy when using multiple inferencing models. Furthermore, in cases where the `premodel` predicts that none of the included `CNN` models can make a successful prediction for an image, inference can be skipped to avoid wasting power and time. Note that since the `premodel` runs on the CPU, its energy footprint ratio of the total Model Selector approach is smaller than that for inference time.

**Accuracy.** Figure 6.3a compares the top-1 and top-5 scores achieved by the selected `CNNs` against a Model Selector approach. In this case, a theoretically perfect predictor, termed `Oracle`, is also presented, showing the theoretically highest possible top-1 and top-5 scores achievable using a Model Selector approach. For clarity, the `Oracle` is a Model Selector approach with a `premodel` that theoretically achieves 100% prediction accuracy. Note that `Oracle` does not achieve 100% top-1 or top-5 scores as there are cases where all of the `CNNs` fail; however, not all `CNNs` fail on the same images. For instance, `ResNet_v1_152` will successfully classify some images that `Inception_v4` will fail on. Therefore, by effectively leveraging multiple `CNNs`, a Model Selector approach outperforms all individual `CNN` models. The Model Selector approach improves on `MobileNet_v1_100` by 16.6% and 6% in top-1 and top-5 scores, respectively. Furthermore, it also improves on the the top-1 accuracy of `ResNet_v1_152` and `Inception_v4` by 10.7% and 7.6%, respectively. Although very little improvement of the top-5 score over `Inception_v4` (just 0.34%) can be observed, the Model Selector approach is 2x faster. Finally, the Model Selector approach delivers over 96% of Oracle performance (86.3% vs 91.2% for top-1 and 95.4% vs 98.3% for top-5). Overall, a 7.52% improvement in accuracy is achieved over the most capable single `CNN` model while reducing inference time by 44.45%.

**Precision, Recall, and F1 Scores.** Finally, Figure 6.3b compares the precision, recall, and F1 scores achieved by the selected `CNNs` against a Model Selector approach. The results here are very similar to those presented in the previous paragraph, with the Model Selector approach outperforming individual `CNN` models. Specifically, the Model Selector approach achieves the highest overall precision score, leading to the overall best F1 score. High precision can reduce false positives, which is important for certain domains like video surveillance because it can reduce the human involvement for inspecting false-positive predictions. Unfortunately, unlike the selected `CNNs`, the Model Selector approach is unable to produce consistent results across precision and recall. This is likely a by-product of the model selector being optimised for top-1 accuracy.

**Conclusion.** Overall, applying a Model Selector to the image classification problem results in an improvement in performance. By implementing a `premodel` in conjunction with off-the-shelf pre-trained `CNNs`, end-to-end accuracy is improved for all metrics (top-

1, top-5, precision, recall, and F1-score), while reducing average end-to-end inference time when compared to the single most capable CNN model (ResNet_v1_152). There are some cases when the Model Selector approach will have a longer end-to-end inference time; however, such cases are few and far between, making up only 2.3% of all images. An in-depth analysis of a Model Selector approach when applied to image classification is presented in Section 6.1.3.

### 6.1.2 Case Study: Neural Machine Translation

To evaluate the Model Selector within the DNN application domain of Neural Machine Translation (NMT), 15 off-the-shelf RNN model architectures were considered (see Section 5.1.2). The *WMT09-WMT16* English-German newstest dataset [27] was used for evaluation, split into a training and validation set as described below. All considered RNN models are built and trained using TensorFlow-NMT, an NMT library provided by TensorFlow [85], and the *WMT09-WMT14* English-German newstest dataset. The *WMT15/16* English-German newstest dataset is used to generate the training data for the Model Selector, and evaluate the overall approach using k-fold cross-validation. In this thesis, RNN models using the following naming convention: gnmt_N_layer, the name is prefixed with gnmt_ where the model uses the Google Neural Machine Translation Attention [148], *N* is the number of layers in the model. For example, 4_layer is a default Tensorflow-NMT model made up of 4 layers. First the Model Selector and feature selection process is described, followed by an analysis of the overall end-to-end performance of the generated Model Selector for machine translation.

#### Model Selector Generation

The premodel and feature selection process are presented in this section. In order to carry out the following steps the premodel training dataset was generated by exhaustively predicting every sentence on every candidate RNN; the optimal RNN for each image was calculated resulting in the training labels for premodel training. First, all candidate features are described, followed by an analysis of the feature selection process, describing which features are removed, and why. Next, the feature analysis process is presented, describing which features are further removed, and explaining why. Finally, the automatic premodel generation process is presented, describing the final premodel architecture that is used for end-to-end evaluation.

**Candidate Features.** For machine translation a total of 11 candidate features were considered, shown in Table 6.4; a Bag of Words (BoW) representation of each sentence was also considered (explained in more detail in the following). Similar to image classification, features were chosen based on previous machine translation work [61,

Table 6.4 All candidate features considered for the machine translation case study.

| Feature | Description |
|---|---|
| *n_words* | # words in the sentence |
| *n_bpe_chars* | # bpe characters in a sentence |
| *avg_bpe* | Average number of bpe characters per word |
| *n_tokens* | # tokens in the sentence when tokenized |
| *avg_noun* | Average number of nouns per word |
| *avg_verb* | Average number of verbs per word |
| *avg_adj* | Average number of adjectives per word |
| *avg_sat_adj* | Average number of satellite adjectives per word |
| *avg_adverb* | Average number of adverbs per word |
| *avg_punc* | Average punctuation characters per word |
| *avg_word_length* | Average number of characters per word |

Table 6.5 Correlation values (absolute) of removed features to the features that were kept. Higher values mean the features are more correlated, up to a maximum value of 1.

| Kept Feature | Removed Feature | Correl. |
|---|---|---|
| *n_words* | *n_bpe_chars* | 0.96 |
| | *n_tokens* | 0.99 |

Table 6.6 The remaining machine translation features after feature selection and importance analysis.

| |
|---|
| *n_words* |
| *avg_adj* |
| *BoW* |

82], such as BoW, as well as intuition based on pilot experiments, such as *n_words* (longer sentences are more complex and require a more complex translator). For a BoW representation of each sentence a domain-specific vocabulary was generated based on all words in the selected training dataset, *WMT09-WMT14* English-German newstest dataset. Chi-square (Chi2) was used to perform BoW feature reduction, a widely used technique for BoW, leaving a feature vector length of 2 000; a full evaluation of the effect of BoW and Chi2 feature selection on the machine translation `premodel` is presented in Section 6.1.3. The following paragraphs present the feature selection process for all features proposed in Table 6.4.

**Feature Selection.** Table 6.5 summarises the features removed using correlation-based feature selection. At this stage just 2 features are removed, leaving 9. Next, the importance of each of the remaining 9 features is evaluated, and a greedy search performed, further reducing the feature count down to 3 (Table 6.6); described in the next paragraph.

**Feature Analysis.** The importance of each feature remaining after feature selection is analysed next. Feature importance is calculated by first training a `premodel` using all 3 features remaining after feature selection ($n$) and taking note of the `premodel` accuracy. Each feature is then removed, in turn, retraining and evaluating the `premodel` on the remaining $n-1$ features, noting the drop in accuracy. Figure 6.4 shows the accuracy

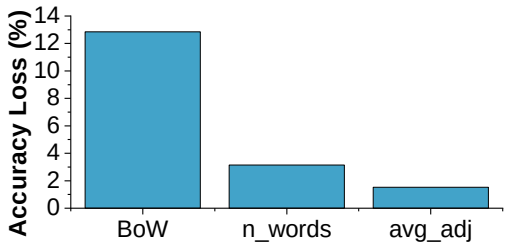


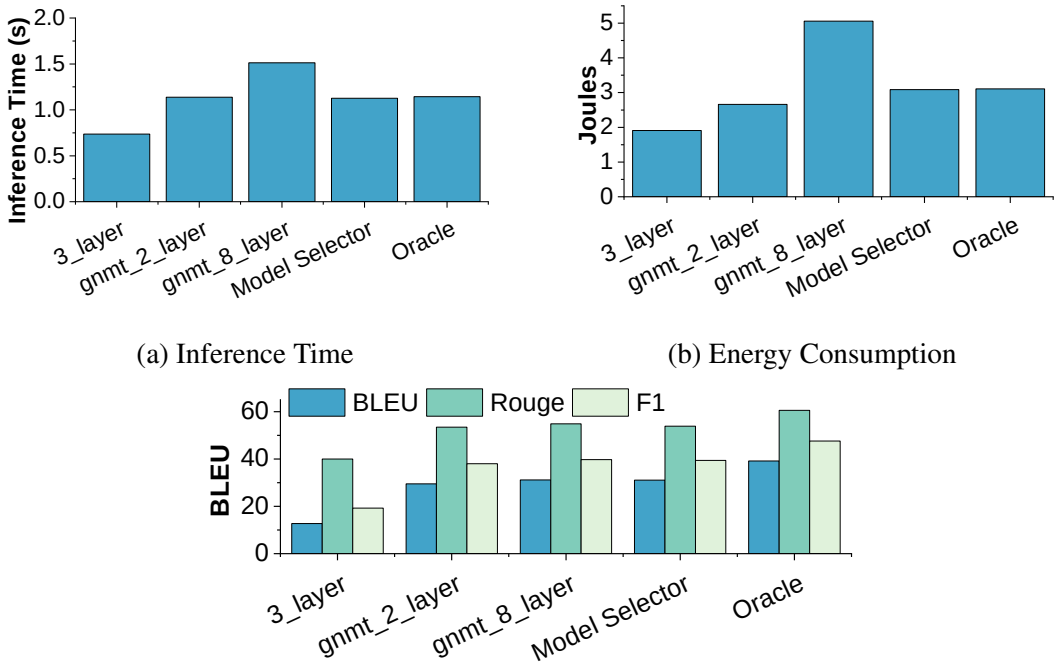

Fig. 6.4 The importance of the remaining features after feature selection and feature analysis. Values denote the loss in accuracy when each feature is not used in the `premodel`. Higher values show more important features.

loss by removing any of the three selected features shown in Table 6.6. It is clear that including BoW as a feature significantly increases `premodel` accuracy; this is to be expected, as BoW is a well-researched and utilised representation of text input. By removing either *n_words* or *avg_adj*, there is a small drop in accuracy; this indicates that BoW is able to capture similar information. Both features are kept as they bring a small increase to accuracy with negligible overhead. All remaining features, presented in Table 6.6, are important for the prediction task at hand.

**Premodel Generation.** Finally, the `premodel` is automatically generated using the automatic `premodel` generation method, described in Section 4.2.2; resulting in a `premodel` using a single classifier architecture using a Naive Bayes (`NB`) model. It is likely that a single classifier architecture `premodel` was chosen in this problem domain because of the reduced dataset, *i.e.*, only one tenth of the training data compared to image classification. Analysis of the `premodel` choice found that the chosen architecture achieves a quick prediction time (<1ms) and high accuracy for this problem; meeting the two goals laid out in Section 4.2.2. When applying the `DNN` selection algorithm *selection_method* was set to 'Accuracy', and $\theta = 2.0$; see Section 6.1.3 for a sensitivity analysis of these parameters. As a result, three models were chosen: `gnmt_2_layer`, `gnmt_8_layer`, and `3_layer` for *Model-1*, *Model-2*, and *Model-3*, respectively. Now the `premodel` architecture has been fully described, using the data generated at the start of this section a new `premodel` training dataset is calculated using only the three selected `RNNs` by the `DNN` Selection Algorithm. The generated `premodel` architecture can now be evaluated using k-fold cross-validation.

### Overall Performance

Figure 6.5 presents the overall performance of a Model Selector on the The *WMT15/16* English-German newstest dataset. This section analyses the end-to-end evaluation results a Model Selector approach, using the generated `premodel`, when compared to its component `RNNs`. The component `RNNs` are used for comparison as they present a range of `RNN`




(a) Inference Time

(b) Energy Consumption

(c) BLEU, Rouge, and F1 Measure

Fig. 6.5 The overall performance of individual `RNN` models compared to an approach using a Model Selector. Inference time (a) and energy consumption (b) of the `premodel` when using a Model Selector is negligible, therefore cannot be seen. Results are presented as the average per sentence across all *WMT15/16* English-German newstest dataset. In (a) and (b) *lower is better*. *Higher is better* in (c).

complexities and capabilities, from low (`3_layer`), to high (`gnmt_8_layer`). Furthermore, the Model Selector approach is compared against an `Oracle`, a theoretical perfect `premodel` approach that achieves the best possible score for each evaluation metric. The following paragraphs analyse the results of inference time, energy consumption, and accuracy scores.

**Inference Time.** Figure 6.5a compares the inference time of the selected `RNNs` against an `Oracle` and Model Selector approach. The fastest `RNN` for inferencing is `3_layer`, running 1.55x faster than `Oracle` and 2.05x faster than the most complex individual `RNN`, `gnmt_8_layer`; however, `3_layer` is the least accurate `RNN` (see Figure 6.5c). A similar inference time is achieved by by the `Oracle`, Model Selector approach, and `gnmt_2_layer`; nonetheless, `gnmt_2_layer` is outperformed in terms of accuracy. The runtime of the Model Selector approach `premodel` and feature extraction is negligible, consisting of less than 1 ms for the `premodel` and less than 5 ms for feature extraction, per sentence. Feature extraction and `premodel` overheads are included in the inference time of the Model Selector approach and Oracle. Incidentally, the Model Selector approach is slightly quicker than the `Oracle`; this is as a result of the `premodel` often mispredicting `gnmt_2_layer` for `gnmt_8_layer` and vice versa. This specific misprediction makes

up 38.5% of the cases where premodel makes an incorrect prediction. To improve accuracy, more data is required to train the premodel in order to reduce the high feature to sentence ratio. Alternatively, a deep investigation into the sentences that are best for each RNN could intuitively reveal a new feature to add to the premodel; however, the differences may not be intuitive to spot. Overall, the Model Selector approach is 1.34x faster than the single most capable RNN without a decrease in accuracy.

**Energy Consumption.** Figure 6.5b compares the energy consumption of the selected RNNs against an Oracle and Model Selector approach. The negligible cost of the premodel (see Section 6.1.3) is included in Oracle and Model Selector approaches. Much like the image classification results, energy consumption is proportional to model inference time for machine translation; therefore, overall inference time is reduced, energy efficiency is improved. A major difference between energy consumption and inference time is the emphasised ratios between each model – for example, gnmt_2_layer is 1.24x quicker than gnmt_8_layer, but it uses 1.90x less energy, nearly half as much. Overall, the Model Selector approach uses 1.39x less energy on average than the single most capable model, without a significant change in F1 measure. Therefore, a Model Selector approach is suitable for power-constrained devices, it can be used to improve energy efficiency while having little impact on accuracy or, in some cases, seeing an improvement in accuracy. Furthermore, in cases where the premodel predicts that none of the included RNN models can make a successful prediction for a sentence, inference can be skipped to avoid wasting power and time. Implementing such a condition results in using 1.48x less energy on average than the single most capable RNN, gnmt_8_layer.

**BLEU, Rouge, and F1-Measure.** Finally, Figure 6.5c compares the BLEU, Rouge, and F1 measure scores achieved by the selected RNNs against an Oracle and Model Selector approach. The results here are very similar to those presented in the previous paragraph, with the Model Selector approach outperforming individual RNN models; the results are consistent across accuracy measures. As all RNNs do not fail on the same sentences, a higher overall F1 measure is achieved by leveraging multiple RNNs. An Oracle achieves an F1 measure of 47.54, a 20% increase over gnmt_8_layer, the single most capable RNN, which achieves 39.71. The Model Selector approach achieves 83% of the Oracle F1 measure. Overall, the Model Selector approach achieves approximately the same F1 measure as the single most capable model and improves upon the accuracy of gnmt_2_layer (the closest single RNN in terms of inference time) by 4%. For the Model Selector premodel to achieve its full potential, as shown by the Oracle, more data is required for premodel training and testing.

**Conclusion.** Overall, applying a Model Selector to the machine translation problem results in an improvement in performance. By implementing a premodel in conjunction

with off-the-shelf pre-trained RNNs, end-to-end accuracy is maintained while reducing average inference time by 25.4% when compared to the single most capable RNN model (gnmt_8_layer). In this case, the Model Selector approach fails to reach the potential of an Oracle premodel; this is likely due to the lack of training data for the premodel- 10x less data is available when compared to image classification. An in-depth analysis of a Model Selector approach when applied to machine translation is presented in the next section.

### 6.1.3 In-Depth Analysis

This section analyses the working mechanisms of a Model Selector approach to DNN inference, aiming to further explain the characteristics of the approach. First, the automatic premodel generation method is analysed, showing it is effective at choosing the best premodel architecture. Next, the importance of all candidate features are analysed, followed by an in-depth analysis of the training and deployment overhead of a Model Selector approach. A soundness analysis of the premodel effectiveness is then reported. Finally, a number of smaller in-depth analyses are carried out.

#### Alternate Premodel Architectures

An automatic method for choosing the premodel architecture was designed for this work as the optimal premodel architecture can change depending on the problem domain, as shown across the two case studies presented in Sections 6.1.1 and 6.1.2. This section aims to show the proposed method is effective by analysing the choices made by the automatic method for both case studies. First, the notation used in this section is explained, followed by an analysis of the premodel architectures for image classification, and then machine translation.

**Notation.** In this section, a simple notation is used to concisely describe the premodel architecture being discussed. Single classifier architectures are simply described using their name or abbreviation from the following:

- **KNN** K-Nearest Neighbours
- **SVM** Support Vector Machine
- **DT** Decision Tree
- **NB** Naive Bayes
- **CNN** Convolutional Neural Network

Multiple classifier architecture premodel configurations are denoted as $X.Y.Z$, where $X$, $Y$, and $Z$ indicate the classifier for the first, second, and third level of the premodel,

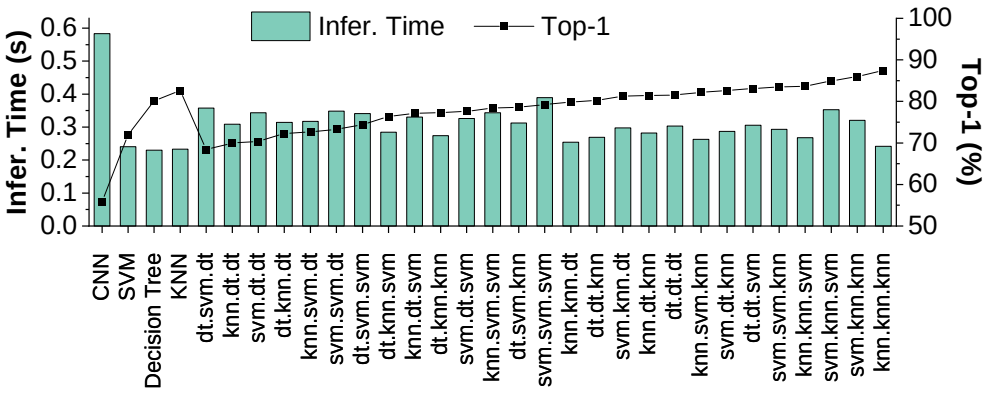



Fig. 6.6 A comparison of alternate `premodel` architectures for image classification. Results are reported as end-to-end performance. Inference time is reported as the average per-image, and top-1 score is the geometric mean of 10-fold cross-validation. *Lower is better* for inference time. *Higher is better* for top-1 score.

respectively. For example, `KNN.SVM.KNN` denotes using a `KNN` classifier for the first and third levels, and an `SVM` classifier for the second level.

**Image Classification.** Figure 6.6 shows the average inference time and top-1 accuracy of different `premodel` architectures when evaluated end-to-end on the image classification problem. Here, each `premodel` architecture is learning to predict which of the `CNNs`, `MobileNet_v1_100`. `Inception_v4`, or `ResNet_v1_152`, to use; the `premodel` can also predict that all of the candidate `CNNs` will fail. The chosen `CNN` is then used to make a prediction on the input image. A total of 31 `premodel` architectures were considered, consisting of: every combination of `KNN`, `SVM`, and `DT`, to make a multiple classifier architecture `premodel` (27); 3 single classifier architecture `premodels` using the same three classifiers; and a `CNN`-based `premodel`. The `CNN`-based `premodel` is based on the MobileNet architecture, as it has been designed for embedded inference; an automated hyper-paramter tuner [63] was used to optimise the training parameters, and the model was trained for more than 500 epochs.

**Results.** Given the high runtime overhead of a `CNN` `premodel`, the results are unexpectedly disappointing. It is likely that the poor performance of the `CNN` `premodel` is due to a lack of training data, `CNNs` typically require significantly more training data than alternate `SML` methods. Single architecture `premodels` based on `KNN`, `DT`, and `SVM` all have very similar overheads; however, `KNN` is slightly faster and out-performs the other two in terms of top-1 accuracy, achieving 82.6%. In this case, multiple classifier architecture `premodels` introduce a small overhead over their single classifier architecture counterparts, while increasing end-to-end top-1 accuracy. `SVM.SVM.SVM` increases top-1 accuracy over `SVM` by 11.4%, `KNN.KNN.KNN` increases top-1 accuracy over `KNN` by around 5%, and `DT.DT.DT` increases top-1 accuracy over `DT` by just 1.4%. It is clear that the `premodel`

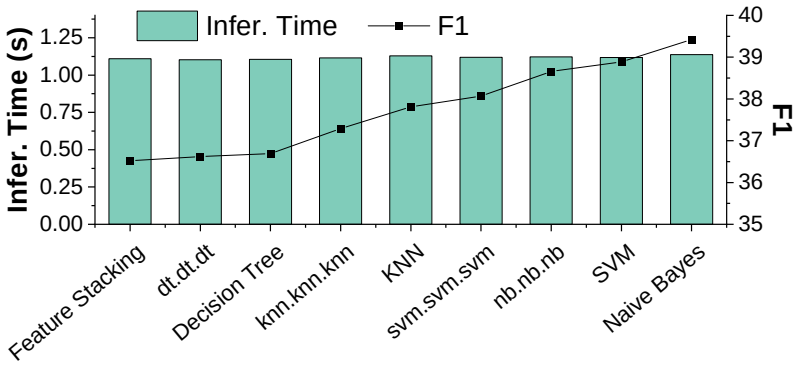



Fig. 6.7 A comparison of alternate `premodel` architectures for machine translation. Results are reported as end-to-end performance. Inference time is reported as the average per-sentence, F1-measure is reported as the geometric mean of 10-fold cross-validation. *Lower is better* for inference time. *Higher is better* for F1-measure.

architecture chosen by the designed automatic `premodel` generation tool (`KNN.KNN.KNN`) is the best choice. It achieves the highest top-1 score of 87.4%, and a fast runtime of 241ms; only the single architecture `premodels` are marginally faster (around 10ms). By utilising a `KNN` classifier at each level of the `premodel` a small optimisation can be made, performing the neighbouring measurement only once, and sharing the results across the `KNN` model at each level. With this optimisation, the runtime overhead is nearly constant if `KNN` classifiers are used across all hierarchical levels. The `KNN` classifiers in the `KNN.KNN.KNN` architecture achieve prediction accuracies of 95.8%, 80.1%, and 72.3%, for the first, second, and third levels, respectively.

**Machine Translation.** Figure 6.7 shows the average inference time and F1-measure of different `premodel` architectures when evaluated end-to-end on the machine translation problem. In this instance, each `premodel` is learning to predict which of the `RNNs`, `gnmt_2_layer`, `gnmt_8_layer`, or `gnmt_3_layer`, to use; the `premodel` can also predict that all of the candidate translators will fail. The chosen `RNN` is the used to translate the input sentence. Based on the results from image classification, it was discovered that the best performance was often achieved by using the same classifier for each component. Therefore, a less exhaustive search was carried out in this case. Both single and multiple classifier architectures have been evaluated for `KNN`, `DT`, `SVM`, and `NB`. Finally, an alternate approach named Feature Stacking [82] was also included, bringing the total up to 9 `premodel` architectures. Using feature stacking, `premodel` selection is split into two classifiers, one using the BoW features and the other using the remaining features; a probability measure of prediction is then used to choose the predicted `RNN`.

**Results.** For machine translation, it is clear that the single classifier architecture of `premodels` always outperforms its multiple classifier architecture alternative. This is likely due to the high-dimensional feature space, with a comparatively low training

set size. In this case, a multiple classifier architecture premodel would be unwise. Feature stacking performed poorly for this problem; in fact, it performs worse than all other considered premodel architectures, indicating that the chosen features work better together. Overall, there is little variance in the inference time of each approach; every premodel architecture achieves an inference time between 1100 and 1140 ms. It is clear that the premodel architecture chosen by the designed automatic premodel generation tool (NB) is the best choice. It achieves the highest overall F1-measure, while achieving a very similar average inference time to all other considered approaches.

#### DNN Selection Algorithm Sensitivity Analysis

Section 4.2.3 describes the created algorithm that is used when deciding which DNNs to include in the premodel selection. This section analyses the tunable parameters of the DNN selection algorithm, and how they effect the resultant end-to-end performance. The results are presented for only the image classification problem as the results are very similar for both image classification and machine translation. For this analysis, the performance is presented as if a perfect predictor was implemented for the premodel, that is, a premodel that is able to correctly predict the best DNN to use for every single input. A perfect predictor is used prevent the premodel accuracy from introducing any noise, allowing for a much clearer analysis of the DNN selection algorithm in isolation. All three valid *selection_method* choices are considered alongside 4 different choices for $\theta$: 5.0, 2.0, 1.0, and 0.5. Every combination of *selection_method* and $\theta$ is considered for a total of 12 parameter configurations.

**Notation.** In this section, a simple notation is used to concisely describe the DNN selection algorithm parameter configuration being discussed. Parameter configurations are denoted as *selection_method*-$\theta$, where *selection_method* is one of: *Accuracy*, *Optimal*, or *Alternate*; and $\theta$ is the numeric threshold parameter. For example, the notation Accuracy-5.0 denotes that the most accurate DNN is selected in each iteration of the DNN selection algorithm, the iterations stop when the accuracy improvement is less than 5.0%.

**Results.** Figure 6.8 shows the effect of different DNN selection algorithm parameters on the end-to-end performance of the image classification problem. As a general rule, as the value of $\theta$ decreases (making the DNN selection algorithm more tolerable), the number of included DNNs increases. For example, for configurations Alternate-5.0 to Alternate-0.5, 3, 4, 5, and 7 DNNs are chosen, respectively. Increasing the number of DNNs can have a positive effect, increasing overall top-1 accuracy; however, there are some drawbacks. More DNNs leads to the premodel needing to make a more complex choice, possibly reducing overall accuracy depending on the premodel accuracy. Furthermore, each additional DNN will need to be held in system memory, this could be an issue for devices with limited memory; resource usage is discussed in more detail in Section 6.1.3. It is

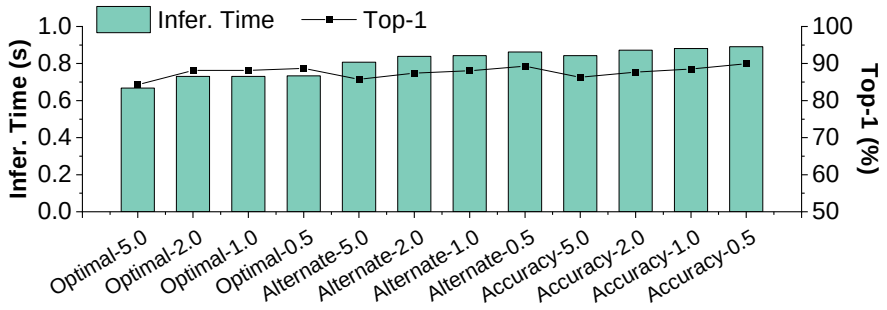



Fig. 6.8 An comparison of the effect of different DNN selection algorithm parameters on premodel performance. Results are reported as end-to-end performance. Inference time is reported as the average per-image, top-1 score is reported as the geometric mean of 10-fold cross-validation. *Lower is better* for inference time. *Higher is better* for top-1 scoring.

worth noting that there is no change in DNN selection from Optimal-2.0 to Optimal-1.0, as the next DNN to be added only only brings an accuracy improvement of 0.508, meaning it will not be included until $\theta = 0.5$. Finally, each *selection_method* displays its own 'profile' – that is, each has its own positive and negative impact. *Optimal* results in an overall faster average inference time; however, it has a lower top-1 accuracy than the other approaches. *Accuracy* is able to achieve the highest possible top-1 accuracy, but this comes at the cost of speed, achieving a slowdown of 1.26x for a 2% increase in accuracy. *Alternate* attempts to find a balance between the other two approaches; it is able to achieve an accuracy and average inference time between *Optimal* and *Accuracy*.

### Feature Importance

As part of this work, an automatic feature selection tool was created to decide which features are best to include in the premodel when presented with a number of candidate features. This section analyses the choices made by the automatic feature selection tool on both of the case studies presented by analysing the importance of all features. Feature importance is calculated by first training a premodel using all features ($n$) and taking note of the premodel accuracy. Each feature is then removed, in turn, retraining and evaluating the premodel on the remaining $n-1$ features, noting the drop in accuracy. First, the image classification features are analysed, followed by the machine translation features.

**Image Classification.** The automatic feature selection process, when applied to the candidate image classification features resulted in seven features being used represent each image for the premodel; described in detail in Section 6.1.1. Figure 6.9 shows the importance of all features remaining after the correlation check (see Table 6.2), sorted in order of importance. The first seven features are the most important, they are the

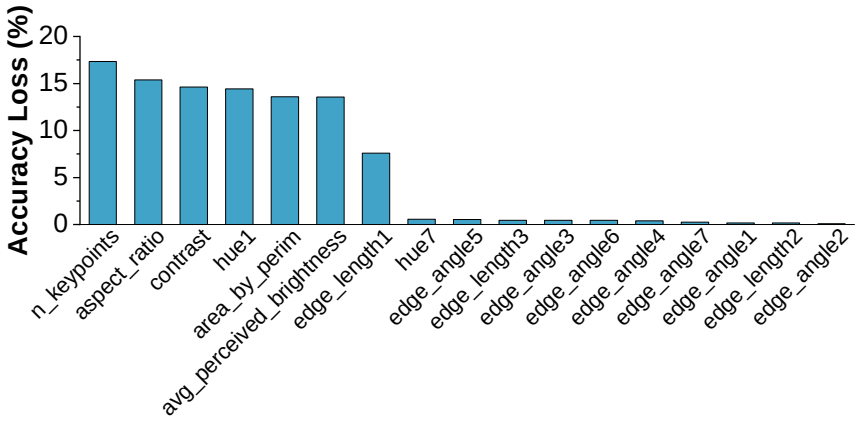



Fig. 6.9 The importance of all candidate features remaining after a correlation check for an image classification `premodel`. Values denote the loss in accuracy when each feature is not used in the `premodel`. Higher values show more important features.

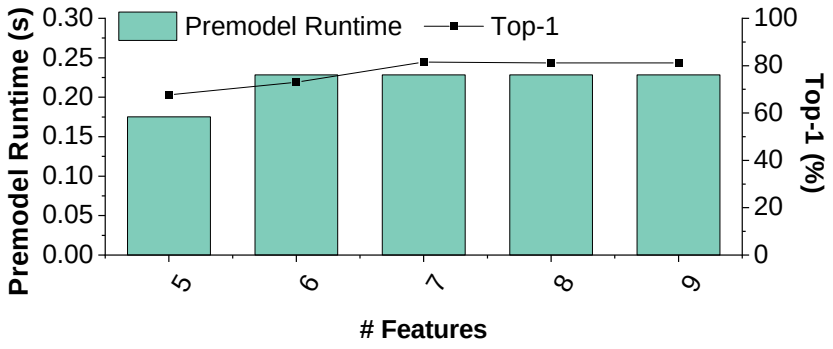



Fig. 6.10 The `premodel` runtime cost and end-to-end top-1 score as the number of features included in the `premodel` changes. End-to-end top-1 score is presented as the geometric mean of 10-fold cross-validation, and `premodel` runtime is presented as the average for each input image. *Lower is better* for `premodel` runtime, *higher is better* for top-1 score.

final chosen features for image classification (see Table 6.3), as expected. There is also a clear sudden drop at feature 8 (*hue7*), emphasising that feature removal during the feature importance analysis stopped at the correct iteration. Furthermore, Figure 6.10 shows how the change in feature count effects the end-to-end top-1 score performance, and the runtime overhead of the `premodel`. The feature count changes by adding or removing the most or least important feature, respectively. When decreasing the number of features there is a dramatic decrease of end-to-end top-1 accuracy score, with very little impact on `premodel` overhead; in fact there is no decrease in `premodel` overhead between 7 and 6 features. In order to reduce `premodel` overhead, the feature count needs to be reduces to 5; however this brings just 50ms of savings in overhead at the cost of 13.9% end-to-end top-1 accuracy. Increasing the feature count has very little impact on the `premodel` runtime overhead, this is due to the fact that the features being added are

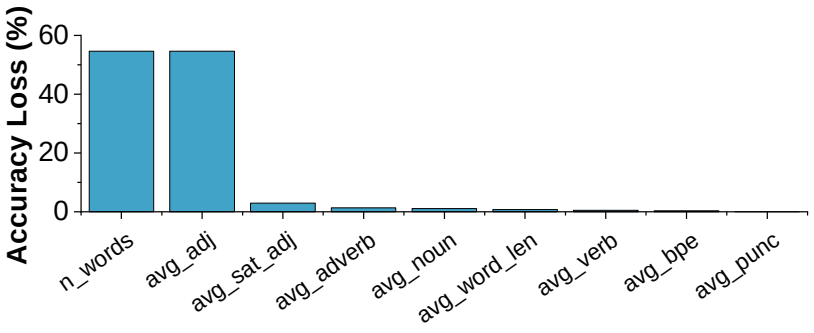



Fig. 6.11 The importance of all candidate features remaining after a correlation check for a machine translation `premodel`. Values denote the loss in accuracy when each feature is not used in the `premodel`. Higher values show more important features.

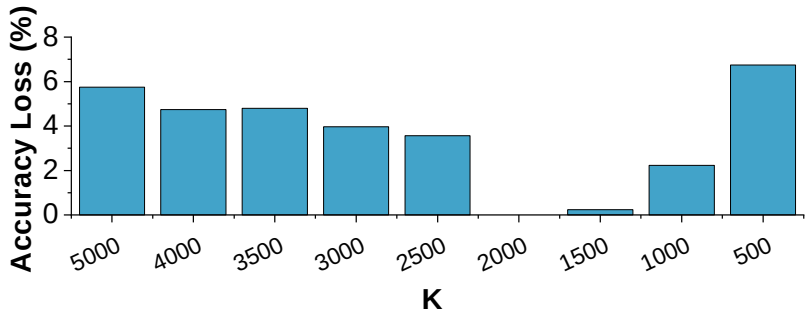



Fig. 6.12 The `premodel` accuracy loss when using different values of $k$ for a BoW representation of sentences; $k$ represents the number of words in the BoW vocabulary. $k = 2000$ is used as the baseline, as that is the value chosen in Section 6.1.2. In this case, *lower is better*.

*hue7* and *edge_angle5*, which are calculated at the same time as *hue1* and *edge_length2* (which are already included), respectively. Surprisingly, increasing the feature count actually introduces a small decrease in top-1 accuracy, this is likely due to the `premodel` making some mis-predictions. From this, it is clear that using seven features is ideal.

**Machine Translation.** Here, the `premodel` features for the machine translation case study are analysed. In this case, the BoW features are analysed separately to allow for a clear analysis of the choices made. The automatic feature selection process, when applied to the candidate machine translation features resulted in just two features (plus a BoW representation) being used represent each input sentence for the `premodel`; described in detail in Section 6.1.2. Figure 6.11 shows the importance of all features remaining after the correlation check (see Table 6.5), sorted in order of importance. It is clear that the first two features are far more important then all the others, as expected; it is clear that removing either feature severely deteriorates the accuracy of the `premodel`. Including *avg_sat_adj* results in a 2.9% increase in `premodel` accuracy; however it is left out because it provides negligible improvements when used in conjunction to BoW.

**Bag of Words.** Applying a Model Selector to machine translation requires classification of each individual sentence in order to predict its optimal `RNN`. Text classification is a notoriously difficult task, the difficulty is only exacerbated when only a single sentence is available to gather features from. It is possible to create a successful `premodel` using only the candidate features presented in Table 6.4; however, with the addition of a BoW representation of each sentence, `premodel` accuracy increases. Furthermore, previous work in sentence classification [61, 82, 87] often makes use of a BoW representation, suggesting that BoW can be useful for characterising and modelling a sentence. A BoW representation of text describes the occurrence of words within the text, represented as a vector based on a vocabulary. Here, the BoW representation used in Section 6.1.2 is analysed.

Figure 6.12 analyses the effect different values of $k$ has on the `premodel` accuracy; $k = 2000$ has been used as a baseline. The chi-squared test is used to test and evaluate each row of the BoW vector, the top $k$ rows are then chosen. It is clear that $k = 2000$ was the correct choice, choosing a value greater than 2000 results in a dramatic loss in accuracy (nearly 4%), which quickly increases to 5.75% as $k$ increases. The loss in accuracy is likely due to the `premodel` over-fitting as $k$ increases due to the large feature count. Setting $k = 1500$ results in a very small loss of `premodel` accuracy (just 0.23%); however reducing it further leads to much larger losses in accuracy, up to 6.75%. In this case, the loss in `premodel` accuracy is likely due to $k$ values less than 1500 being unable to capture all of the information necessary for accurate predictions. Therefore, the optimal value of $k$ sits around the 2000 to 1500 mark. As the overhead of increasing $k$ is negligible and $k = 2000$ achieves slightly better `premodel` accuracy, it is the best choice.

### Training and Deployment Overhead

In order to implement a Model Selector approach for any `DNN` domain a training and deployment overhead is introduced, this section analyses theses costs. Training of a `premodel` is a one-off cost that is dominated by the time taken for `premodel` training data generation. In total `premodel` training took less than a day on the GPU-server; `premodel` training data generation could be spread across multiple machines in order to speed up total `premodel` training time. When compared to the typical training time of a `DNN`, the training of a `premodel` is negligible. Due to a Model Selector approach trading off RAM space for improved accuracy and reduced inference time, an evaluation of resource utilisation is provided in Section 6.1.3. In addition to the case studies presented in Sections 6.1.1 and 6.1.2, `premodel` runtime overhead was analysed for object detection using the COCO dataset [80]; the runtime overhead makes up less than 13.5% of the average inference time per input.

**Image Classification.** For image classification, the runtime overhead of a `premodel` is minimal, as depicted in Figure 6.2a. The total average execution time per image is just less than 1 second, the `premodel` accounts for 28% of this time. In comparison, this is 12.9% and 71.7% of the average execution time of the most (`ResNet_v1_152`) and least (`MobileNet_v1_100`) expensive models, respectively. Furthermore, the deployment overhead in terms of energy consumption is smaller, making up just 11% of the total cost on average, per image. In comparison to the most and least expensive models that is an overhead of 7% and 25% of their costs, respectively. The deployment overhead of a `premodel` is dominated by the cost of feature extraction from images; therefore, the overhead can be reduced by optimising feature extraction.

**Machine Translation.** Feature extraction costs for machine translation are significantly cheaper when compared to image classification, resulting in much smaller overheads in this `DNN` domain. On average, the `premodel` accounts for just 0.5% (less than 6ms) of the end-to-end time taken to translate a sentence. Similarly, the energy cost of the `premodel` accounts for just 0.48% of the overall energy cost. The memory footprint of the `premodel` is also insignificant.

### Soundness Analysis

Inherently, it is possible that a `premodel` will provide an incorrect prediction for any input. In other words, the `premodel` could choose either a `DNN` that gives an incorrect result or a more expensive `DNN`. This section analyses the possibility of such an event happening in order the provide a soundness guarantee on the prediction ability of a `premodel`. Theoretical proof of soundness guarantee of machine learning models is an outstanding challenge and is outside the scope of this thesis [5]. Due to the difficulty in soundness guarantee, results are only presented for image classification. Nonetheless, there are two possible ways to empirically estimate the prediction confidence: (1) using the distance on the feature space as a soundness measurement, or (2) using statistical assessments. Both methods are described below, in turn.

**Distance Measurement.** Figure 6.13 shows how the end-to-end top-1 accuracy score is affected as the permissible distance for choosing the nearest training images changes. Recall that each training image is associated with an optimal model for that image, and by choosing the nearest training images to the input, a voting scheme can be used to determine which of the associate `DNNs` to use for the input image. Here, the distance is calculated by computing the Euclidean distance between the input testing image and a training image on the feature space. The results are averaged across all testing images using 10-fold cross-validation. From 0 there is a steep increase in end-to-end top-1 score, before reaching a peak when the radius is equal to 2. The increase in accuracy is due to short distances reducing the chances of finding a testing image that is close enough,

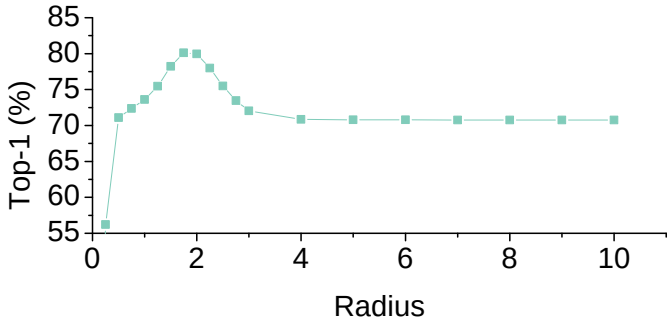



Fig. 6.13 The change in end-to-end top-1 accuracy when changing the radius of considered values in a `KNN` based `premodel`. As radius increases, each input will consider a wider radius of points when choosing which label it belongs to. In this case, *Higher is better*.

leading to a default prediction indicating that all `premodel DNNs` will fail. However, when the permissible distance is increase beyond 2, the end-to-end top-1 score begins to drop until it plateaus around 72%. The drop in end-to-end top-1 accuracy is likely due to more training images being considered for each input, introducing noise, and an incorrect selection. This example shows that the permissible distance can be empirically determined and used as a proxy for the accuracy confidence.

**Statistical Assessment.** Another method for soundness guarantee is to combine probabilistic and statistical assessments. This can be done by using a Conformal Predictor (CP) [117] to determine to what degree a *new, unseen* input conforms to previously seen training samples. The CP is a statistical assessment method for quantifying how much a model's prediction can be trusted. This is achieved by learning a nonconformity function from the model's training data. This function estimates the "strangeness" from input features, $x$, to a prediction output, $y$, by looking at the input and the probability distribution of the model prediction. Specifically, a nonconformity function, $f$, is learned from the `premodel` training dataset, which produces a nonconformity score for the `premodel`'s input $x_i$ and output $y_i$, formalised as:

$$f(x_i, y_i) = 1 - \hat{P}_h(y_i|x_i) \tag{6.1}$$

Here, $\hat{P}_h$ is the statistical distribution of the `premodel`'s probabilistic output, calculated as:

$$p_{x_i}^{y_i} = \frac{|\{z_j \in Z : a_j > a_i^{y_i}\}|}{q+1} + \theta \frac{|\{z_j \in Z : a_j = a_i^{y_i}\}| + 1}{q+1}, \theta \in [0,1] \tag{6.2}$$

where $Z$ is part of the training dataset chosen by the CP, $q$ is the length of $Z$, $a_i$ is the calibration score learned from training data, $a_i^{y_i}$ is the statistical score for `premodel` prediction $y_i$, and $\theta$ is a calibration factor learned by the CP.

The learned function $f$ produces a non-conformity score between 0 and 1 for every class for each given input. The closer the score to 0, the more likely the input is to

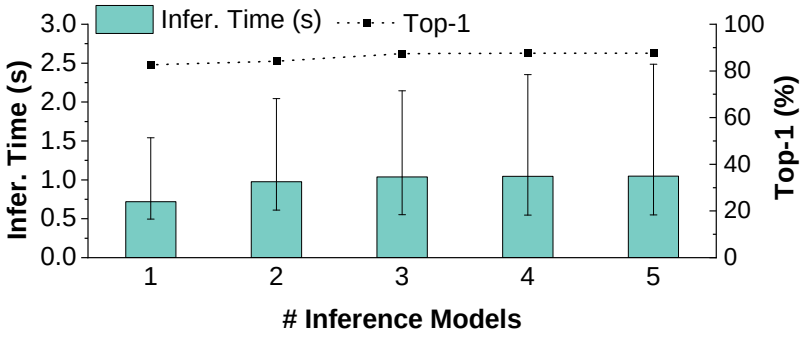



Fig. 6.14 The change in end-to-end top-1 accuracy and average inference time per image when changing the number of DNNs included in the premodel. Values are reported as the geometric mean of 10-fold cross-validation. *Higher is better* for top-1 score, *lower is better* for inference time.

conform to the premodel's output, *i.e.* the input is similar to training samples of that class. By choosing a threshold, it is possible to predict whether the premodel will give an incorrect DNN for a given input, comparing this prediction to the actual results the accuracy of the CP can be calculated. By implementing an SVM based conformal predictor for image classification, and using a threshold value of 0.5, it is possible to correctly predict when the premodel will choose an incorrect DNN 87.4% of the time; achieving a false positive rate of 5.5%. This experiment shows that a CP can be used to estimate if the premodel's output can be trusted in order to provide a certain degree of soundness guarantee.

### Further In-Depth Analysis

This section presents a few short in-depth analyses of different aspects of the Model Selector approach. The analyses are reported for the image classification case study only; when applied to machine translation very similar results are observed. First, the effects of altering the number of DNNs included in the premodel is analysed, followed by an analysis of the system resource utilisation. Finally, a small study is presented on how a Model Selector can be used in conjunction with DNN compression to produce faster overall inference times.

**Varying the Premodel Size.** Section 4.2.3 describes the DNN selection algorithm, that is, the method that is used to select which DNNs to include in a premodel. Using *Accuracy* as the *selection_method*, and temporarily ignoring the threshold parameter $\theta$ in Algorithm 1, Figure 6.14 was created. Figure 6.13 compares the end-to-end top-1 accuracy and average inference time per image when including up to five DNN models. As the number of DNN inference models increases, the average end-to-end inference time per image also increases, this is due to more expensive DNN inference models being

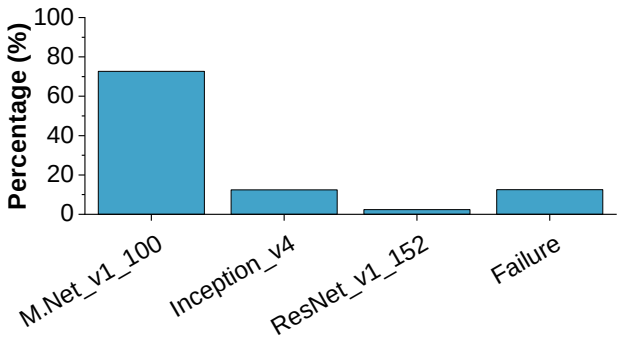
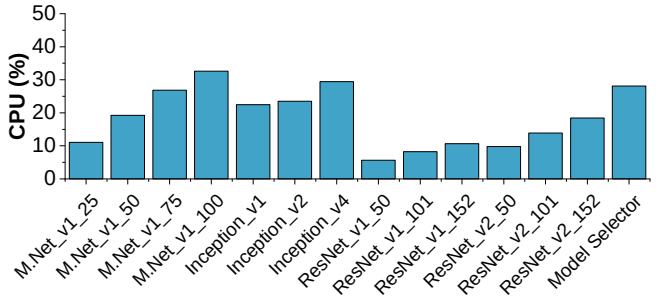



Fig. 6.15 The utilisation percentage of each DNN included in the image classification premodel. Values should be read as the percentage of test images that were predicted to have each DNN as its optimal model. The total of all values adds up to 100%.

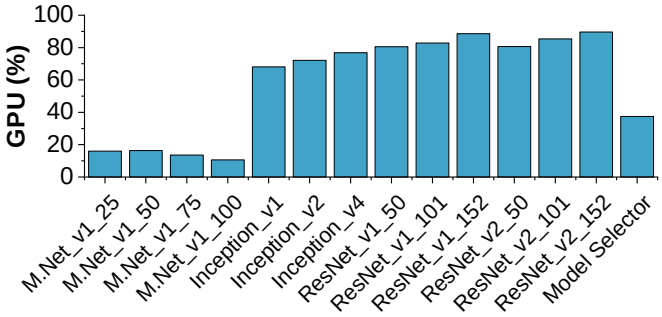
included and chosen more often. At the same time, however, the top-1 accuracy reaches a plateau of $\approx 87.5\%$ when just three DNN inferencing models are used. The optimal solution in this case is to used three DNN inferencing models, as there is no longer an accuracy increase to justify the costs of using more; this is in line with the choice of $\theta = 0.5$ during Section 6.1.1. Additionally, Figure 6.15 shows the utilization percentage of each DNN inferencing model when using the premodel described in Section 6.1.1. The Model Selector approach can also choose to not select any of the DNN inferencing models for an input image if it deems that none of the available models are suitable; the label Failure is used to represent this case. Overall, 87.5% of the time a model is selected, with MobileNet_v1_100 making up the majority of it (72.7%), leaving 12.5% of the time Failure is selected.

**Resource Utilisation.** Figure 6.16 shows the average CPU, GPU, and memory utilisation of all candidate CNNs for image classification against the Model Selector approach implemented in Section 6.1.1. The reported values are the averages across every single image in the testing dataset (ImageNet ILSVRC 2012 *validation* dataset). Each resource is briefly discussed in turn below.

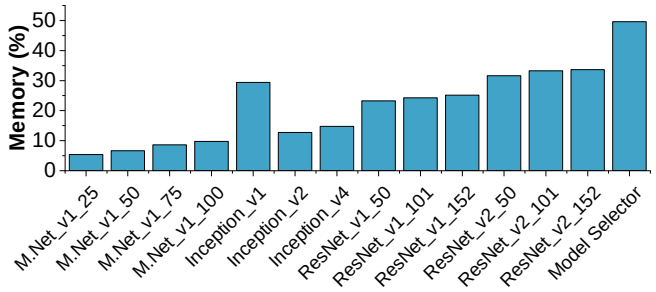
- **CPU.** Figure 6.16a shows the CPU utilization. All DNNs primarily run on the GPU, hence the low CPU utilization overall; no DNN has a CPU utilisation higher than 30%. The Model Selector approach is one of the most expensive, using 28.11% of the CPU; it is only cheaper than MobileNet_v1 and Inception_v4, which use 32.63% and 29.42%, respectively. In this category, the Model Selector approach is expensive, as it include the two most expensive models in terms of CPU utilisation.
- **GPU.** Figure 6.16b shows the GPU utilisation. As expected, the GPU utilisation is much higher in comparison to CPU utilisation as the GPU is the primary processor for DNNs. The majority of DNNs use between 70% and 90% of the GPU. In contrast, the Model Selector approach has a much lower average utilisation of 37.46%,




(a) CPU Utilisation




(b) GPU Utilisation




(c) Memory Utilisation

Fig. 6.16 The average CPU, GPU, and memory utilisation of the implemented Model Selector approach compared against each individual candidate `CNN`. The values are presented as the average utilisation across every test image in a 10-fold cross-validation. *Lower is better* for all figures.

which is 52.18% lower than the most expensive model (`ResNet_v2_152`), which a Model Selector approach outperforms. This is achieved by utilising `MobileNet_v1` whenever possible, which has an average GPU utilization of 10.57%, bringing the Model Selector approach much lower.

- **Memory.** Finally, Figure 6.16c compares the memory utilisation of a Model Selector approach against individual `DNNs`. In this category, the Model Selector approach is the most expensive as it requires that all component `DNNs` that the `premodel` chooses between need to be held in memory. However, this approach only requires 16% more memory than the most expensive model, a small cost

Table 6.7 The change in size of `ResNet_v2_152` when using different compression techniqies to shrink the `DNN`.

| Model | Size (MB) |
|---|---|
| Without Compression | 691 |
| Deep Compression | 317.12 |
| Quantization | 473.42 |
| Both Compression Methods | 226.22 |

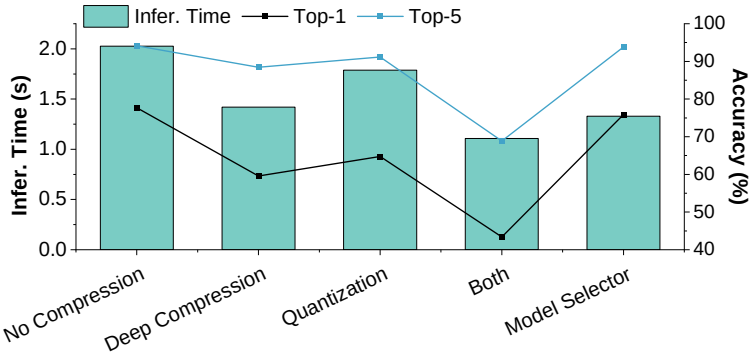



Fig. 6.17 The end-to-end top-1, top-5, and average inference time of a Model Selector approach when used in conjunction with `DNN` compression techniques. *Higher is better* for top-1 and top-5, *lower is better* for average inference time.

to pay for reduced CPU and GPU load, and a faster inference time with higher accuracy.

**Compression.** Up until now, all results have focussed on a scenario where multiple pre-trained candidate `DNNs` are available to solve the problem at hand. In some cases, only a single trained `DNN` is available; this could occue for several reasons, such as no pre-trained models being available and a restricted training time frame. This section presents a method of combining `DNN` compression with a Model Selector approach to improve average end-to-end inference time without a significant loss in accuracy. For this example, two different compression algorithms have been used: Deep Compression [36], and quantisation [56]. By first applying Deep Compression followed by quantisation, there is effectively have a third compression "algorithm". Compression is designed to make a `DNN` lighter, giving it a faster inference time and a smaller size (see Table 6.7); however, as a consequence, the model accuracy also degrades. `ResNet_v2_152` was chosen as the starting `DNN`. It is the most complex `DNN` considered in this work, meaning it achieves the highest top-1 and top-5 accuracy, and the longest inferencing time, at 2026ms. A total of four `DNNs` can be created by applying the three compression algorithms, plus the original `DNN`; this will form the pool of candidate `DNNs`. The entire end-to-end process can now be applied to select features, generate a `premodel`, and use

10-fold cross-validation to evaluate. Figure 6.17 compares a Model Selector approach against the original DNN (No Compression) and each compressed version of the DNN. Each approach is evaluated on average end-to-end inference time and top-1 and top-5 accuracy scores. A clear trend is shown in Figure 6.17, applying compression to the original DNN inference time is reduced at the cost of accuracy, as expected. Applying both compression methods in practice would result in an unacceptable accuracy drop, reducing top-1 accuracy by 34.32%. However, in this case, it makes sense. The Model Selector approach is able to make use of a DNN with such low accuracy by leveraging its relatively fast inference time (1.1s), almost 2x quicker than the base DNN, for the 43.4% of images it can correctly classify under top-1 scoring (or 68.86% for top-5 scoring). End-to-end, a Model Selector approach is able to achieve only a minor drop in accuracy, 1.76% for top-1, and 0.31% for top-5, while reducing average inference time by 1.52x. Effectively, a Model Selector approach is able to utilise the positive effects of compression (reduced inferencing times) while maintaining the accuracy of the original DNN.

### 6.1.4 Revisit Research Goals

In this section the research questions originally defined in Chapter 1 are revisited, they are discussed in order to formally assess the accomplishments of this evaluation with regards to the overall research goals. Only a brief discussion is presented here, a more comprehensive discussion can be found in Section 7.2. Two of the defined research questions (RQ1 and RQ2) have been conclusively answered by this evaluation, a third question has been touched on (RQ3), but not thoroughly evaluated. The following paragraphs discuss each research question in turn, referring to where each has been evaluated.

**[RQ 1]** *By combining multiple DNNs, is it possible to reduce the average inference time and computational cost across a dataset without causing a reduction in accuracy? Moreover, how much can inference time be reduced by?*

This research question has been answered during this evaluation by the Oracle premodel, showing the best possible average reduction in computational complexity and inference time, while increasing end-to-end accuracy. Both case studies clearly answer this question. The image classification case study (Section 6.1.1) shows that average inference time can be reduced by 45%, while increasing top-1 and top-5 scores by 14.48% (to 91.2%) and 3.16% (to 98.3%), respectively, when compared to the single most capable DNN. The machine translation case study (Section 6.1.2 shows that the end-to-end accuracy can potentially be increased by 25.7%, while decreasing average inference time by 24.5%

**[RQ 2]** *Is it possible to train a statistical machine learning model to choose the optimal DNN, at runtime, depending on the input and precision requirement?*

The two case studies presented in this evaluation try to answer this question; briefly, the evaluation shows that it is possible. There are two important caveats to note here: (i) use of this method requires all component DNNs to be held in memory, therefore there is a trade-off of increased memory consumption for the gains presented; and (ii) the reported inference time gains are on average across the dataset, there are some inputs that the input will take longer using this approach. Cases where the inference time is longer are few and far between, making up less than 2.3% of all inputs across both case studies; in all cases the inference time is only by the length of time the premodel takes to execute. The image classification case study (Section 6.1.1) shows that a trained premodel can achieve a 7.52% increase in top-1, and a 2.9% increase in top-5 accuracy, when compared to the single most capable DNN, with a 44.45% reduction in average runtime. The machine translation case study (Section 6.1.2 is less successful than the image classification case study, but is still able to achieve a 25.37% decrease in average inference time, with no significant impact on the end-to-end accuracy. The less successful machine translation premodel can be attributed to significantly less premodel training data being available, 10x to be exact.

**[RQ 3]** *Can orthogonal DNN optimisation techniques such as model compression be used in conjunction with a statistical machine learning model to further reduce inference time without a cost in accuracy?*

This research question is briefly touched on during this evaluation, but it is not thoroughly investigated. The upcoming evaluation (Section 6.2) covers this research question more comprehensively. Section 6.1.3, specifically Figure 6.17, analyses a simple approach of employing DNN compression techniques to produce a number of smaller and faster DNNs based on a starting model. This analysis indicates some potential for such an approach, specifically achieving a 90 ms reduction in average inference time, with a 16.37% improvement in end-to-end accuracy when compared to the closest comparable compression technique in terms of inference time.

### 6.1.5 Summary

This section presented a novel approach to efficient DNN inference for embedded systems, termed a Model Selector approach. The presented approach leverages multiple DNNs through the use of a premodel that is able to dynamically select the optimal DNN to use, depending on the model input and evaluation criterion, at runtime. Furthermore, an automatic approach to premodel generation, feature selection and tuning, and DNN selection was presented. The Model Selector approach was evaluated against two popular DNN application domains: image classification and machine translation; making use of convolutional and recurrent neural network architectures, respectively. The

presented experimental results indicate that a Model Selector approach to DNN inference optimisation is able to achieve good performance across DNN application domains and neural network architectures.

**Image Classification.** Experimental results show that a Model Selector approach achieves an overall top-1 accuracy of 87.44%, translating into an improvement of 7.52%. Furthermore, a 44.5% reduction in average end-to-end inference time is achieved when compared to the most accurate single image classification DNN.

**Machine Translation.** For machine translation a Model Selector approach is able to reduce average end-to-end inference time by 25.4%, when compared to the single most capable model RNN, without significantly effecting accuracy. If more premodel training data was available the Model Selector approach could achieve the Oracle premodel performance: a 25.4% reduction in average end-to-end inference time while increasing F1 measure by 20.51%.

The next section of evaluation will take the approach one step further. This section focused on the premodel, using off-the-shelf DNNs, whereas the next section focusses on the creation of a set of specialised DNNs to use in conjunction with a premodel.

## 6.2 DNN Specialisation - Evaluation

This section is based on work that is not yet published. The previous section evaluated the proposed approach to DNN inference optimisation of using a premodel to choose the optimal DNN to use, at runtime. Upon evaluation, some exciting results were revealed: it is possible to reduce inference time through the use of a premodel, in fact it is possible to even increase top-1, top-5, precision, recall, and F1 score. This section aims to take that work one step further, by generating the component DNNs. More specifically, this sections attempts to answer the question: *Is it possible to automatically synthesise both the premodel and the component DNNs (which form the ensemble) at the same time, from a single seed-DNN, in order to gain improved accuracy and inference time?*

Evaluation is split into four main sections: an end-to-end evaluation, analysis of the Data Segmentation Process (introduced in Section 4.3.2), analysis of the Sub-DNN Creation Process (introduced in Section 4.3.3), and a further in-depth analysis of the entire process. An end-to-end evaluation is provided in order to fully evaluate the DNN specialisation process from the point a user provides input until the final inference accuracy and runtime is presented. The end-to-end evaluation aims to provide a higher level of evaluation. Data Segmentation and Sub-DNN Creation analysis sections investigate their respective DNN specialisation components individually, aiming to further explain the characteristics of this approach. Next, a number of further in-depth analyses are

provided. Finally, this section ends by revisiting the research goals, then providing a short summary.

### 6.2.1 End-To-End Evaluation

End-to-end evaluation of the DNN Specialisation is carried out using image classification as a case study. The initial end-to-end results based on an Oracle premodel, that is able to predict the correct sub-DNN to use every time, proved to be promising for image classification. However, once a premodel was trained based on the Model Selector approach, these results were not realised; the premodel was just not accurate enough. Image classification was chosen to evaluate this work due to its excellent results in the previous evaluation.

Moreover, while ResNet_v2_152 was used during the evaluation of the Model Selector approach, it could not be used here. Previously, each DNN was only used for inference during premodel design, DNN specialisation requires some model training in order to function. Training a DNN requires significantly more GPU memory than inference as it is completed in batches, which inference does not require. ResNet_v2_152 is considerably large, containing around 60.4 million parameters; a server capable of training a model of such size was not available; ResNet_v2_50 was used instead, which contains approximately 25.6 million parameters. Due to both DNNs being the same ResNet architecture (ResNet_v2_152 containing more of the repeated ResNet module) it is reasonable to assume that the inference time gains presented in this section would carry over to ResNet_v2_152. It is more difficult to compare the potential accuracy scores between the two DNN models, and whether DNN specialisation would still see an increase in accuracy when ResNet_v2_152 instead.

First, the evaluation setup is described, followed by the data segmentation and sub-DNN creation processes. The section ends with an analysis of the end-to-end performance of the generated sub-DNNs and premodel when working together.

#### Setup

A pre-trained version of ResNet_v2_50 was used that was built and trained by independent researchers using Keras and the ImageNet ILSVRC 2012 *training* dataset. A subset of the ImageNet ILSVRC 2012 training dataset was used for data segmentation, allowing for a much faster search during data segmentation. The data segmentation training dataset was formed of 500k randomly selected images from the ImageNet ILSVRC 2012 training dataset; an equal number of images were selected from each of the 1k classes - forming a training dataset that is just less than half the size of the original. The ImageNet ILSVRC 2012 *validation* dataset is used to train and evaluate the premodel using k-fold cross-validation.

### Data Segmentation

The data segmentation process and its respective feature selection process is presented in this section. Throughout, the 500k images forming the data segmentation training dataset is used; all candidate features are extracted from each image. A value of 0.1 is used for $\delta$, the MSC tolerance parameter, as it produced good results in pilot experiments, this parameter is analysed in more detail in Section 6.2.2. First, all candidate features are described, alongside the initial feature selection process. Next, the greedy feature search is described, followed by a search for the best number of data segments. Finally, the process of selecting the best data segmentation is presented.

**Features.** A total of 29 candidate features were considered for data segmentation, shown in Table 6.1; the same features are used in the image classification case study of the Model Selector (Section 6.1.1). The same features were used as they performed exceptionally well for a `premodel`, achieving 98% accuracy overall. Again a correlation-based feature selection process was used to remove redundant features; Table 6.2 summarises the features removed, leaving 17 features. Next, a greedy search is performed in order to find the best set of features for clustering.

**Greedy Feature Search.** The importance of each remaining feature, after feature selection, is analysed next. Feature importance is determined by first generating a clustering using all features ($n$) and taking note of the Mean Silhouette Coefficient (MSC). Each feature is then removed, in turn, retraining and evaluating the clustering on the remaining $n-1$ features, noting the drop (or increase) in MSC. The feature that, when removed, results in the lowest drop (or highest increase) in MSC score is permanently removed, and the next iteration of the greedy search begins. This process is repeated iteratively until only 2 features remain; the MSC and Mean Squared Error (MSE) is recorded at the start of each iteration. The recorded MSC and MSE values are used later on when choosing the best features and data segment count. Once all iterations are complete the best three performing feature sets, in terms of MSC score, are recorded. Next, the search for the best number of data segments is described.

**Data Segment Search.** The above greedy feature search is repeated for numerous numbers of clusters, in this case values between 2 and 10 are considered. For each iteration of the data segment search the best three performing feature sets is recorded. Feature sets are named using the following convention *KMeans_i_j*, where *i* is the number of clusters, and j is the ID of the feature set used; each considered feature set is given a unique ID. A full list of the features contained in each relevant feature is provided in Appendix A. For reference, the feature set with ID 199 contains the following features: *edge_angle_4*, *area_by_perim*, *aspect_ratio*, *hue1*, and *hue7*. Next, all of the saved results are analysed to find the best segmentation of the training data.

Table 6.8 The best performing feature sets from each number of clusters. The feature sets that produce invalid cluster sizes have been removed. A full list of the features contained in each feature-set is provided in Appendix A.

| | | | |
|---|---|---|---|
| KMeans_2_193 | KMeans_2_187 | KMeans_3_228 | KMeans_3_233 |
| KMeans_3_171 | KMeans_4_320 | KMeans_4_190 | KMeans_5_373 |
| KMeans_6_199 | KMeans_6_376 | | |

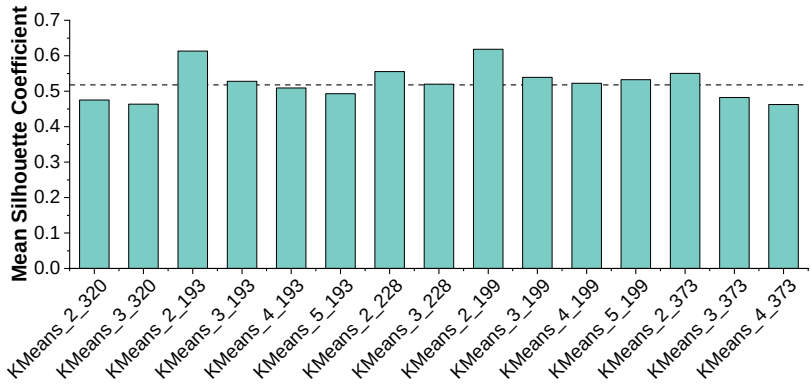



Fig. 6.18 Mean Silhouette Coefficient (MSC) of the best feature sets for every considered number of clusters. The $x$ axis of this figure matches Figure 6.19. Lower MSC threshold, determined as $max(MSC) - \delta$ is shown as a dashed line (where $y = 0.518$). For MSC, *higher is better*.

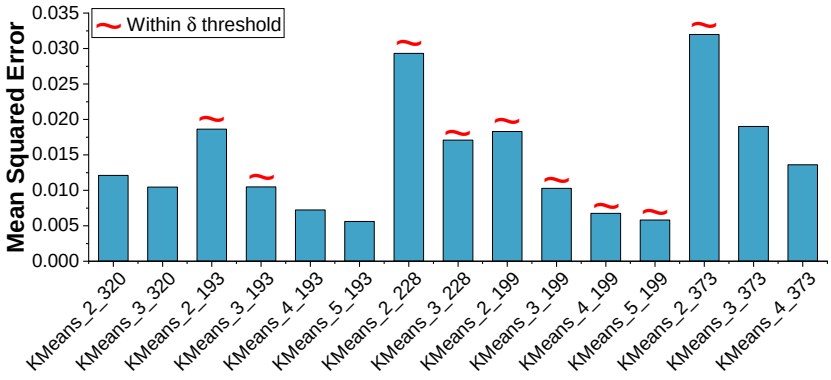



Fig. 6.19 Mean Squared Error (MSE) of the best feature sets for every considered number of clusters. The $x$ axis of this figure matches Figure 6.18. Red tilde marks show the feature sets that are greater than the lower MSC threshold of 0.518. For MSE, *lower is better*.

**Best Data Segmentation.** Finally, the best feature set and number of clusters is chosen; the chosen values are then used to split the training data into data segments. As stated in Section 4.1.2, MSC and MSE can be misleading when considered alone, therefore a hybrid of both methods is used. First, each of the best performing feature sets for each number of clusters are analysed, any feature set that produces a single cluster

containing less than 10% of the data is considered invalid and removed. A *minimum cluster size threshold* ($\theta$) value of 10 (representing 10% of the data) was chosen as a sub-DNN trained on less than 10% of the data would have reduced effectiveness [47, 155]. Table 6.8 shows the feature sets that achieve the highest MSC score, after invalid feature sets are removed. Next, every feature set remaining is scored against every number of clusters; again, any feature set that produces a single cluster containing less than 10% of the data is considered invalid and removed. Figures 6.18 and 6.19 show the MSC and MSE scores achieved by all remaining feature sets at this stage. It is clear in Figure 6.18 that many of the remaining feature sets achieve very similar scores, there is no runaway winner, emphasising the need for a hybrid approach. The lower MSC threshold is determined next as $MSC - \delta$, producing a value of 0.518, marked with a dashed line in Figure 6.18; every feature set that achieves a MSC score lower than the lower MSC threshold is removed and no longer considered. Looking at Figure 6.19 there is a clear winner, *KMeans_5_199* achieves the lowest MSE score, therefore data segmentation will create 5 data segments of the training data using the feature set with ID 199. Each data segment will be used to train an individual sub-DNN, based on the original seed DNN, `ResNet_v2_50`; the process is described in full in the next section.

### Sub-DNN Creation

The sub-DNN creation process is described in this section; this process follows on from data segmentation described in the previous section. As input, this process requires a copy of the seed-DNN, in this case `ResNet_v2_50`, and the data segments created in the previous section. Data segments are received via text files containing a list of images corresponding to a different data segment, numbered 0-4. During the sub-DNN creation process the first 35% of each sub-DNN is left untouched, as it learns generic features [98]; 35% was chosen as it produced good results in pilot experiments. If utilised effectively, the first 35% of the DNN that each sub-DNN will share can be leveraged to reduce storage requirements and memory usage; analysed further in Section 6.2.4. For reference, the base `ResNet_v2_50` achieves an average per image inference time of 369.4ms, and top-1 and top-5 accuracy scores of 67.25% and 87.75%, respectively. The next few paragraphs describe the initial fine-tuning, pruning and fine-tuning, and final fine-tuning steps, providing a description of the sub-DNN performance at each stage.

**Initial Fine-Tuning.** To begin, the seed-DNN is cloned, creating an exact copy, ready to be specialised into a sub-DNN; 5 sub-DNNs are created in this case. This step aims to slightly adjust the weights of each sub-DNN in order to prevent the first iteration of pruning from removing the same filters for every sub-DNN. In turn, each data segment is assigned a sub-DNN, and brief fine-tuning is performed to begin the specialisation process. A learning rate of $1 \times 10^{-5}$ is used for 10 epochs, or until the validation loss begins to

increase instead of decrease, indicating over-fitting. No sub-DNNs stopped early in this case. After initial fine-tuning the sub-DNN overall accuracy (across the entire ImageNet ILSVRC *validation* dataset) did not significantly change. Inference runtime is also not effected at this stage.

**Pruning and Fine-Tuning.** Now that the sub-DNN has been fine-tuned, pruning can begin. Pruning is an iterative process that happens per layer. After every 4 layers of pruning, the sub-DNN is fine-tuned for 3 epochs, at a low learning rate ($1 \times 10^{-5}$). During this process, every convolutional layer (after the first 35%) is pruned, in turn, by 40%; that is, 40% of all filters in a layer will be removed, based on their ranking. Pruning by 40% in each layer proved to be effective at reducing DNN overheads without significantly affecting accuracy during pilot experiments. In order to reduce the time taken in this stage of sub-DNN creation the models were not evaluated after each fine-tuning step, only at the end of all pruning and fine-tuning. However, investigating the fine-tuning loss, which was stable throughout this step, reveals that the final evaluation score is indicative of the end-to-end accuracy scores throughout. At the end of this stage of sub-DNN creation, top-1 accuracy for all 5 sub-DNNs ranges between 33% and 37%, the final fine-tuning stage next aims to increase this. Inference runtime of each sub-DNN decreased by around 35% during this stage as well.

**Final Fine-Tuning.** Finally, each sub-DNN undergoes some fine-tuning, designed to recover some of the accuracy lost during the pruning and fine-tuning stage. In this stage, the first 35% of the sub-DNN is not changed. A learning rate of $1 \times 10^{-5}$ was used until 20 epochs were completed, or the loss started to increase instead of decrease, indicating over-fitting. All sub-DNNs completed at least 16 epochs before exiting fine-tuning; only one sub-DNN reached 20 epochs. Final fine-tuning increased overall accuracy for all 5 sub-DNNs to a range between 47% and 54%; sub-DNN accuracy on its data segment only is analysed in Section 6.2.3. Inference runtime was not effected during this stage. Overall the sub-DNN creation process took around 20 hours per sub-DNN, around 4 days in total for 5 sub-DNNs, when running the the GPU-server described in Section 5.1.1.

### Premodel Generation

The premodel, considering only the sub-DNNs created so far, and feature selection process is presented in this section; it is similar to the process described in section 6.1.1 as the same candidate features are used, shown in Table 6.1. There is no need to utilise the DNN selection algorithm in this case as the sub-DNNs have been designed specifically to work together. Unfortunately, an accurate premodel could not be created using the premodel generation method described in Section 4.2; therefore, some alternate premodel methods are described in this section. Possible reasons for the low premodel accuracy are described in more detail in Section 6.2.4. The premodel is also discussed

further in Section 7.3. First, this section describes the candidate features used and the feature selection method to remove redundant features. Next, the feature analysis method is described. Alternate methods of premodel generation are then described, followed by a CNN based premodel. Finally, this section ends in a short summary.

**Feature Selection.** A total of 29 candidate features were considered based on previous image classification work [40, 130]. First, a correlation based feature selection method is applied, designed to remove redundant features. As this feature selection method does not consider the output label of each data point, this check results in the same features being removed as in Section 6.1.1, shown in Table 6.2. Of the 29 candidate features, only 17 remain; reduced further by feature analysis.

**Feature Analysis.** The feature analysis process was carried out individually for each of the premodel methods described in the next paragraph. During feature analysis, the importance of each feature is analysed, and the least important feature is removed, iteratively, until just 2 features remain. The accuracy of the proposed premodel is recorded each time a feature is removed, and the set of features that give the highest accuracy are chosen for that premodel. This stage aims to reduce the feature count without reducing premodel accuracy; in some cases accuracy can increase when bad features are removed.

**Premodel Training Data.** Unfortunately, an accurate premodel could not be created using the premodel generation method described in Section 4.2. A discussion of possible reasons is provided in Section 6.2.4. In order to work around this issue, a couple of alternate premodel training data generation techniques were implemented and tested, these methods utilise the data segmentation procedure to create training data better suited to the premodel. Each of the presented methods follow the full premodel generation method described in Section 4.2, the difference is the labels given to the training data. The following methods of training data generation were used:

- **Optimal DNN Premodel.** The premodel training data is generated in the same way as when off-the-shelf DNNs were used. An optimal DNN is found for each image in the ImageNet ILSVRC *validation* dataset. Cross-validation is used to evaluate the premodel, which is trained to predict the optimal DNN for each input it receives.

- **ClusterBased.** This premodel training data is based on the output of the data segmentation clustering, hence the name. A premodel is trained on the ImageNet ILSVRC *training* dataset, using the labels generated from data segmentation; the premodel is trained to predict which data segment, and therefore which sub-DNN, each image belongs to. The premodel then predicts the data segment for every image in the ImageNet ILSVRC *validation* dataset, and the corresponding sub-DNN

is used for image classification. In the cases where a sub-DNNs fails the classify an image, the image is re-assigned to a "failed" label. A premodel is then cross validated, including the added "failed" label.

- **ClusterBased-Improved.** This method builds upon the previous method. During experimentation it was revealed that some images assigned to a data segment by the ClusterBased method could not be correctly classified by the corresponding sub-DNN. However, in some cases, an alternate sub-DNN could correctly classify the image. Using this method, such images were reassigned to the closest data segment that could correctly classify it. This premodel aims to predict the correct data segment for each input image is receives.

**CNN based Premodel.** Similar to Section 6.1.1, a CNN based premodel is trained and evaluated. Previously, a CNN based premodel achieved low accuracy, translating into low end-to-end top-1 and top-5 scores. The low accuracy can be attributed to the small amount of training data available in comparison the typical CNN training data. However, data segmentation provides a solution to this problem. By utilising the ClusterBased method of premodel training data generation, a premodel effectively gains 500k training data points. With more training data, a CNN based premodel should be able to achieve a higher accuracy, and therefore, higher end-to-end top-1 and top-5 scores. The CNN based premodel uses a pre-trained MobileNet_v1_100 network as a base, before fine-tuning the network on the ClusterBased premodel training data.

**Summary.** Finally, the training data has been split, individual sub-DNNs created for each data split, and a number of premodels generated ready for evaluation. The next section presents an evaluation of the generated premodels and sub-DNNs against the seed-DNN, ResNet_v2_50.

### Overall Performance

Figures 6.20, 6.21, and 6.22 present the overall performance of DNN specialisation, in conjunction to a premodel, on the ImageNet ILSVRC 2012 *validation* dataset. This section analyses the end-to-end performance of such an approach, using the generated sub-DNNs and premodel described above, at a high level; individual components of this approach are analysed later. Each version of the premodel (utilising the sub-DNNs) is compared against the individual sub-DNN performance, an Oracle premodel (showing the best possible performance), and the starting seed-DNN: ResNet_v2_50. The following paragraphs analyse the inference times, accuracy scores, and precision, recall, and F1 scores of each of the approaches, before ending with a short summary.

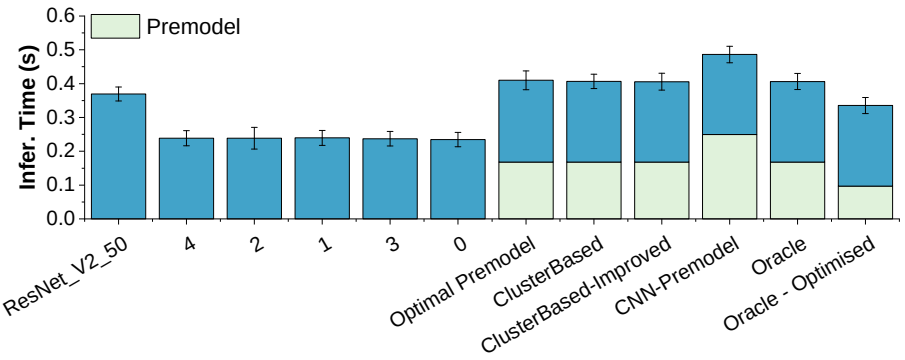



Fig. 6.20 A comparison of the end-to-end average inference times of each individual sub-DNN (numbered 0-4), different premodel methods, and the starting seed-DNN (ResNet_v2_50). The cost of the premodel is clearly shown in a lighter colour. An Oracle premodel is also shown, showing the inference time cost if a premodel is 100% accurate; the assumed cost of the Oracle premodel is the average of all other method's costs. The final column, Oracle-Optimised, shows an optimised premodel runtime. This is not directly implemented due to library dependencies. *Lower is better.*

**Inference Time.** Figure 6.20 compares the inference times of each of the DNNs and premodel approaches; the cost of the premodel is also included and clearly shown. The best performance in terms of inference time is each of the sub-DNNs, running around 35% faster than the seed-DNN. It is important to note that the reduction in inference time also comes at a cost in end-to-end accuracy when each sub-DNN is considered individually. Without considering the premodel cost, the end-to-end average inference time of each of the premodel approaches, including the Oracle, all achieve a reduction of approximately 35%. It is expected that a Model Selector approach will achieve a similar average runtime to the created sub-DNNs as it only utilises those models. Therefore, for this approach to reduce the inference time, including the premodel overhead, each sub-DNN needs a reduction in inference time greater than the premodel overhead. Unfortunately, in this case the premodel overhead is slightly higher than the inference time saved; 130ms is saved, and the premodel overhead is 167.8ms. However, provided the same parameters are used for sub-DNN creation using a bigger seed-DNN such as ResNet_v2_152, the same percentage of runtime reduction is expected, equating to a reduction of 303ms per image. Most premodel overhead is due to feature extraction. Furthermore, Oracle-Optimised shows the runtime of the premodel after some optimisation. The optimisation could not be applied directly to the production premodel due to contention between library dependencies on the NVIDIA Jetson TX2; the optimisation includes some GPU support in OpenCV when extracting features. Not all feature extraction tools are supported on the GPU at the time of writing, therefore further optimisation is expected. It is worth noting that the premodel inference time stated is the worst case scenario, where a single image is being processed at a time; if images are processed in batches the premodel

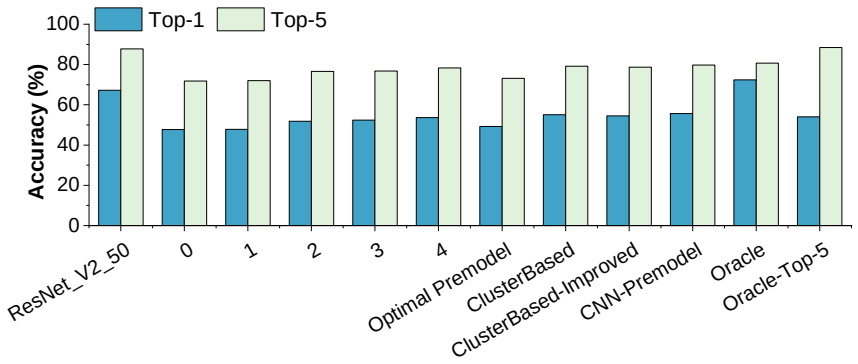



Fig. 6.21 A comparison of the end-to-end top-1 and top-5 scores of each individual sub-DNN (numbered 0-4), different premodel methods, and the starting seed-DNN (ResNet_v2_50). Top-1 (Oracle) and top-5 (Oracle-top-5) premodels are also shown, showing the highest achievable top-1 and top-5 scores if a premodel is 100% accurate, and optimising for that metric. *Higher is better* for all measures.

overhead would be significantly reduced per image. Finally, as expected the CNN based premodel is the most expensive, it would only be worth the added cost if it was a more accurate premodel, which is does not appear to be. Nevertheless, only considering ResNet_v2_50, given that DNN specialisation has the potential to significantly improve accuracy compared the the seed DNN, the small added inference time cost is acceptable.

**Accuracy Scores.** Figure 6.21 compares the top-1 and top-5 accuracy of each of the DNNs and premodel approaches. Here, the Oracle premodel shows the theoretically highest possible top-1 and top-5 scores achievable using the generated sub-DNNs. Interestingly, the Oracle premodel is able to achieve a higher top-1 accuracy than the original seed-DNN, increasing accuracy by 5.15%; indicating that each sub-DNN has learned some features not in the original seed-DNN. In this case, the Oracletop-5 accuracy has reduced in comparison to the seed-DNN, this is likely due to the each sub-DNN losing its generalisability; this could possibly be improved by training each sub-DNN on the entire training dataset for a few epochs. Furthermore, the Oracle is currently optimising for top-1 accuracy, if a premodel is trained to optimise for top-5 accuracy, such as Oracle-top-5, the end-to-end top-5 accuracy is also improved by 0.7%; as Section 6.1.1 shows, a it is much harder to improve top-5 accuracy when using a Model Selector approach. Both Oracle, and Oracle-top-5 use the same sub-DNNs, but they have different training data depending on their optimisation goal. Unfortunately, none of the proposed premodel approaches are capable of achieving the Oracle performance, this could be for a number of reasons, discussed in detail in Section 6.1.3. It is especially surprising that a CNN based premodel is unable to successfully classify the incoming images. Overall, if a premodel is able to achieve close to the Oracle performance, such as in Section 6.1.1, a 5.15% or 0.7% improvement in top-1 or top-5 scores can be achieved.

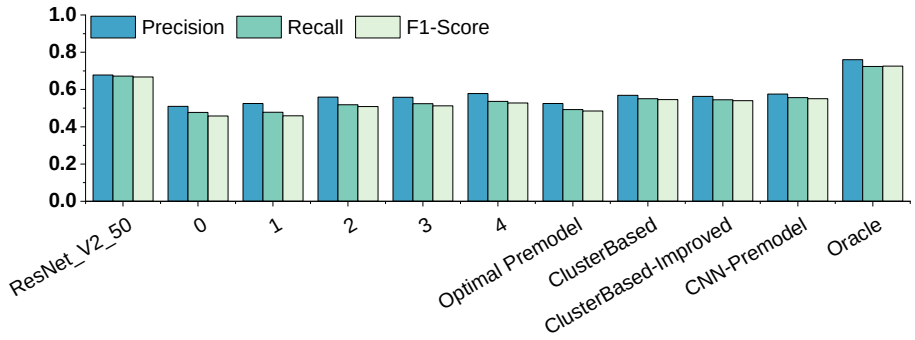



Fig. 6.22 A comparison of the end-to-end precision, recall, and f1 scores of each individual sub-DNN (numbered 0-4), different premodel methods, and the starting seed-DNN (ResNet_v2_50). An Oracle premodel is also shown, showing the highest achievable scores if a premodel is 100% accurate. *Higher is better* for all measures.

**Precision, Recall, and F1 Score.** Finally, Figure 6.22 compares the precision, recall and F1 scores of each of the DNNs and premodel approaches. The results here are very similar to those presented in the previous paragraph, the Oracle premodel is able to significantly improve all metrics over the seed-DNN, however, none of the premodel approaches are able to match it. Specifically, the Oracle is able to improve precision, recall, and f1 scores by 12.1%, 7.7%, and 8.7%, respectively. High precision can reduce false positives, which is important for certain domains like video surveillance because it can reduce the human involvement for inspecting false-positive predictions.

**Conclusion.** Overall, the Oracle shows that DNN specialisation has the potential to improve DNN accuracy in every measure (top-1, top-5, precision, recall, and F1-score), however, none if the proposed premodel approaches are able to reach that potential. The failure of the premodel approaches, and possible solutions, is discussed in detail in Section 6.1.3. As mentioned during the introduction to this section, ResNet_v2_50 was used for evaluation as there was no server available that is capable of training ResNet_v2_152. Due to both DNNs being part of the same architecture - ResNet_v2_152 has more repeating ResNet modules than ResNet_v2_50 - some assumptions can be made to extrapolate the results presented here to ResNet_v2_152. As both models are the same architecture, it is safe to assume that the reduction in inference time would carry over, leading to a 725ms reduction in average inference time per sub-DNN, over the normal runtime of ResNet_v2_152 of 2048ms. When the cost of the premodel is accounted for, this translates into a 27.3% reduction in end-to-end average inference time using DNN specialisation and ResNet_v2_152. Furthermore, as ResNet_v2_152 is a larger model it is reasonable to expect that it will be possible to use a higher level of pruning for each sub-DNN, as it a much bigger model classifying the same problem that ResNet_v2_50 is able to; this is discussed in more detail in Section 6.2.4. There are some cases where the DNN specialisation approach is unable to classify an image that the original seed-DNN

could, however these cases are few and far between, occurring just 5.50% of the time, analysed further in Section 6.2.4.

This sections attempts to answer the question: *Is it possible to automatically synthesise both the premodel and the component DNNs (which form the ensemble) at the same time, from a single seed-DNN, in order to gain improved accuracy and inference time?* The results presented up until now would imply that such an approach is not possible. Remaining sections in this evaluation analyse the inner workings of DNN specialisation, seeking to answer why it did not work in this case, when use of off-the-shelf DNNs did. Analysis has narrowed down the answer to one of 3 possible factors: (i) the metrics used to evaluate data segmentation are not sufficient or accurate, (ii) the features used to segment the training data are not sufficient, or (iii) choosing between multiple sub-DNNs is simply too complex for SML models. The following sections analyse individual components of DNN specialisation, making the case for the above factors. First the data segmentation process is analysed, followed by sub-DNN creation, before ending on an in-depth analysis of DNN specialisation.

### 6.2.2 Data Segmentation Analysis

This section analyses the working mechanisms of the data segmentation process in DNN specialisation, aiming to further explain the characteristics of such an approach. First the relative sizes of the data segments are analysed between the ImageNet ILSVRC training and validation datasets. Next, the feature selection process is analysed, followed by the number of data segments chosen.

#### Segment Size Changes

Figure 6.23 compares the sizes of the data segments between the initial data segmentation clusters, and those produced by each of the premodel approaches described in Section 6.2.1. Ideally, the best data segmentation would have a fairly even split between all segments that is also replicated in the premodel approach; this would indicate that the premodel and data segmentation are learning similar splits across the training and validation datasets. Each of the premodel approaches are evaluated in turn below.

**Optimal Premodel.** An optimal premodel approach produces similar cluster sizes to the DNN utilisation shown in Section 6.1, on which it is based; the skew is caused by some overlap in the sub-DNN prediction capabilities. Sub-DNN 0 is slightly faster than all other sub-DNNs, therefore it is chosen as the optimal DNN for every image it is capable of predicting, leaving sub-DNNs such as 2 to be left with just 1.86% of the data. It is likely that this is why the optimal premodel is so inaccurate (only 49%).

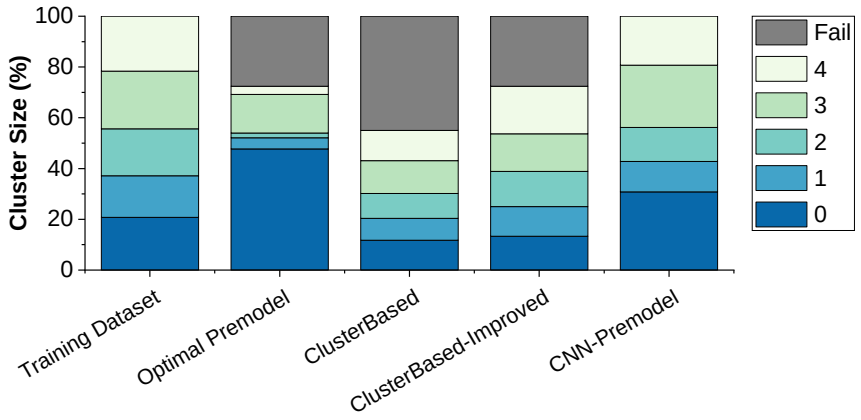



Fig. 6.23 A comparison of the change in data segment sizes between training dataset, which data segmentation is carried out on, and different `premodel` approaches on the ImageNet ILSVRC *validation* dataset. Numbers 0-4 represent different data segments, fail represents the set of images that the `premodel` learns cannot be correctly predicted by any of the sub-`DNNs`. The segment sizes show the predictions made by each of the `premodel` approaches. An `Oracle` `premodel` is not included here as it is ambiguous.

**ClusterBased and ClusterBased-Improved.** The ClusterBased and ClusterBased-Improved `premodel` approaches reveal the need for an improved version, and a potential fault in the data segmentation approach. Figure 6.23 clearly shows a huge amount of the validation data being marked as Fail, 44.97%, whereas only 27.6% of the images actually fail. Originally, segment labels are assigned based on a `SML` model trained on the training dataset segments, which then predicts the labels for the validation dataset; in this case a number of images originally assigned to each segment cannot be correctly classified by the sub-`DNN`, resulting in the high Fail rate. ClusterBased-Improved builds on this by re-assigning Fail images to the next nearest segment (in terms of clusters) able to make a correct classification, revealing that 17.37% can be re-assigned for correct classification. It is not clear if this disconnect between training and validation datasets comes from an inherent disconnect between the datasets, meaning training a `SML` model in one and predicting on the other will always give bad results; or the data segmentation parameters chosen are simply bad parameters, resulting in sub-`DNNs` that are not specialising well to segment of images assigned to it. ClusterBased-Improved gives 5 fairly evenly distributed clusters.

**CNN-Premodel.** It was expected that the `CNN-premodel` would segment the validation dataset in a similar way to the segmentation of the training dataset. Data segments 3 and 4 are very similar to one another, both within 2%. However, segment 0 grows by 10.1% in the validation set, with 1 and 2 both shrinking by around 5% each. As a CNN-`premodel` is expected to be the most accurate (it is the most powerful model used), a deeper investigation into the change in segment size could reveal some key insights.

Table 6.9 The features that make up each of the feature sets used in Figure 6.24. Model_Selector_Features are the same as those used in Section 6.1.1. All Features are described in full in Table 6.1.

| Feature Set | Features | | | |
|---|---|---|---|---|
| **Model_Selector_Features** | *n_keypoints*<br>*contrast* | *avg_perc_brightness*<br>*area_by_perim* | *aspect_ratio*<br>*edge_length1* | *hue1* |
| **fid_199** | *edge_angle3*<br>*aspect_ratio* | *area_by_perim* | *hue1* | *hue7* |
| **fid_193** | *edge_angle3*<br>*aspect_ratio* | *area_by_perim*<br>*n_keypoints* | *hue1* | *hue7* |

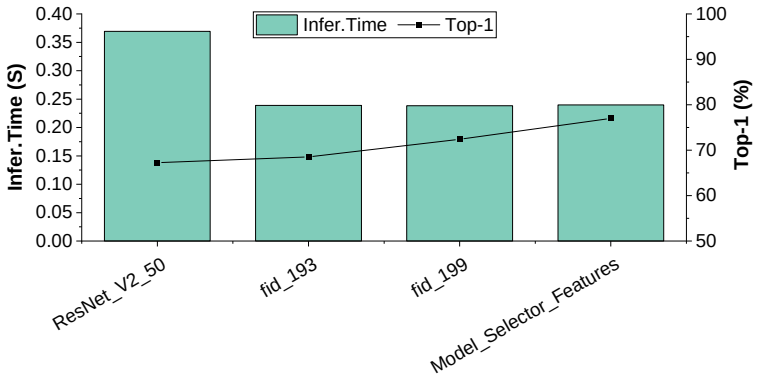



Fig. 6.24 A comparison of the end-to-end top-1 accuracy and average inference time achieved by an `Oracle premodel` when changing the feature set used; feature sets are described in Table 6.9. An `Oracle premodel` shows the best achievable scores in each metric. Note that the right axis starts at 50, to better show the top-1 accuracy change. *Higher is better* for top-1 accuracy, *lower is better* for average inference time.

### Feature Selection

Figure 6.24 compares the end-to-end top-1 accuracy and average inference time achieved when using different feature sets to form the data segments; `Oracle premodel` scores have been used to ensure the `premodel` does not introduce noise. Feature sets are described in Table 6.9. Five data segments have been used each time, which reduces the number of valid feature sets (from Section 6.2.1) down to 2. The results presented in Section 6.2.1 are the same as *fid_199*. The feature sets are also compared against the seed-`DNN` used, and a data segmentation process using the features selected during Model Selector evaluation (Section 6.1.1), termed *Model_Selector_Features*; these features have been proved to generate a good `premodel`. For context, the *Model_Selector_Features* clustering achieves a MSC of 0.274, much lower than *fid_199* and *fid_193* scores of 0.532 and 0.613, respectively. Looking at Figure 6.24, the three feature sets being compared all appear to produce very similar performance for both average inference time, and top-1 accuracy. Given the MSC scores, *Model_Selector_Features* is expected

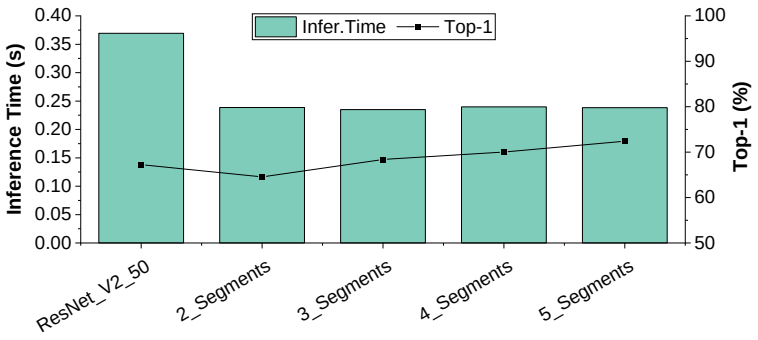
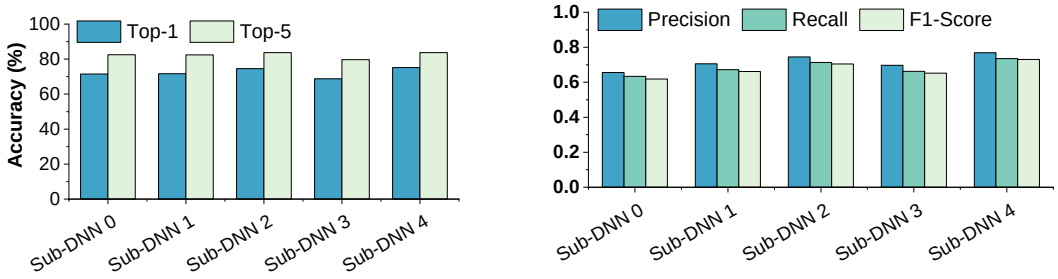



Fig. 6.25 A comparison of the end-to-end top-1 accuracy and average inference time achieved by an `Oracle premodel` when changing the number of data segments used. An `Oracle premodel` shows the best achievable scores in each metric. Note that the right axis starts at 50, to better show the top-1 accuracy change. *Higher is better* for top-1 accuracy, *lower is better* for average inference time.

to perform much worse than *fid_199* and *fid_193*, however that is not the case. Surprisingly, *Model_Selector_Features* actually out-performs the other two feature sets in every evaluation metric, implying that MSC and MSE are bad predictors of data segmentation performance after sub-`DNN` evaluation. Solutions to this problem are discussed further in Section 7.3.

### Data Segment Counts

Figure 6.25 compares the end-to-end top-1 accuracy and average inference time achieved when using different counts of data segments; `Oracle premodel` scores have been used to ensure the `premodel` does not introduce noise. The same feature set (*fid_199*) was used each time. No more than 5 data segments were considered as any more produced invalid data segmentation clusterings, see Section 6.2.1. The results presented in Section 6.2.1 are the same as *5_segments*. Across every count of segments the end-to-end average inference time stays the same, this is because the level of pruning has not been changed. As the data segment count increases each sub-`DNN` can be pruned more without affecting accuracy, this is evaluated further in Section 6.2.3. Figure 6.25 reveals a clear trend, as the data segment count increases, so does the potential for end-to-end top-1 accuracy; in fact, using 3 data segments or more increases top-1 accuracy. It is interesting that end-to-end top-1 accuracy can be improved simply by splitting a problem into numerous smaller problems. Perhaps breaking the problem down, and assigning each to its own `DNN`, is similar to simply using a larger `DNN`. Both approaches introduce more computational power for the same problem, however, breaking the problem down means an increase in inference time is not be necessary. Furthermore, the Model Selector approach uses off-the-shelf `DNNs`, it could be the case that different `DNN` architectures are better at learning different features, and therefore capable of classifying different images. The fact




(a) Accuracy Scores

(b) Precision, Recall, and F1

Fig. 6.26 The overall performance of each sub-DNN created during Section 6.2.1 on the ImageNet ILSVRC 2012 *validation* dataset images that are predicted to belong to that data segment. The prediction is made by the most accurate premodel approach: CNN-premodel. *Higher is better* for all metrics.

that utilising a premodel works with off-the-shelf DNNs and not for DNN specialisation implies that the fault lies in the data segmentation. Similar to feature selection, the best performers achieve bad MSC and MSE scores, providing further evidence that MSC and MSE are bad predictors of data segmentation performance after sub-DNN evaluation. Possible solutions are discussed in Section 7.3.

### 6.2.3 Sub-DNN Creation Analysis

This section analyses the working mechanisms of the sub-DNN creation process in DNN specialisation, aiming to further explain the characteristics of such an approach. First, the accuracy of each individual sub-DNN is analysed, followed by an analysis of the effect pruning has on end-to-end accuracy scores and inference times.

#### Sub-DNN Accuracy

Figure 6.26 presents the accuracy scores of each of the sub-DNNs created during Section 6.2.1. Unlike Figure 6.21, which shows the accuracy across the entire ImageNet ILSVRC 2012 *validation* dataset, this Figure only shows each sub-DNN's accuracy on images that are predicted to belong to that data segment. The labels used were made by the best performing Oracle premodel approach: ClusterBased-Improved; the failed images have been assigned their original sub-DNN. If the Fail label was included, each sub-DNN would achieve 100% accuracy as all the images that could not be predicted would never be seen by a sub-DNN. Across the entire ImageNet ILSVRC 2012 *validation* dataset the sub-DNNs achieve top-1 accuracy scores ranging from 47.7% to 53.6%. When each sub-DNN focusses on the images it has been trained to specialise in, top-1 accuracy scores range from 68.71% (sub-DNN 3) to 75.14% (sub-DNN 4), a significant increase. Moreover each sub-DNN on its dataset out-performs the seed-DNN, ResNet_v2_50, across

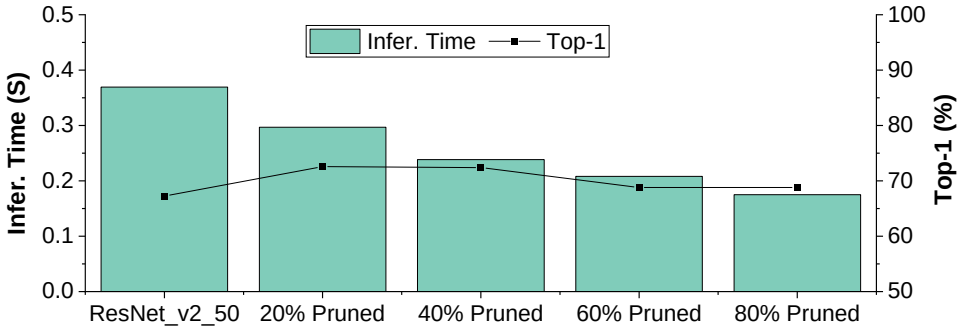
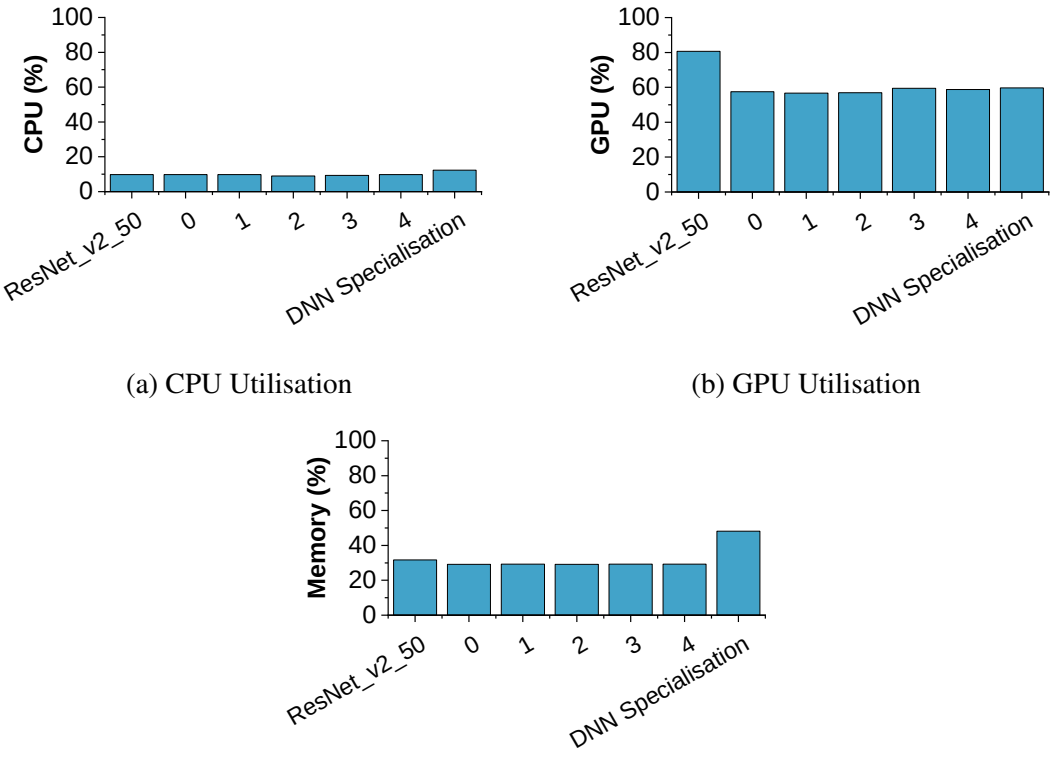



Fig. 6.27 A comparison of the end-to-end top-1 accuracy and average inference time achieved by an `Oracle premodel` when changing the level of pruning used during sub-DNN creation. An `Oracle premodel` shows the best achievable scores in each metric. Note that the right axis starts at 50, to better show the top-1 accuracy change. *Higher is better* for top-1 accuracy, *lower is better* for average inference time.

all images; this is what causes the increase in overall top-1 accuracy when using a DNN specialisation approach. Unfortunately, for top-5 accuracy each sub-DNN is out-performed by around 5% by the seed-DNN, indicating a loss of generalisability of each sub-DNN- this is to be expected as each sub-DNN is being specialised for a subset of the total dataset. Finally, all sub-DNNs except sub-DNN 0 out-perform the seed-DNN in precision, recall, and F1 scores; sub-DNN 0 achieves a recall score 0.05 lower than the seed-DNN.

### Sub-DNN Pruning

Figure 6.27 compares the end-to-end top-1 accuracy and average inference time achieved when using different levels of pruning during sub-DNN creation; `Oracle premodel` scores have been used to ensure the `premodel` does not introduce noise. The results presented in Section 6.2.1 are the same as *40% Pruned*, only the level of pruning was changed between evaluations. As expected, the general trend shows that more pruning leads to a faster end-to-end average inference time. By pruning the seed-DNN by 60% during specialisation a average inference time gain of around 208ms per can be achieved, however this comes at the cost of accuracy, dropping accuracy by 68.81%; slightly higher accuracy than the seed-DNN which achieves 67.25%. Furthermore, by pruning each sub-DNN by 80%, inference time can be further reduced to 181ms, while still increasing accuracy over the seed-DNN by 1.55%. The perfect level of pruning to use, that is, the level of pruning that reduces inference time without losing accuracy, in this case is unclear. Furthermore using the same level of pruning for all layers may not be the ideal approach, however that is out of the scope of this work, and is discussed further in Section 7.3. Another interesting point of investigation could be an analysis of how the number of data segments relates to the best level of pruning, also discussed further in Section 7.3.




(a) CPU Utilisation

(b) GPU Utilisation

(c) Memory Utilisation

Fig. 6.28 The average CPU, GPU, and memory utilisation of the implemented DNN Specialisation approach compared against each individual sub-DNN and the seed-DNN. The values are presented as the average utilisation across every test image in a 10-fold cross-validation. *Lower is better* for all figures.

### 6.2.4 Further Analysis

This section presents a few short in-depth analyses of different aspects of DNN specialisation. First, the resource utilisation of the DNN specialisation approach is analysed, followed by a comparison of the predictive power of the seed-DNN and DNN specialisation approach. Next, each of the proposed premodel approaches is analysed and discussed. Finally, DNN specialisation is evaluated using a toy dataset and DNN model, and the results discussed.

#### Resource Utilisation

In general, DNN specialisation has a similar resource utilisation as a Model Selector approach using pre-trained off-the-shelf DNNs. Figure 6.28 compares the average CPU, GPU, and memory utilisation of the implemented DNN Specialisation approach (Section 6.2.1) against each individual sub-DNN and the seed-DNN. Each SML premodel approach produces similar resource utilisation as they are all based on the sub-DNNs. The reported values are the averages across every single image in the testing dataset

Table 6.10 The predictive power of DNN specialisation compared to the seed-DNN, ResNet_v2_50. Total is the top-1 accuracy of each approach on the ImageNet ILSVRC 2012 *validation dataset*. Overlap is the percentage of all images that both approaches can correctly classify under top-1 scoring, and Unique is the percentage of all images that each approach can correctly classify and the other cannot, under top-1 scoring.

| Model | Total | Overlap | Unique |
|---|---|---|---|
| ResNet_v2_50 | 67.25% | 61.75% | 5.50% |
| DNN Specialisation | 72.40% | 61.75% | 10.65% |

(ImageNet ILSVRC 2012 *validation* dataset). Each resource is briefly discussed in turn below.

- **CPU.** Figure 6.28a presents the CPU utilisation. Similar to a Model Selector approach all DNNs run on the GPU, therefore CPU utilisation is not significantly effected, and depends on the utilisation of the sub-DNNs used. In this case ResNet_v2_50 was used, giving a CPU utilisation of 12.32%.
- **GPU.** Figure 6.28b shows the GPU utilisation; as expected, this is much higher than the CPU utilisation. Similar to a Model Selector approach, DNN specialisation results in lower average GPU utilisation across the dataset when using DNN specialisation the drop in GPU utilisation directly relates to the amount that each sub-DNN is pruned. Using DNN specialisation, GPU utilisation is 20.96% lower on average.
- **Memory.** Finally, Figure 6.28c compares the memory utilisation of a DNN specialisation approach. DNN specialisation is the most expensive in this area as it required that all sub-DNNs are held in memory ready for inference. If each sub-DNN in pruned further, memory consumption can be reduced; discussed further in Section 6.2.3. This approach only requires 16.52% more memory than the seed-DNN alone, a small cost to pay for reduced GPU load, a faster inference time, and a more accurate predictor.

### Predictive Power

It is possible that the generated sub-DNNs are not able to make all of the same predictions as the original seed-DNN. In other words, there could be a set of images that the seed-DNN is capable of correctly classifying under top-1 or top-5 accuracy scoring, that the set of sub-DNNs is unable to classify under the same scoring metric. This section analyses the predictive power of each of the approaches and compares them in order to quantify the occurrence of such an event. Table 6.10 presents a comparison of the predictive capabilities of DNN specialisation and ResNet_v2_50 on the ImageNet ILSVRC 2012 *validation* dataset. Although the sub-DNNs are based on the seed-DNN, they 'forget' how

to classify 5.50% of the validation images; however, collectively they learn how to classify an extra 10.65% of the images, resulting in a net increase in accuracy of 5.15%. It is unclear why DNN specialisation is unable to classify some of the images that the seed-DNN can, two reasons are suggested here: (i) it is due to the images requiring a more complex DNN, something that DNN specialisation does not provide; or (ii) it is due to the sub-DNNs over-fitting as they have been fine-tuned on a smaller set of images. Furthermore, this also indicates that the overall end-to-end top-1 accuracy could be increased to 77.90%, a 10.65% increase over the seed-DNN, if it is also included in the premodel's pool of DNNs to choose from. This opens up a new research opportunity, adding the seed-DNN would make the premodel's task more complex - a task that the premodel is currently unable to accurately predict as is. More research is required to improve premodel accuracy.

#### Premodel Analysis

This section analyses each of the premodel methods introduced in Section 6.2.1, analysing the automatic premodels that were created, and discussing why they are not able to perform as well as in Section 6.1.1. First, the architecture of each premodel approach (generated using the automatic method introduced in Section 4.2) is analysed alongside their respective accuracies. Next, an alternate premodel approach, ignoring the Fail label is discussed, followed by a summary and short discussion of potential future work regarding the premodel.

**Premodel Approaches.** Each of the premodel approaches introduced in Section 6.2.1 utilised the automatic premodel generation process introduced in Section 4.2. This section provides a short summary of the results of that automatic process; every multiple classifier architecture contained 5 classifiers, one for each sub-DNN. Below a short analysis of the premodel generated, and premodel accuracy for each of the premodel approaches is given.

- **Optimal Premodel.** This was the initial premodel generated, choosing a single classifier architecture KNN model. An accuracy of just 48% was achieved by this premodel approach.
- **Clusterbased.** Based on the assumption that data segments in the training and validation datasets would be the same, this premodel also chose a single classifier architecture KNN model. This premodel approach improved on the Optimal premodel slightly, achieving a prediction accuracy of 54%.
- **Clusterbased-Improved.** The only premodel approach to choose a multiple classifier architecture, made of KNN models. This approach was designed with the

aim of improving end-to-end top-1 accuracy, at the cost of potentially introducing some noise into the training data. However, no noise seemed to be introduced, this approach achieved a prediction accuracy of 54% too. Unfortunately, end-to-end top-1 accuracy was not improved.

- **CNN-Premodel.** Finally, it was expected that the `CNN-premodel` would achieve a high end-to-end top-1 accuracy score due to its ability to solve complex problems, and large amount of training data. Unfortunately this was not the case, achieving similar end-to-end top-1 accuracy as ClusterBased. Positively, ClusterBased and this `premodel` approach often agreed on the correct sub-`DNN` for an image, not counting the images predicted as Fail (as `CNN-premodel` is unable to make that prediction).

For each of the `premodel` approaches a greedy feature search was carried out to find the best features. The features chosen were often the same, or very similar, to the features used during data segmentation. In the cases where very similar feature sets were chosen, the `premodel` was again analysed using the data segmentation features, achieving a prediction accuracy that was not statistically significant from the chosen features. This indicates there is a strong relationship between the segments created during data segmentation, and the resulting segments in the validation dataset; `premodel` classifiers were always chosen to be `KNN` models, which is closely related to KMeans, indicating further evidence of the relationship. Furthermore, experimentation revealed that when utilising `DNN` specialisation the automatic `premodel` generator appears to be more likely to choose a single architecture `premodel`, this is likely due to the bigger choice of `DNNs` available alongside the generally lower accuracy scores.

**Alternate Premodel Approaches.** This section briefly explores an alternate approach considered for `premodel` training data generation. The approach discussed here was not included in the full results due to its bad end-to-end performance results, nonetheless; it reveals interesting insights into the segmentation of the training and validation dataset. After creation of the ClusterBased-Improved `premodel` training dataset, it became clear that a large number of images were being predicted as Fail, 47.65% to be exact, much higher than the `Oracle` prediction rate of 27.6%. It was speculated that this could be due to the actual Fail images not forming any proper boundaries outside of the predicted data segments, therefore introducing noise into the model. In order to test this theory a new training dataset was created based on ClusterBased-Improved, except the images that no sub-`DNN` could predict were not re-assigned to a Fail label. Using this approach a `premodel` was generated using the automatic `premodel` generation method and evaluated for its end-to-end performance. Although this approach increased `premodel` accuracy from 54% (using the ClusterBased-Improved approach) up to 86%, the end-to-end top-1 and top-5 scores did not significantly change. This analysis further indicates that the

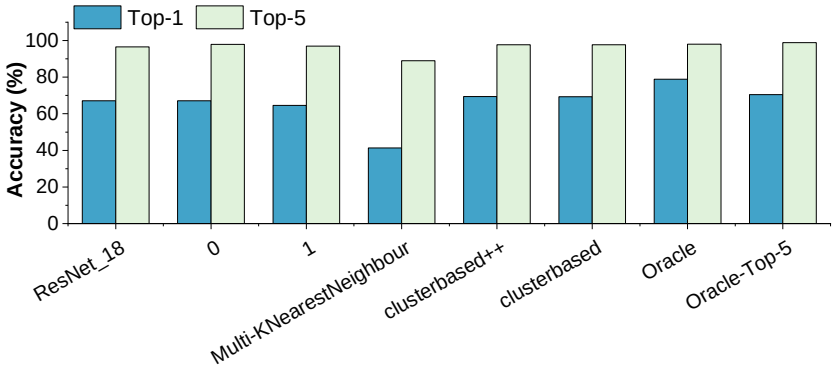



Fig. 6.29 A comparison of the end-to-end top-1 and top-5 scores of each individual sub-DNN (0, 1), different premodel methods, and the starting seed-DNN (ResNet_v2_18). Top-1 (Oracle) and top-5 (Oracle-top-5) premodels are also shown, showing the highest achievable top-1 and top-5 scores if a premodel is 100% accurate, and optimising for that metric. *Higher is better* for all measures.

segmentation in sub-DNN prediction power is not the same as the segmentation produced by training a model on the training data segmentation, then predicting on the validation data.

**Summary.** This section analysed each of the premodel approaches introduced in Section 6.2.1. It is clear from the results presented that the suggested premodel approaches are not able to accurately predict the correct sub-DNN to use for each image. Furthermore, when a premodel is cross-validated for the same task (predicting the correct data segment for an image) on the ImageNet ILSVRC 2012 *training* dataset, it is able to achieve a high accuracy. For example, a ClusterBased premodel trained using the training data of data segmentation in Section 6.2.1 results in 87% accuracy, much higher then the validation dataset accuracy of 54%. Clearly, the parameters used to evaluate data segmentation clusters (MSC and MSE) are not up to the task, they are unable to accurately predict if a data segmentation will be good when evaluated end-to-end. Solutions to this problem are discussed in more detail in Section 7.3.

### Toy Dataset and Model

This section evaluates the DNN specialisation approach against a toy dataset and DNN model. The evaluation has been carried out in order to reveal new insights into why the DNN specialisation premodel does not perform as well as the premodel described in Section 6.1.1. Although toy datasets and models are not always representative of the performance gain that can be provided by a full DNN and dataset, they do allow rapid prototyping and testing; that has been their use case here. The following paragraphs describe the setup of this experiment, followed by a discussion of the results and

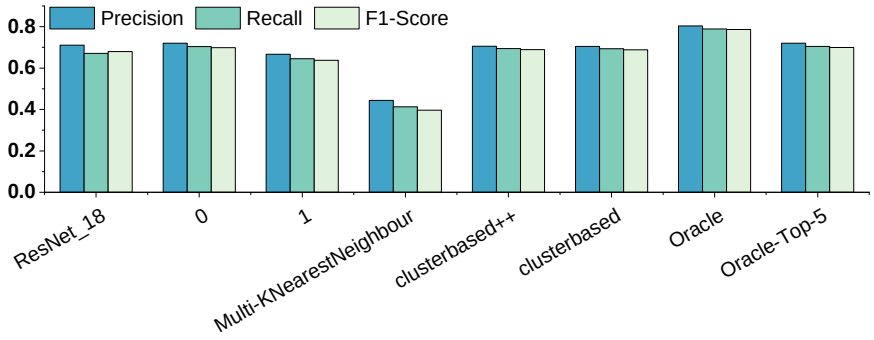



Fig. 6.30 A comparison of the end-to-end precision, recall, and f1 scores of each individual sub-DNN (0, 1), different premodel methods, and the starting seed-DNN (ResNet_v2_18). An Oracle premodel is also shown, showing the highest achievable scores if a premodel is 100% accurate. *Higher is better* for all measures.

premodel accuracy. To end this section, a discussion is provided on the implications of the presented results.

**Setup.** ResNet_v2_18 and Cifar-10 [1] were chosen due to their popularity and availability. ResNet_v2_18 follows the same ResNet architecture as ResNet_v2_50, however, it is much smaller. Cifar-10 contains 60,000 images which belong to 10 classes; 50,000 of the images form the training dataset, and the remaining 10,000 images form the test dataset. The presented results are from the top-scoring data segmentation during that process. A small discussion has been added at the end of this section; other data segmentation results are discussed there. All other aspects of the toy dataset and model evaluation were exactly the same as the experiments described in Section 6.2.1.

**Results.** Figures 6.29 and 6.30 compare the end-to-end accuracy of the generated premodels and Oracle against the seed-DNN and created sub-DNNs. Runtime and energy consumption data is not shown here; however, the results follow the same patterns shown in Section 6.2.1, that is, the sub-DNNs and premodel result in faster runtime, and lower energy use. A big difference between this and previous results is the low drops in accuracy for sub-DNNs. Sub-DNN 0 shows a negligible drop in accuracy, almost matching the seed DNN, and sub-DNN 1 only drops 2.6% accuracy over the seed DNN. The reasons for this are likely two fold: (1) Only 2 sub-DNNs were created, therefore, each will be less specialised and generalise better to the whole dataset; (2) Cifar-10 is a much simpler dataset, containing only 10 classes, making their predictions much easier than Imagenet predictions. More accurate sub-DNNs easily translate into better end-to-end accuracy, discussed in the next paragraph.

**Premodel.** Similar to the overall results, a number of different premodel architectures were considered, alongside two different Oracle premodels. As a reminder, an Oracle

[1] http://www.cs.toronto.edu/ kriz/cifar.html

in this thesis is a perfect predictor that is capable of achieving 100% accurate predictions; it indicates the best possible end-to-end accuracy that could have been achieved. In this case the best `premodel` (*clusterbased*) was more accurate – reaching 87.45% accuracy. This is likely due to a much less complex prediction that needs to be carried out; the `premodel` only needs to choose between two sub-`DNN`s. Higher `premodel` accuracy translates into higher end-to-end accuracy improving upon the seed `DNN` alone; albeit only a 2.21% accuracy increase. For this experiment, none of the `premodel` architectures first introduced in Section 6.1.1 were able to achieve a high accuracy. Multi-KNearestNeighbour achieved the best accuracy, at only 43.76%. This indicates that data segmentation is having an impact on the ability of the `premodel` to achieve accurate predictions, and a different kind of `premodel` is required, such as *clusterbased*, which uses the data from data segmentation to create a `premodel`. Finally, the `Oracle` achieves a 11.72% increase over the seed-`DNN`. slightly higher than the full evaluation in Section 6.1.1, again this is likely due to the simpler task at hand.

**Summary.** So far, the results have focussed on the best performing segmentation from the data segmentation process. As performing end-to-end evaluations is much quicker using toy datasets, a number of top performing datasets, and some badly performing datasets, were evaluated end-to-end. The results follow the same pattern as those presented in Section 6.2.2: the accuracy scores achieved do not correlate with MSC and MSE scores; data segmentations which score low with MSC and MSE are able to achieve high end-to-end accuracy. This again indicates that data segmentation, specifically MSC and MSE, are to blame for the difference between the results in this section and 6.1.1. A more exhaustive search, perhaps using the toy dataset and model described here, will be required in order to develop a better set of heuristics for evaluation of different data segmentations. This is discussed further in future work (Section 7.3).

### 6.2.5 Revisit Research Goals

This section again revisits the research questions originally laid out in Chapter 1, in the context of the above results. Both RQ1 and RQ2 were conclusively show to be answered in Section 6.1.4. Some results in this evaluation bring RQ2 into question, however further analysis reveals the `premodel` is likely not to blame. In order to keep this section brief, RQ2 is discussed further in Section 7.2. This evaluation mainly focussed on attempting to answer RQ3 and RQ4, they are discussed in turn below, referring to where each has been evaluated.

**[RQ 3]** *Can orthogonal `DNN` optimisation techniques such as model compression be used in conjunction with a statistical machine learning model to further reduce inference time without a cost in accuracy?*

This research question can be answered by investigating the Oracle performance in Section 6.2.1. In brief, the results presented during this evaluation remain inconclusive. The case study provided, using a relatively simple DNN (ResNet_v2_50), shows the potential for this approach, but is unable to prove it. Extrapolating the results to applying this approach to a more complex DNN, such as ResNet_v2_152, it is easy to assume the same percentage drop in inference time (35%) can be achieved. It is likely that each sub-DNN can be further pruned as the seed-DNN is much larger, but trying to solve the same sub-problem. Also, a smaller percentage of the DNN will be frozen in order to maintain the common, simple features, yielding even further inference time reductions. Furthermore, this approach allows for more efficient memory usage, reducing memory consumption over off-the-shelf DNNs. This approach is potentially more beneficial than simple DNN compression alone.

**[RQ 4]** *Can a set of DNNs be generated that are optimised to work together, when combined with a statistical machine learning model, that achieve even further reductions in computational costs and inference times?*

Similar to RQ3, the evaluation in this section has been unable to conclusively answer this research question. The evaluation does show the potential for this approach, but is unable to reach it. Section 6.2.1 shows the potential of automatic ensemble creation, indicating that with an accurate premodel inference time can be reduced and accuracy increased. In order to conclusively answer this question, more work is needed. Sections 7.2 and 7.3 discuss this question in more depth, and the future work required on order to answer this question more conclusively.

### 6.2.6 Summary

This section built upon the previous to present DNN specialisation, combined with a premodel, to create a novel approach to efficient DNN inference for embedded systems. The presented approach analyses the DNN training data and splits it into a number of data segments. The data segments are designed so that each segment contains similar images while being as dissimilar as possible to every other data segment. Each data segment is used to train and specialise a DNN based on a pre-trained seed-DNN, such as ResNet_v2_152, termed a sub-DNN. Finally, a premodel (from the Model Selector approach in the previous section) is trained to dynamically select the correct sub-DNN to use at runtime. The DNN specialisation approach was evaluated against image classification, a popular DNN application domain. Based on an Oracle premodel, that is, a premodel able to predict the correct sub-DNN to used for every input, the presented results indicate that DNN specialisation has the potential to reduce inference time and increase accuracy, similar to a Model Selector approach. Furthermore, compared to a

Model Selector approach, DNN specialisation has the potential to further reduce inference time, GPU utilisation, and memory consumption while still retaining an increase in accuracy. Unfortunately the results also show that no premodel could be generated that is accurate enough to utilise this potential. This could be due to pruning, sub-DNN training, or data segmentation, further investigation is required; discussed further in Section 7.3

# Chapter 7

# Conclusion

To end this thesis a brief summary of the work presented, the goals that were met, the goals that need more work, and a discussion on the future work in this area is given. First, a summary of this thesis is presented, highlighting the main successes and contributions of this work. Next, the research questions (laid out in Chapter 1) are discussed, noting where and how each research question has been answered. A discussion of the possible future work which could enhance and improve on what has been presented here is next. Finally, this thesis ends with some final remarks.

## 7.1 Thesis Summary

In recent years, DNNs have continuously set new state-of-the-art performance in numerous complex tasks that conventional methods struggled to solve, two of which have been explored in this thesis: image classification and machine translation. However, to a certain extent these breakthroughs have become possible through adding layers and increasing the size of the networks [143]. One drawback of the resulting DNNs is the increase in computational complexity, and therefore runtime [9]. Furthermore, many of the applications of DNNs are also of interest to mobile and embedded systems in order to bring forward advances such as automated driving, augmented reality, and more intelligent personal assistants. However, the resultant complexity of the state-of-the-art DNNs often mean it is prohibitive to run the model on-device, due to long inferencing times and power consumption, yet off-loading computation leads to privacy concerns, connectivity issues, and long latency for results.

Mobile and embedded systems manufacturers have helped to alleviate this problem by designing devices with multiple CPU cores of different levels of energy efficiency, and a GPU capable of DNN inference [124]. Popular mobile phone operating systems are also helping, iOS and Android now natively support DNN applications through CoreML [133] and TensorFlow Lite [81], respectively. Furthermore, the NVIDIA Jetson TX2, an

embedded deep learning platform, was designed specifically for embedded applications that require high performance computing [30]; designed to run GPU workloads on-device in a fast and energy efficient manner [112]. However this is not enough, the most complex state-of-the-art DNNs are still slow on mobile and embedded devices. For example, ResNet_v2_152, a very accurate DNN for image classification, takes around 2 seconds to make a prediction on a single image on the NVIDIA Jetson TX2 [130].

This thesis offered a novel software-based approach to executing DNN inference on embedded systems, aiming to reduce end-to-end inference runtime without a loss in accuracy. In fact, as a by-product, by utilising multiple DNN models it is possible to actually increase end-to-end accuracy across a dataset such as the ImageNet ILSVRC 2012 dataset; even when compared to state-of-the-art models such as ResNet_v2_152. The key insight in this thesis is that there is no one-size-fits-all model for all inputs, the optimal model - the one that is able to give the correct input in the fastest time - varies as the input and evaluation criteria changes. Furthermore, given the quickly evolving nature of DNN research and its many applications, the best selection strategy is likely to change over time. Therefore, an adaptive strategy that is able to learn the best DNN for each input and evaluation criteria is required; that is what this thesis aimed to supply.

The work in this thesis can be broken down into two main components: a Model Selector, and DNN specialisation; summarised below:

**Model Selector.** Alone, the model selector leverages multiple off-the-shelf pre-trained DNNs through the use of a premodel that dynamically selects the optimal DNN to use, at runtime, depending on the model input and evaluation criterion. An automatic approach to premodel generation, feature selection, and premodel tuning was presented. The Model Selector approach was applied to two typical DNN application domains: image classification and machine translation, which utilise convolutional and recurrent neural network architectures, respectively. Experimental results show that such an approach is able to deliver portable good performance across application domains and neural network architectures. For image classification, an overall top-1 accuracy of is 87.44% is achieved, translating to a 7.52% increase over the most-capable single DNN model, while reducing inference time by 44.45%. For machine translation, inference time is reduced by 25.37% over the most-capable model with negligible impact on the quality of the translation. With more training data an Oracle premodel (one that is 100% accurate) could achieve the same reduction in inference time while increasing F1 measure by 20.51%.

**DNN Specialisation.** To further advance the Model Selector approach, a method of starting with a single seed DNN and generating a pool of smaller, specialised DNNs, designed to work together was suggested, termed DNN specialisation. The smaller, specialised DNNs are designed to be inference-time-optimised while not causing a drop in end-to-end accuracy when used in conjunction with a Model Selector. The key insight

being that different parts of a DNN learn and utilise different features of the input which are used to larger or smaller degrees in classifying particular inputs [103, 98]; in other words, every image does not use every part of a DNN. An automatic approach to split the DNN training data into segments was proposed, each training data segment being used to train and specialise a DNN. A premodel is then trained to choose between the specialised DNNs. Due to the lack of available machine translation training data, this was only evaluated on the image classification application domain. Unfortunately, during experimentation a premodel struggled to accurately predict the correct DNN to use. It is not clear why an accurate premodel could not be trained, potential issues and their solutions are discussed further in Section 7.3. Looking at an Oracle premodel can give an indication of the potential of such an approach, should an accurate enough premodel be trained. Considering an Oracle premodel, DNN specialisation has the potential to further reduce resource utilisation over using a Model Selector alone, while reducing inference time and increasing inference accuracy by 5.15%.

## 7.2 Revisiting The Research Questions

**[RQ 1]** *By combining multiple DNNs, is it possible to reduce the average inference time and computational cost across a dataset without causing a reduction in accuracy? Moreover, how much can inference time be reduced by?*

This thesis has conclusively shown that the average inference time and computational cost can be reduced across a dataset by combining the capabilities of multiple DNNs. Moreover, this thesis has shown that it is also possible to increase end-to-end accuracy across a dataset. Key to this approach working is the utilisation of a diverse set of DNNs ranging in terms of computational complexity and inference time. In this way, the faster and computationally cheaper DNN can be utilised for every input it is capable of correctly classifying, a more expensive DNN is then only employed when required. Furthermore, the possible reduction in inference time depends on the inference time range of DNNs available and the cost of the premodel used. This question is answered in the following places:

- Section 4.1.1, specifically Figure 4.1, first shows the potential of using multiple off-the-shelf DNNs. It shows that multiple DNNs are capable of classifying the same images, and if the fastest capable DNN is chosen for each image then the average inference time can be reduced. Furthermore, it shows that accuracy scores are not effected by using multiple DNNs.
- Section 4.2.1, specifically Equation 4.2, formalises the potential average inference time gain depending on the DNNs available, and the cost of a premodel. The

equation shows that the best way to retain a high accuracy while reducing inference time is to include a range of DNNs of varying complexity. The faster the fastest DNN is, the better, and the more accurate the most accurate DNN is, the better.

- By analysing the performance of an Oracle premodel, Section 6.1.1 quantitatively shows the full potential of utilising multiple DNNs when applied to image classification. This section shows that average inference time can be reduced by 45%, while increasing top-1 and top-5 scores by 14.48% (to 91.2%) and 3.16% (to 98.3%), respectively, when compared to the single most capable DNN: ResNet_v1_152.

- Similar to image classification, Section 6.1.2 analyses an Oracle premodel to show the full potential of utilising multiple DNNs when applied to machine translation. This sections shows that the end-to-end accuracy can potentially be increased by 25.7%, while decreasing average inference time by 24.5%.

**[RQ 2]** *Is it possible to train a statistical machine learning model to choose the optimal DNN, at runtime, depending on the input and precision requirement?*

This thesis has shown that it is possible to train a SML model, termed premodel in this thesis, capable of choosing the optimal DNN at runtime; a new model needs to be trained for different precision requirements. Such an approach requires that all component DNNs that the premodel chooses from must be held in memory for fast inferencing. Therefore the cost of the proposed solution is an increase in memory usage. This research question has been evaluated in a similar way to RQ 1, with a small distinction. RQ 1 focusses on the maximum potential accuracy increase and inference time decrease, whereas this research question is evaluated based on the achieved values when utilising a premodel. This question is answered in the following places:

- Section 6.1.1 analyses the end-to-end performance of an implemented premodel when applied to image classification. This premodel achieves very close to perfect performance, achieving a prediction accuracy of 98%. With such high accuracy the premodel achieves close to the maximum potential premodel accuracy at a 7.52% increase in top-1, and a 2.9% increase in top-5 accuracy, when compared to the single most capable DNN. A 44.45% reduction in average inference time is also achieved.

- Section 6.1.2 analyses the end-to-end performance of an implemented premodel when applied to machine translation. In this case, the premodel is not as successful, achieving a 25.37% decrease in average inference time, with no significant impact on the end-to-end accuracy; a big difference from the potential 24.5% decrease

in average inference time, and 25.7% increase in end-to-end accuracy. Lower performance of the machine translation premodel can be attributed to significantly less premodel training data being available, 10x to be exact.

- Section 6.2.3 discusses the premodel with respect to DNN specialisation and image classification. In this Section it is not clear if the fault of low premodel accuracy lies with the premodel or the underlying choice of data segmentation. However, the analysis presented in Section 6.2.2 implies that the fault lies in data segmentation. Furthermore, given the positive results achieved using pre-trained off-the-shelf DNNs, this point is further reinforced; it is likely that the premodel is not at fault here.

**[RQ 3]** *Can orthogonal DNN optimisation techniques such as model compression be used in conjunction with a statistical machine learning model to further reduce inference time without a cost in accuracy?*

This thesis has shown that typical DNN compression techniques can be used in conjunction with a premodel to reduce average inferencing time without reducing end-to-end accuracy. Other orthogonal DNN optimisation techniques, such as kernel computation optimisation, could be used alongside the work in this thesis as well, provided they do not effect DNN accuracy. Here, the technique involves using DNN to create multiple models of varying accuracies and complexities, then get the premodel to choose the optimal DNN for the task at hand. Similar to RQ 1, the potential gain in inference time is dependant on how much the original DNN is compressed by. Provided that the premodel is able to accurately choose between the new compressed DNNs and the original DNN, there will be no reduction in accuracy. This question is answered in the following places:

- Section 6.1.3, specifically Figure 6.17, analyses a simple approach of employing DNN compression techniques to produce a number of smaller and faster DNNs based on a starting model. Each DNN created is smaller, but it suffers a reduction in accuracy at the same time. Figure 6.17 shows that by employing DNN compression alongside a premodel, accuracy can be better maintained while reducing inference time slightly more than using DNN compression alone; therefore the proposed solution works even when multiple pre-trained DNNs are not available. Specifically, the closest comparable compression technique is improved upon by a 90 ms reduction in average inference time, with a 16.37% improvement in end-to-end accuracy, translating to a 1.76% accuracy reduction when compared to the starting DNN.

- Section 6.2.1 takes DNN compression one step further. Instead of simply compressing a starting DNN to different levels, this section attempts to smartly segment

the dataset in order to generate smaller and more specialised DNNs. While an accurate enough premodel could not be generated to utilise the specialised DNNs, this section shows the high potential of such an approach. With an accurate enough premodel, this approach is capable of reducing inference time while maintaining, and even sometimes improving, on end-to-end accuracy. Such an approach is potentially even more successful than simple DNN compression alone.

**[RQ 4]** *Can a set of DNNs be generated that are optimised to work together, when combined with a statistical machine learning model, that achieve even further reductions in computational costs and inference times?*

Finally, this thesis investigated the potential of automatic ensemble creation, generating a pool of specialised DNNs for the premodel to choose from. Unfortunately, the results here are less conclusive. Evaluation of this work has shown clearly that such an approach has the potential to yield further reductions in computational costs and inference times while maintaining accuracy. However, the work in this thesis was unable to create a premodel accurate enough to utilise the generated specialised DNNs effectively. The following sections indicate the potential for automatic ensemble creation, and analyse the inner workings of the suggested approach, indicating where further work is needed to definitely answer this question.

- Section 6.2.1 shows the potential of automatic ensemble creation, indicating that with an accurate premodel inference time can be reduced and accuracy increased.
- Section 6.2.2 analyses the data segmentation component of DNN specialisation. This analysis indicates that data segmentation is to blame for the low premodel accuracy. The metrics used to evaluate each potential segmentation of the data are not sufficient to fully evaluate the potential end-to-end performance. Further work is needed here.
- Section 6.2.3 analyses the sub-DNN creation process component of DNN specialisation. This analysis indicates that, in itself, sub-DNN creation performs well, producing fast and highly accurate sub-DNNs (for their data segment). This section shows the high potential for DNN specialisation.
- Section 6.2.4 analyses a number of suggested premodel strategies. Here, the key take away is that all proposed premodels perform equally badly when evaluated end-to-end, even the powerful CNN based premodel. These results indicate that the problem the premodel is trying to solve is simply too complex, however this does not necessarily mean the premodel is to blame. Further analysis in this sections shows that there is a disparity in the data segmentations of the training dataset and

the validation dataset with regards to the best sub-DNN. Further investigation is needed here.

## 7.3 Future Work

Finally, this thesis finishes with a discussion of possible research directions that this work can continue in. The excellent work utilising pre-trained off-the-shelf DNNs suggest future work in: processor choice, and machine translation optimisation. The less conclusive results in automated ensemble creation suggest that there are a number of interesting areas that future work could investigate; evaluation of automated ensemble creation revealed the potential for improved optimisation over using off-the-shelf DNNs. Areas for improvement include: memory consumption, evaluating data segmentation, feature importance based data segmentation, and sub-DNN pruning optimisation. The future work is split into two sections, directly relating to the two main components of the work presented. First, research directions that would benefit and improve the accuracy or capabilities of a premodel are presented. To finish, further investigations into the DNN Specialisation process are discussed; potential methods to improve Oracle potential, and premodel accuracy.

### 7.3.1 Model Selector

**Computation Offloading.** This work focuses on accelerating inference on the current device. There are a number of drawbacks to computation offloading, however future research may find solutions to such issues. In such a case this work would still be beneficial, by only offloading work when there are no drawbacks. Future work could involve an environment with the opportunity to offload some of the computation to either cloud servers, edge devices, or even other mobile and embedded devices [25]. Accomplishing this would require a method to measure and predict network latency, allowing an educated decision to be made at runtime. SML techniques are shown to be effective in learning a cost function for profitability analysis [33]. This can be integrated with the current proposed learning framework.

**Processor choice.** By default, inference is carried out on the GPU, but this may not always be the best choice. Previous work has already shown SML techniques to be successful at selecting the optimal computing device [129]. This can be integrated into our existing learning framework. Furthermore, an interesting direction that the community has yet to explore may be to combine this work with computation offloading. This could result in a framework which is able to identify when it is beneficial to offload

computation, given the current state of the environment (which processors are free, or best for the task at hand).

**Machine Translation.** There is room for improvement in the machine translation `premodel`. During evaluation it was not possible to reach the full potential shown by the `Oracle`. To aid the `premodel` in reaching its full potential would require improving its accuracy; in order to achieve this, more training data is required. Only 5K sentences were available for machine translation, in comparison to 50k images for image classification. Furthermore, with more training data it would be possible to test and evaluate `DNN` specialisation on the machine translation problem, potentially revealing new insights into the research.

### 7.3.2 DNN Specialisation

**Memory Consumption.** The work in this thesis utilises multiple `DNN` models for inference, in comparison, the default method is to simply use a single model. Therefore, our approach requires more storage space. A solution for this would involve investigating how the models weights change during the sub-`DNN` Creation stage of `DNN` specialisation. The weights that show very little change between models could be locked at the same value and stored once for all sub-`DNNs`. This would be an especially effective tool for the weights at the input end of each sub-`DNN` as they are not fine-tuned due to them learning basic features relevant to all inputs. The result of this is numerous models sharing many weights in common, allowing the cost of using multiple models to be amortised.

**Evaluating Data Segmentation.** Evaluation revealed that the scoring methods used in order to evaluate and choose the best data segmentation are not sufficient. To this end, it would be useful to develop an approach to predict or infer the `premodel` accuracy and/or end-to-end top-1/top-5 accuracy scores based on a data segmentation. This is a non-trivial task, a large enough set of data segmentations would need to be fully evaluated from end-to-end, requiring sub-`DNNs` to be created, and their achieved scores recorded. The recorded data will then need analysing to reveal the key indicators of a good data segmentation. Furthermore, if such a method of segmentation analysis is cheap enough, then a smarter or more exhaustive search can be carried out to find better data segmentations for each `DNN` application domain.

**Feature Importance Based Data Segmentation.** Extending a Model Selector to include `DNN` specialisation did not appear to work during the evaluation presented in this thesis. Due to the number of moving parts in the approach, it is difficult to narrow down precisely what the cause of the problem is; one reason could be a bad data segmentation. If an effective method of evaluating the importance of each neuron for each image is

developed, it could be used for data segmentation. It makes intuitive sense, based on the motivation presented in Section 4.1.2, that this would work; the goal of data segmentation is to group images together which agree that the same neurons are unimportant, so they can be pruned. However, this is a non-trivial task. For a relatively small DNN such as ResNet_v2_50 this would result in 26500 features, one for each convolutional filter. With so many features, for even a small DNN, it could lead to an in-accurate model that is prone to over-fitting. Therefore, this method requires some work to summarise the features in a way that makes sense to the data it represents.

**Sub-DNN Pruning Optimisation.** During evaluation of DNN specialisation,| pruning of the sub-DNNs was required in order to achieve an overall end-to-end inference speed up; however the level of pruning was not deeply investigated. An interesting research direction that is yet to be explored may be an investigation into the perfect level of pruning to use per sub-DNN. That is, the level of pruning that reduces inference time without a loss in accuracy. Further interesting research topics involve: investigating how the ideal level or pruning changes layer to layer, and sub-DNN to sub-DNN; and an analysis of how the perfect level of pruning varies as the number of data segments increases or decreases. As with much research in pruning, it is expected that more pruning leads to a higher drop in accuracy. However, in this case the user would simply be sacrificing the accuracy gains achieved by using multiple DNNs; potentially leading to even further pruning, without a loss in end-to-end accuracy, than conventional methods.

## 7.4 Final Remarks

In summary, this thesis presented a novel approach to DNN inference optimisation. By utilising multiple DNN models, and choosing the best one to use at runtime, the average inference time can be greatly reduced, without reducing accuracy; in fact, accuracy can even be increased in cases such as image classification. The drawback of the proposed approach is an increase in memory usage, however it was not an issue on the evaluation platform used. Furthermore, the work was extended to generate a pool of smaller, specialised DNNs from a starting single seed DNN; the generated DNNs are designed to work together. Unfortunately, the evaluation of automated ensemble creation were less conclusive, however primary results indicate the potential for a further reduction in inference time and resource consumption over a method using pre-trained off-the-shelf DNNs. Finally, a number of directions have been suggested in order to improve on the suggested automated ensemble creation, and reach its full potential. With the success of the suggested future work the work presented in this thesis would help bring forward fast and accurate DNN inference on embedded an mobile devices.

# References


[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Amodei, D. et al. (2016). Deep speech 2: End-to-end speech recognition in english and mandarin. In *ICML*.

[3] Anwar, S., Hwang, K., and Sung, W. (2015). Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1131–1135. IEEE.

[4] Bahdanau, D. et al. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[5] Bai, J., Li, Y., Li, J., Jiang, Y., and Xia, S. (2019). Rectified decision trees: Towards interpretability, compression and empirical soundness. *arXiv preprint arXiv:1903.05965*.

[6] Bellec, G., Kappel, D., Maass, W., and Legenstein, R. (2017). Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*.

[7] Bhattacharya, S. and Lane, N. D. (2016). Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *SenSys*.

[8] Breaban, M. and Luchian, H. (2011). A unifying criterion for unsupervised clustering and feature selection. *Pattern Recognition*, 44(4):854–865.

[9] Canziani, A. et al. (2016). An analysis of deep neural network models for practical applications. *CoRR*.

[10] Casolla, G., Cuomo, S., Di Cola, V. S., and Piccialli, F. (2019). Exploring unsupervised learning techniques for the internet of things. *IEEE Transactions on Industrial Informatics*.

[11] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400.

[12] Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. (2015). Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294.

[13] Cheng, W., Kasneci, G., Graepel, T., Stern, D., and Herbrich, R. (2011). Automated feature generation from structured knowledge. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1395–1404. ACM.

[14] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[15] Chollet, F. et al. (2015). Keras. https://keras.io.

[16] Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M., and Yang, S. (2017). Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org.

[17] Courbariaux, M., Bengio, Y., and David, J.-P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131.

[18] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*.

[19] Delimitrou, C. and Kozyrakis, C. (2014). Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144. ACM.

[20] Deng, L., Li, J., Huang, J.-T., Yao, K., Yu, D., Seide, F., Seltzer, M., Zweig, G., He, X., Williams, J., et al. (2013). Recent advances in deep learning for speech research at microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8604–8608. IEEE.

[21] Deng, Y. (2019). Deep learning on mobile devices: a review. In *Mobile Multimedia/Image Processing, Security, and Applications 2019*, volume 10993, page 109930A. International Society for Optics and Photonics.

[22] Dettmers, T. and Zettlemoyer, L. (2019). Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*.

[23] Devan, G. (2020). Smartphone statistics for 2020. https://techjury.net/stats-about/smartphone-usage/. Accessed: 20/05/20.

[24] Donahue, J. et al. (2014). Decaf: A deep convolutional activation feature for generic visual recognition. In *ICML*.

[25] Elkhatib, Y., Porter, B., Ribeiro, H. B., Zhani, M. F., Qadir, J., and Rivière, E. (2017). On using micro-clouds to deliver the fog. *IEEE Internet Computing*, 21(2):8–15.

[26] Emani, M. K. and O'Boyle, M. (2015). Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments. In *ACM SIGPLAN Notices*, volume 50, pages 499–508. ACM.

[27] EMNLP (2015). Emnlp 2015 tenth workshop on statistical machine translation. shared task: Machine translation. https://www.statmt.org/wmt15/translation-task.html.

[28] Eshratifar, A. E. and Pedram, M. (2018). Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 111–116. ACM.

[29] Fang, B., Zeng, X., and Zhang, M. (2018). Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127. ACM.

[30] Franklin, D. (2017). Nvidia jetson tx2 delivers twice the intelligence to the edge. *NVIDIA Accelerated Computing| Parallel Forall*.

[31] Gale, T., Elsen, E., and Hooker, S. (2019). The state of sparsity in deep neural networks. *CoRR*, abs/1902.09574.

[32] Georgiev, P., Bhattacharya, S., Lane, N. D., and Mascolo, C. (2017). Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–19.

[33] Grewe, D., Wang, Z., and O'Boyle, M. F. (2013). Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE.

[34] Guo, T. (2017). Towards efficient deep inference for mobile applications. *CoRR*, abs/1707.04610.

[35] Han, S. et al. (2016). Eie: efficient inference engine on compressed deep neural network. In *ISCA*.

[36] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[37] Han, S., Pool, J., Tran, J., and Dally, W. (2015b). Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143.

[38] Hanson, S. J. and Pratt, L. Y. (1989). Comparing biases for minimal network construction with back-propagation. In *Advances in neural information processing systems*, pages 177–185.

[39] Harrison, B., Purdy, C., and Riedl, M. O. (2017). Toward automated story generation with markov chain monte carlo methods and deep neural networks. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[40] Hassaballah, M., Abdelmgeid, A. A., and Alshazly, H. A. (2016). Image features detection, description and matching. In *Image Feature Detectors and Descriptors*, pages 11–45. Springer.

[41] Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171.

[42] Hastie, T., Tibshirani, R., Friedman, J., and Franklin, J. (2005). The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2).

[43] He, K. et al. (2016a). Deep residual learning for image recognition. In *CVPR*.

[44] He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer.

[45] He, T., Fan, Y., Qian, Y., Tan, T., and Yu, K. (2014). Reshaping deep neural network for fast decoding by node-pruning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 245–249. IEEE.

[46] He, Y., Kang, G., Dong, X., Fu, Y., and Yang, Y. (2018). Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*.

[47] Hensman, P. and Masko, D. (2015). The impact of imbalanced training data for convolutional neural networks. *Degree Project in Computer Science, KTH Royal Institute of Technology*.

[48] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[49] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019). Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*.

[50] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[51] Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. (2016). Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

[52] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708.

[53] Huynh, L. N., Lee, Y., and Balan, R. K. (2017). Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95. ACM.

[54] Hwang, K. and Sung, W. (2014). Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE.

[55] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*.

[56] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713.

[57] Kaggle (2019). Kaggle's state of data science and machine learning 2019. *https://www.kaggle.com/kaggle-survey-2019*.

[58] Kang, Y. et al. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*.

[59] Kaufman, L. and Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons.

[60] Ketchen, D. J. and Shook, C. L. (1996). The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal*, 17(6):441–458.

[61] Khoo, A., Marom, Y., and Albrecht, D. (2006). Experiments with sentence classification. In *Proceedings of the Australasian Language Technology Workshop 2006*, pages 18–25.

[62] Khoram, S. and Li, J. (2018). Adaptive quantization of neural networks. In *International Conference on Learning Representations*.

[63] Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2016). Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*.

[64] Kontschieder, P., Fiterau, M., Criminisi, A., and Rota Bulo, S. (2015). Deep neural decision forests. In *Proceedings of the IEEE international conference on computer vision*, pages 1467–1475.

[65] Krizhevsky, A., Nair, V., and Hinton, G. (2014). The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55.

[66] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *NIPS*.

[67] Lane, N. D. et al. (2016). Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *IPSN*.

[68] Lane, N. D. and Warden, P. (2018). The deep (learning) transformation of mobile and embedded computing. *Computer*, 51(5):12–16.

[69] Laros III, J. H., Pedretti, K., Kelly, S. M., Shu, W., Ferreira, K., Van Dyke, J., and Vaughan, C. (2012). *Energy-efficient high performance computing: measurement and tuning*. Springer Science & Business Media.

[70] Latifi Oskouei, S. S., Golestani, H., Hashemi, M., and Ghiasi, S. (2016). Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1201–1205.

[71] Leather, H., Bonilla, E., and O'Boyle, M. (2009). Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91. IEEE Computer Society.

[72] LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2:18.

[73] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

[74] Lee, H. et al. (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *NIPS*.

[75] Lehmann, E. L. and Casella, G. (2006). *Theory of point estimation*. Springer Science & Business Media.

[76] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

[77] Lin, C.-Y., Cao, G., Gao, J., and Nie, J.-Y. (2006). An information-theoretic approach to automatic evaluation of summaries. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 463–470. Association for Computational Linguistics.

[78] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

[79] Lin, S., Ji, R., Yan, C., Zhang, B., Cao, L., Ye, Q., Huang, F., and Doermann, D. (2019). Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2790–2799.

[80] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[81] Louis, M. S., Azad, Z., Delshadtehrani, L., Gupta, S., Warden, P., Reddi, V. J., and Joshi, A. (2019). Towards deep learning using tensorflow lite on risc-v. *Proc. ACM CARRV*.

[82] Lui, M. (2012). Feature stacking for sentence classification in evidence-based medicine. In *Proceedings of the Australasian Language Technology Association Workshop 2012*, pages 134–138.

[83] Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE.

[84] Luo, J.-H., Zhang, H., Zhou, H.-Y., Xie, C.-W., Wu, J., and Lin, W. (2018). Thinet: pruning cnn filters for a thinner net. *IEEE transactions on pattern analysis and machine intelligence*.

[85] Luong, M.-T., Brevdo, E., and Zhao, R. (2017). Neural machine translation (seq2seq) tutorial. *https://github. com/tensorflow/nmt*.

[86] Lysiak, R., Kurzynski, M., and Woloszynski, T. (2014). Optimal selection of ensemble classifiers using measures of competence and diversity of base classifiers. *Neurocomputing*, 126:29–35.

[87] Magdy, W., Elkhatib, Y., Tyson, G., Joglekar, S., and Sastry, N. (2017). Fake it till you make it: Fishing for catfishes. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 497–504.

[88] Marco, V. S., Taylor, B., Porter, B., and Wang, Z. (2017). Improving spark application throughput via memory aware task co-location: a mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 95–108. ACM.

[89] Marco, V. S., Taylor, B., Wang, Z., and Elkhatib, Y. (2020). Optimizing deep learning inference on embedded systems through adaptive model selection. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(1):1–28.

[90] Markov, A. A. (1971). Extension of the limit theorems of probability theory to a sum of variables connected in a chain. *Dynamic probabilistic systems*, 1:552–577.

[91] Martins, L. G., Nobre, R., Delbem, A. C., Marques, E., and Cardoso, J. M. (2014). Exploration of compiler optimization sequences using clustering-based selection. In *ACM SIGPLAN Notices*, volume 49, pages 63–72. ACM.

[92] Mattson, P., Reddi, V. J., Cheng, C., Coleman, C., Diamos, G., Kanter, D., Micikevicius, P., Patterson, D., Schmuelling, G., Tang, H., et al. (2020). Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16.

[93] McMahan, B., R. D. (2017). Federated learning: Collaborative machine learning without centralized training data. https://github.com/BenWhetton/keras-surgeon. Accessed: 20/05/20.

[94] Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*.

[95] Moreau, T., Chen, T., Vega, L., Roesch, J., Zheng, L., Yan, E., Fromm, J., Jiang, Z., Ceze, L., Guestrin, C., et al. (2019). A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*.

[96] Motamedi, M., Fong, D., and Ghiasi, S. (2017). Machine intelligence on resource-constrained iot devices: The case of thread granularity optimization for cnn inference. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):151.

[97] Nystrup, P., Madsen, H., and Lindström, E. (2015). Stylised facts of financial time series and hidden markov models in continuous time. *Quantitative Finance*, 15(9):1531–1541.

[98] Olah, C., Mordvintsev, A., and Schubert, L. (2017). Feature visualization. *Distill*, 2.

[99] Osia, S. A., Shamsabadi, A. S., Taheri, A., Katevas, K., Sajadmanesh, S., Rabiee, H. R., Lane, N. D., and Haddadi, H. (2017). A hybrid deep learning architecture for privacy-preserving mobile analytics. *arXiv preprint arXiv:1703.02952*.

[100] Pandit, A. A., Pimpale, B., and Dubey, S. (2019). A comprehensive review on unsupervised feature selection algorithms. In *International Conference on Intelligent Computing and Smart Communication 2019: Proceedings of ICSC 2019*. Springer Nature.

[101] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

[102] Parkhi, O. M. et al. (2015). Deep face recognition. In *BMVC*.

[103] Qin, Z., Yu, F., Liu, C., and Chen, X. (2018). How convolutional neural network see the world-a survey of convolutional neural network visualization methods. *arXiv preprint arXiv:1804.11191*.

[104] Qiu, X., Zhang, L., Ren, Y., Suganthan, P. N., and Amaratunga, G. (2014). Ensemble deep learning for regression and time series forecasting. In *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*, pages 1–6. IEEE.

[105] Rallapalli, S. K. et al. (2016). Are very deep neural networks feasible on mobile devices? Technical report, University of Southern California.

[106] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer.

[107] Ravi, S. (2017). Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630*.

[108] Ren, J. et al. (2017). Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM*.

[109] Rethinagiri, S. K., Atitallah, R. B., and Dekeyser, J.-L. (2011). A system level power consumption estimation for mpsoc. In *2011 International Symposium on System on Chip (SoC)*, pages 56–61. IEEE.

[110] Riera, M., Arnau, J.-M., and Gonzalez, A. (2019). (pen-) ultimate dnn pruning. *arXiv preprint arXiv:1906.02535*.

[111] Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65.

[112] Rungsuptaweekoon, K., Visoottiviseth, V., and Takano, R. (2017). Evaluating the power efficiency of deep learning inference on embedded gpu systems. In *2017 2nd International Conference on Information Technology (INCIT)*, pages 1–5. IEEE.

[113] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252.

[114] Samreen, F. et al. (2016). Daleel: Simplifying cloud instance selection using machine learning. In *NOMS*.

[115] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626.

[116] Servia-Rodriguez, S., Wang, L., Zhao, J. R., Mortier, R., and Haddadi, H. (2017). Personal model training under privacy constraints. *training*, 40(33):24–38.

[117] Shafer, G. and Vovk, V. (2008). A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9(Mar):371–421.

[118] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[119] Song, M., Hu, Y., Chen, H., and Li, T. (2017). Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE.

[120] Srinivas, S. and Babu, R. V. (2015). Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*.

[121] Stahlberg, F., de Gispert, A., and Byrne, B. (2018). The university of cambridge's machine translation systems for wmt18. *arXiv preprint arXiv:1808.09465*.

[122] Stahlberg, F., de Gispert, A., Hasler, E., and Byrne, B. (2016). Neural machine translation by minimising the bayes-risk with respect to syntactic translation lattices. *arXiv preprint arXiv:1612.03791*.

[123] Sun, Y., Chen, Y., et al. (2014). Deep learning face representation by joint identification-verification. In *NIPS*.

[124] Swain, M., Singh, R., Hashmi, M. F., and Gehlot, A. (2020). Performance analysis of various embedded linux firmwares for arm architecture based iot devices. In *International Conference on Intelligent Computing and Smart Communication 2019*, pages 1451–1460. Springer.

[125] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[126] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826.

[127] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828.

[128] Tan, M. and Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*.

[129] Taylor, B., Marco, V. S., and Wang, Z. (2017). Adaptive optimization for opencl programs on embedded heterogeneous systems. In *ACM SIGPLAN Notices*, volume 52, pages 11–20. ACM.

[130] Taylor, B., Marco, V. S., Wolff, W., Elkhatib, Y., and Wang, Z. (2018). Adaptive deep learning model selection on embedded systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–43. ACM.

[131] Teerapittayanon, S. et al. (2017). Distributed deep neural networks over the cloud, the edge and end devices. In *ICDCS*.

[132] Teklehaymanot, F. K., Muma, M., and Zoubir, A. M. (2018). Bayesian cluster enumeration criterion for unsupervised learning. *IEEE Transactions on Signal Processing*, 66(20):5392–5406.

[133] Thakkar, M. (2019). Introduction to core ml framework. In *Beginning Machine Learning in iOS*, pages 15–49. Springer.

[134] Tuytelaars, T., Mikolajczyk, K., et al. (2008). Local invariant feature detectors: a survey. *Foundations and trends® in computer graphics and vision*, 3(3):177–280.

[135] Usama, M. et al. (2017). Unsupervised machine learning for networking: Techniques, applications and research challenges. *CoRR*, abs/1709.06599.

[136] Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus.

[137] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. (2016). Ernest: efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 363–378.

[138] Wang, J., Cao, B., Yu, P., Sun, L., Bao, W., and Zhu, X. (2018). Deep learning towards mobile applications. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1385–1393. IEEE.

[139] Wang, Z. et al. (2014). Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO*.

[140] Wang, Z., Grewe, D., and O'boyle, M. F. (2015). Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):42.

[141] Wang, Z. and O'Boyle, M. (2018). Machine learning in compiler optimisation. *Proc. IEEE*.

[142] Wang, Z. and O'Boyle, M. F. (2010). Partitioning streaming parallelism for multicores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 307–318. ACM.

[143] Wei, C. and Ma, T. (2019). Data-dependent sample complexity of deep neural networks via lipschitz augmentation. In *Advances in Neural Information Processing Systems*, pages 9722–9733.

[144] Wen, G., Hou, Z., Li, H., Li, D., Jiang, L., and Xun, E. (2017). Ensemble of deep neural networks with probability-based fusion for facial expression recognition. *Cognitive Computation*, 9(5):597–610.

[145] Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082.

[146] Whetton, B. (2017). Keras surgeon. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html. Accessed: 25/05/20.

[147] Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. (2016a). Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828.

[148] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016b). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[149] Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500.

[150] Yazdani, R., Riera, M., Arnau, J.-M., and González, A. (2018). The dark side of dnn pruning. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–801. IEEE.

[151] You, Y., Zhang, Z., Hsieh, C.-J., Demmel, J., and Keutzer, K. (2018). Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM.

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

[153] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

[154] Zeiler, M. D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q. V., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., et al. (2013). On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE.

[155] Zheng, J., Yang, W., and Li, X. (2017). Training data reduction in deep neural networks with partial mutual information based feature selection and correlation matching based active learning. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2362–2366. IEEE.

[156] Zhou, Z.-H. and Feng, J. (2017). Deep forest. *arXiv preprint arXiv:1702.08835*.

[157] Zöller, M.-A. and Huber, M. F. (2019). Survey on automated machine learning. *arXiv preprint arXiv:1904.12054*, 9.

[158] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.

# Appendix A

# DNN Specialisation Feature Sets

Table A.1 The features that make up each of the feature sets used in used in this thesis. All Features are described in full in Table 6.1.

| Feature Set | Features | | | | |
|---|---|---|---|---|---|
| **fid_171** | *aspect_ratio* | *edge_angle1* | *edge_angle2* | *edge_angle3* | *hue7* |
| | *n_keypoints* | *edge_angle4* | *edge_angle5* | *hue1* | |
| **fid_187** | *aspect_ratio* | *area_by_perim* | *edge_angle2* | *edge_angle3* | *hue7* |
| | *n_keypoints* | *hue1* | | | |
| **fid_190** | *aspect_ratio* | *area_by_perim* | *edge_angle2* | *edge_angle3* | *hue7* |
| | *n_keypoints* | *edge_angle5* | | | |
| **fid_193** | *aspect_ratio* | *area_by_perim* | *edge_angle3* | *hue1* | *hue7* |
| | *n_keypoints* | | | | |
| **fid_199** | *aspect_ratio* | *area_by_perim* | *edge_angle3* | *hue1* | *hue7* |
| **fid_228** | *aspect_ratio* | *edge_angle1* | *edge_angle3* | *edge_angle5* | *hue7* |
| | *n_keypoints* | | | | |
| **fid_233** | *aspect_ratio* | *edge_angle1* | *edge_angle3* | *edge_angle5* | *hue7* |
| **fid_320** | *aspect_ratio* | *area_by_perim* | *edge_angle2* | *edge_angle4* | *hue7* |
| | *n_keypoints* | | | | |
| **fid_373** | *aspect_ratio* | *edge_angle1* | *edge_angle3* | *edge_angle5* | *hue7* |
| | *area_by_perim* | *hue1* | | | |
| **fid_373** | *aspect_ratio* | *area_by_perim* | *edge_angle1* | *edge_angle3* | *hue7* |
| | *n_keypoints* | *hue1* | | | |