UNIVERSITY OF SOUTHAMPTON

Progressive Intelligence for Low-Power Devices

by

Jack Dymond

A thesis submitted in fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

August 2024

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Progressive Intelligence for Low-Power Devices

by Jack Dymond

As our world has become increasingly advanced so too has the pervasiveness of technology throughout it. This has led to a desire to have these technological capabilities embedded in the devices we interact with on a daily basis, this spans from entire industrial sectors down to individual consumer devices. Artificial intelligence, in particular machine learning, is one of these advancements which is progressing at an unprecedented rate. However, it is also becoming incredibly resource intensive to operate. Progressive intelligence is a solution to this issue, which approaches the machine learning inference process incrementally, first making low-cost–low-confidence predictions and improving on these incrementally according to application requirements.

This work explores progressive intelligence, and discusses where it fits in the machine learning field. Machine learning approaches can be split into data-based and model-based approaches, it is shown progressive intelligence can be implemented in both. In the data-based paradigm an upper bound on error as the number of samples increases is derived. In the model-based paradigm branched neural networks are selected as a good system on which to implement progressive intelligence ideas. A comprehensive study of progressive intelligence in branched neural networks is undertaken, from their training to their inference processes. Following this, the novel concept of an early-exit reject is introduced, where three scenarios are studied in which samples can be rejected at inference time to save on computational resources.

Contents

Acknowledgements

I would like to extend my deepest gratitude to everyone who has supported me throughout this journey. To my mum, for inspiring me to push myself academically, and to all of my family for supporting me when I have needed them. To my fiancé, Annabelle Culling, for your love, patience, and understanding, and to the Culling family as a whole for their kindness and support over the last 4 years.

I am thankful to Dstl for the opportunities they have afforded me throughout this process and for their financial support which made this work possible. I also want to express my appreciation to the students and management staff of the CPS group, the CCAIS team, and the MINDS CDT, for fostering a collaborative environment which has allowed myself along with every student to flourish.

Finally, a special thank you to my supervisors, Professors Steve Gunn and Sebastian Stein, whose guidance, expertise, and mentorship have been invaluable to my research and personal growth.

Thank you all for your indispensable contributions to this work.

Chapter 1

Introduction

1.1 Machine Learning: Its Successes and Shortcomings

Over the last decade the amount of data that is available in our society has been increasing exponentially (Duarte, 2023). From football match statistics to personal information, data has become another currency with which companies compete with one another. This data is often unstructured, but even when structured the volume will often require significant resources and infrastructure to enable it to be used effectively.

This has meant the demand for analysts and data scientists has increased across every industry, both in academic and commercial environments. This heightened demand has spurred significant advancements in the tools and algorithms they employ, thus leading the field of machine learning into a period of unprecedented growth.

Heavy investment from tech giants such as Google and Microsoft, newcomers such as DeepMind and OpenAI, and collaborations with universities has allowed for greater acceleration in this progress. Similarly, hardware manufacturers such as ARM and Nvidia have shifted towards giving AI a central role in their business models. Hence, both ends of the technology industry are contributing resources to the field helping facilitate this surge in progress.

While the amount of data being collected increases, so too do issues surrounding its storage and usage. As data becomes more abundant, rich, and complex, our understanding of that data and how it should be used becomes more important (Abdallah et al., 2023). Broadly speaking there are two ways we can approach this problem.

One approach is to focus on the data and distil it down, focusing on what it is we want to understand from it. Some features will have greater importance on the patterns we wish to discover, some will be irrelevant. Some inputs will be particularly useful, others might simply fill in the gaps, some patterns will help us understand overarching relationships, whilst others might be coincidental.

The other aspect we can focus on is our learning machine, as we could instead scale our models accordingly. More complex models which can process more features will be able to parameterise a lot of this understanding. The drawback of this is that often these complex models are more difficult to understand, meaning our insights into the data will be limited.

So, these problems can be approached from either end, you wish to clean and understand our data to use it (Abdallah et al., 2023). But likewise, as the the datasets become increasingly high-dimensional our models need to scale accordingly (Dherin et al., 2022). Hence, in practice it is approached using a mixture of both.

Our approaches from the data side of this have been an active area of research for decades and are well understood in the field, with approaches such as k nearest-neighbours (KNN) being developed decades ago (Silverman and Jones, 1989). In contrast, our scaling of machine learning models has been a more recent advancement. For example AlexNet, published in the the last decade, can be considered a pivotal paper in the uptake of larger neural networks (Krizhevsky et al., 2012).

This has prompted much interest from many industries around the world. From the defence to the pharmaceutical sector, companies are forming the same academic alliances as tech companies to help improve the products and services they can offer using machine learning. Indeed, this has sparked many developments that will be influential for years to come. For example, DeepMind's *AlphaFold* presented by Senior et al. (2020) is a deep learning system that has solved the 50-year-old protein folding problem in biochemistry first proposed by Levinthal (1969).

However, an issue with the most successful models is their large parameter counts and computational complexity, which make them very cumbersome and difficult to train. To do so, large amounts of computational resources need to be used. Hence, tech giants with thousands of processing units at their disposal help dominate the frontier of the deep learning field, both in the results they can obtain and the ideas they are able to explore.

Their size also make them difficult to deploy once trained, as seen in GPT-4 for example, a language model capable of performing well on a variety of language tasks (Brown et al., 2020). These include correcting grammatical errors, using new words in sentences after being given definitions, and generating short 'news articles' which could hardly be distinguished from human-written articles . Although producing unprecedented results, this model is estimated to use over a trillion parameters and is an example of how over-parameterisation is distancing deep learning systems from being usable on consumer hardware. Similar models have been developed more recently which also have billions of parameters (Ramesh et al., 2021, 2022). That is, practical constraints in memory, training time, and even power usage, mean deploying such models is infeasible and creating them impossible without using large amounts of computational resources.

There will also be scenarios where machine learning systems need to be ran offline, and at the edge, as opposed to in a data center. This might be when there are privacy concerns with the data used by the model, or unreliable internet connectivity. Currently, many state-of-the-art approaches cannot be operated on edge devices.

So, if deep learning is to be moved to the devices where industries and consumers will benefit the most, models will need to be reduced in size and complexity, such that they can be stored and used on more attainable hardware. Furthermore, our use of large datasets will become intractable in the absence of these resources and attention should also be given to how best we can utilise this data at inference time.

This thesis will explore one solution to this problem in particular: Progressive intelligence. Progressive intelligence systems are machine learning systems which approach the inference process incrementally. They first make low-cost–low-confidence inference and improve upon this incrementally according to the requirements of the application scenario.

1.2 Progressive Intelligence

Progressive intelligence refers to the incremental nature of the solution, first making low-cost–low-confidence inference. Then, subject to operating conditions, it improves upon this progressively through further inference. Progressive intelligence can be compared to conventional models by representing them as pareto-fronts. A pareto-front is formed when trying to co-optimise two properties which negatively impact each other. Optimising a system for both properties in such an environment is known as a pareto-optimisation problem, for example, lower computational cost will typically degrade the accuracy of a model. Pareto-optimal refers to the optimum solution in a pareto-optimisation problem and it is a key idea in progressive intelligence.

Conventional machine learning models will operate at a discrete cost-performance point, whereas progressive intelligence models will operate along a continuous distribution in this space. This is shown in figure 1.1.

A key distinction between progressive intelligence models and conventional models is that the progressive intelligence models can adapt and operate across a range of power modes, at the expense of a small drop in performance when compared to a conventional model.

In the following, application areas, particular platforms, and the requirements of such systems will be detailed.

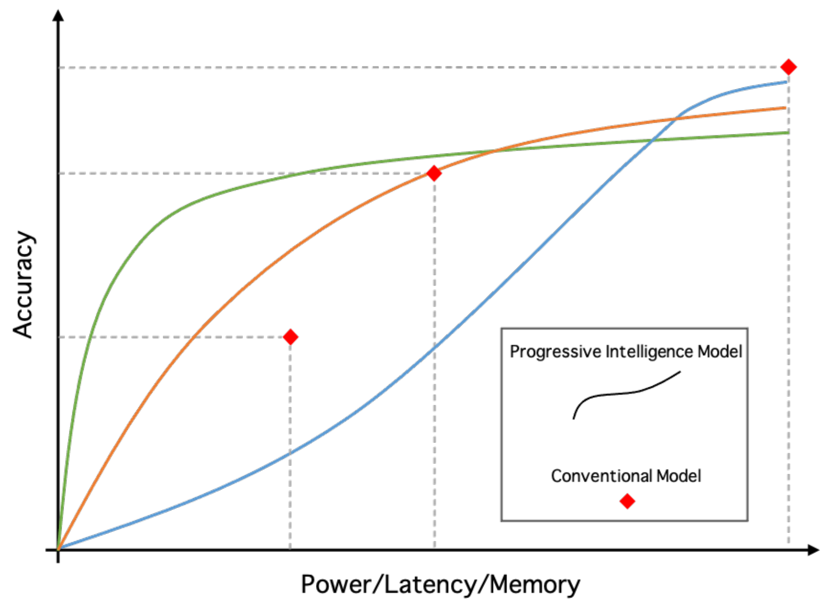


FIGURE 1.1: Accuracy-Cost space comparing progressive intelligence models and conventional models. Power, Latency, and Memory are examples of costs in machine learning systems. Three progressive intelligence models are depicted, each outperforming the others across different cost ranges. The green model prioritises early performance gains, whereas the blue model operates at a low performance before a sharp increase in accuracy at the highest power range. The orange model is a compromise between the two, and can be considered the closest to a pareto-optimal solution.

1.2.1 Application Areas and Platforms

The particular platform on which a machine learning model is implemented will depend on its application. For example one moderating social media posts can be run on a computing server, whereas a deployed land surveillance drone will run its algorithms on low-power, embedded platforms. Clearly, these two platforms have varying requirements, and as such, the deployed algorithm will need to conform to these requirements. Taking these two platforms as examples, we can explore this domain further with some case studies.

1.2.1.1 Case Study 1: Land Surveillance Drone

Drone technology has been improved significantly in the last decade, from search and rescue to domestic use, fixed wing and ‘quadcopter’ drones are becoming an increasingly pervasive solution to many of our problems. When coupled with artificial intelligence algorithms, these drones can be very powerful tools (Aoun et al., 2019; Immanuel Damanik et al., 2022).

However, regardless of their use-case, they will always be limited in their power availability. In fixed wing drones for example there are a wide range of power capabilities, from ~20W-60kW (Bertran and Sànchez-Cerdà, 2016; Hamza et al., 2019). Machine learning algorithms are commonly deployed on CPUs and deep learning algorithms on GPUs, using anywhere between ~60W and 500W. Hence, a considerable percentage of power available to a system will be used if these algorithms are to be deployed on them and operate at their fullest capacity. Memory will present less of a challenge in such applications, as the form factor of a drone permits necessary data storage.

Power usage will also vary dynamically, for example, solar powered drones may have unreliable power availability when flying in sub-optimal conditions and a quadcopter will use more power when flying in windy conditions due to the constant need to correct its path. It follows that deploying machine learning algorithms onto these platforms will compound these issues, particularly if the algorithm cannot adapt to the varying amount of power it can be afforded at any given time.

Therefore, it would be beneficial to deploy a machine learning algorithm which can adapt to changing environments with its platform. If the drone needs to stay out longer or is using an abnormal amount of power, the algorithm should decrease its power usage. If the drone is in an area of high importance, the algorithm should provide the most confident outputs it can, whilst this constraint can be relaxed in less important areas. Furthermore, some classes may have a greater importance than others, necessitating greater confidence and thus power usage in classification. Such an algorithm will increase the flight time of the system without compromising the confidences achieved in classification and will be adaptable to changing system requirements at run-time.

1.2.1.2 Case Study 2: Social Media Moderation

With social media platforms growing considerably since their inception, traditional moderation solutions are decreasing in efficacy and are being replaced by automated solutions. These automated solutions need to be run on computing platforms, which will often come in the form of cloud computing servers (Laaksonen et al., 2020; Jahan and Oussalah, 2023).

Cloud computing servers generally vary in cost dynamically depending on demand and availability, hence under a fixed financial budget the compute available will vary over time (Narayanan et al., 2020). Furthermore, if a social media platform does its computing in-house, the services that are sharing this resource will vary, meaning compute cannot be guaranteed. Hence the deployed algorithm will need to adapt dynamically to the compute available to it.

When the resource availability is at its greatest, a conventional algorithm will be best suited, which maximises its resource and with it accuracy. However, when resource availability decreases, a compromise needs to be made: either decrease the volume of posts you can moderate, or spend time loading a less demanding, but less accurate model across the available components. Neither of these solutions are optimal: one will increase the time a potentially damaging post can be active on the site, and the other might miss it entirely until it is flagged by a user on top of causing service downtime.

Clearly there is a potential here to improve such a system, by using a model which can adapt dynamically to such a situation, much like in the previous case study. One which can operate in a number of different modes, whilst also providing confident classifications where necessary. Such an algorithm would maximise the number of posts moderated under a fixed financial budget whilst minimising downtime.

Two case studies have been discussed which would benefit from machine learning algorithms that are adaptive and dynamic. The next section will cover the system requirements more specifically.

1.2.2 System Requirements

In order to operate on systems with various power availability modes, the machine learning system itself will be required to have a number of operating modes available which can be switched between at inference time. Furthermore, in the event that some inputs require a greater degree of confidence, or if some inputs are more difficult to classify than others, the system should have a degree of input dependence in these power modes. That is, given a certain input, the model should be able to dynamically determine the power mode with which to process it. However, this may lead to a scenario whereby an input is classified to an insufficient confidence level and inference needs to be repeated. Thus, to avoid repeated computation in this scenario, it will be beneficial for the model to complete the inference incrementally. Hence we have three core requirements for our system:

Multiple Operating Modes The system should have a number of operating points, which use increasing amounts of power to obtain increasingly confident results.

Input dependence The system should also be able to dynamically use these in an input-wise manner.

Incremental Inference Most importantly the system should never waste computation, if an input is not classified sufficiently, it should build upon its classification as opposed to restarting the inference process.

This thesis will focus on the development of such systems, with an application emphasis on the second case study, in section 1.2.1.1, due to the increased activity in, and accessibility of, the computer vision space. However, there is scope to apply progressive intelligence to other problems, as in the social media moderation case study.

There are a number of open questions in the progressive intelligence space, and a number of areas where development could be made. The next section will outline the specific research questions of this thesis, and the research goals that aim to answer them.

1.3 Research Questions and Goals

Progressive intelligence is a formulation of machine learning which can help provide a new lens on the way machine learning algorithms are designed, with efficiency and adaptability being the core principle of their design. However, to obtain these systems, there are some core questions that need to be answered:

1.3.1 Research Questions

- Which machine learning methods are best suited for progressive intelligence models?
- How does progressive intelligence affect the machine learning framework? For example, is accuracy degraded in data-based systems? Or are there improvements in intermediate representations of model-based systems?
- Can progressive intelligence models be made pareto-optimal solutions to machine learning tasks?
- Which machine learning systems are best suited for computationally efficient progressive intelligence?
- Can new inference policies be developed for progressive intelligence?
- How can inference policies maximise performance through information sharing, namely during incremental inference?
- How do progressive intelligence systems deal with non-target inputs, for example out-of-distribution (OOD) and adversarial data?
- Can additional information be incorporated to inference policies to maximise performance on a given task? This might include label information or intermediate outputs from the model.
- Are there additional modes of incremental inference available, to increase the flexibility of such systems?

Some of the answers to these question can be obtained through the collation of existing literature, and some will use additional research. The main goals of this thesis are to:

1.3.2 Research Goals

- Identify good machine learning platforms for progressive intelligence in both the model and data space.

- Find ways to represent pareto-optimality in these systems and make their platforms more efficient where possible.
- Optimise and implement new inference strategies in machine learning which provide progressive intelligence.
- Introduce additional information to inference strategies leading to new modes of incremental inference.

This thesis investigate these goals and its contributions are detailed below. The next chapter collates existing literature to identify a machine learning platform for progressive intelligence, and the subsequent chapters investigate them more deeply.

1.4 Contributions

The main contributions of this PhD are listed below, starting with chapter 3 which considers progressive intelligence and the dichotomy of data-space and the model-space systems in the machine learning field.

Theoretical Understanding of Data-Based Progressive Intelligence We derive some theoretical expectations for the error rate of a data-space progressive intelligence system as the number of training samples increases. We test this expectation with a 1-dimensional gaussian classification problem, finding that it accurately reflects the behaviour of the number of samples against error in our measurement of the classification threshold.

Prototype Data-Based and Model-Based Progressive Intelligence Systems We produce prototype data-space and model-space progressive intelligence systems. An adaptive KNN model is used for the former, and a CNN-based ensemble the latter. We introduce novel inference algorithms for both and test them in suitable benchmarks.

Branched Neural Networks as a Progressive Intelligence We also study branched neural networks as a progressive intelligence system. We find on the CIFAR10 benchmarks they outperform the prototype model-space system.

Chapter 4 follows on by considering branched neural networks in more detail and how they can be adapted to be more effective progressive intelligence systems. It makes the contributions listed below and was published in a conference paper (Dymond et al., 2022).

Representational Effect of Branches in Optimisation Early exit branches included in the optimization of the branched neural network backbone are shown to improve linear probe accuracy and class separation. These benefits are enhanced with greater branch loss weighting but decrease final layer performance. Centered kernel alignment is used to show that branches push important representations backwards.

Architectural Changes for Efficient Inference We vary backbone scaling and design to understand how they effect the power range branched networks operate over, allowing us to understand how to best use model design to influence the performance of a branched neural network.

Mutual Agreement as an Inference Policy A new method of performing early exiting inference is presented. It uses a simple principle during inference: if the current early exit agrees with the previous one, exit. This allowed the inference process to be made much cheaper computationally. When combined with an entropic condition, this was able to improve upon existing training-free approaches in the field.

Finally we consider the contributions made in chapter 5, which considers using additional information in the inference process of branched neural networks. Its contributions are listed below. Some of the work was also published in a conference paper (Dymond et al., 2023).

Early Exit Rejection in Branched Networks We introduce the novel concept of early exit rejection in branched neural networks. This identifies opportunities to halt inference when it can be determined it is not going to produce a meaningful output.

Class-Based Rejection Exit Policy In scenarios where certain classes in the input space have precedence over others, we introduce an early exit mechanism which *rules out* target classes. This allows class-based early exit rejection, producing savings in power without impacting target class performance.

Nested-Set Inference A combination of OOD detection and Class-Based rejection allows for further power savings in environments where there are target classes varying at runtime and exposure to unseen classes. In all cases this improves upon conventional early exiting methods on the target class distribution.

The remainder of the thesis is structured as follows. First a literature review is undertaken in chapter 2, starting with foundational methods in machine learning, then shifting towards neural networks, and finishing with more recent attempts to reduce the cost of neural networks. Chapter 3 considers the dichotomy of data-based and model-based approaches in the field of machine learning, and develops our understanding of progressive intelligence in this domain. Then in Chapter 4 we consider branched neural networks specifically, from their training to their inference policies we optimise them for progressive-intelligence. Finally, in chapter 5 we focus on utilising additional information in the inference policies of branched neural networks, namely embedding information for OOD detection and label hierarchy information. This allows us to open novel avenues for early exiting. Chapter 6 makes conclusions from the work, highlights its contributions, and proposes directions future follow-on work could take.

Chapter 2

Progressive Intelligence

To understand how the need for Progressive Intelligence has arisen we can consider a number of areas that are relevant to the field. All of which are well established, when viewed in isolation however, they do not completely meet the requirements of progressive intelligence. Only when they are adapted or used together can they be considered a progressive intelligence.

First, in section 2.1, we consider different types of machine learning algorithms that have been developed over recent years. As discussed in chapter 1 there exists a dichotomy in the machine learning field, of model-based systems and data-based systems. Model-based systems have learnable parameters which are learnt using the data and produce compressed, or expanded, representations of it. In the case of data-based systems the model will operate within the representational space of data itself. Naturally, these incur different computational requirements for inference, which are discussed in chapter 3.

In section 2.2 we focus more closely on neural networks, due to their prevalence in the most recent advances, in particular computer vision; it is also where we begin to understand the rising cost of progress in the field.

Finally, in section 2.3 we focus on recent advances in compressing these networks to reduce this cost, both efforts to reduce model size and their inference cost. We then also view these algorithms through the eye of progressive intelligence and investigate dynamic inference techniques as a means of cost saving.

To conclude the literature review, section 2.4 considers the gaps in the literature and how this thesis aims to explore them. A section containing relevant mathematical theory is also provided in the appendix A.

2.1 Background: The Advent of Machine Learning

Over the last decade the field of machine learning has seen periods of rapid development and many of these techniques are applicable to the field of progressive intelligence. However, the inception of machine learning came much earlier, when Turing theorised a *learning machine* (recently published in Turing (2009), though originally proposed in the 1950s). Shortly after, algorithms were proposed which are still used today. Hence, we start at these historical developments when considering machine learning algorithms for use in this work.

The first example of a learning machine is the perceptron algorithm, developed by Rosenblatt (1958). The perceptron algorithm is a model-based system which takes a linear combination of inputs to produce a single output; weights for each input are learnt according to the cost function. The perceptron remains relevant in tackling basic machine learning challenges, for instance, Naumowicz et al. (2022) leveraged a multi-layered perceptron. Albeit necessitating a nonlinear activation function, they integrated this into an implantable chip designed to identify vesicle fusion within multi-cellular organisms.

Following the development of the perceptron algorithm, the k-nearest neighbour (KNN) algorithm was proposed by Cover and Hart (1967). The KNN algorithm is a data-based approach and thus parameter free. It uses relative distances between training inputs and test inputs to assign classification at test time. It is readily applicable to multi-class problems where there are C classes for a test input. The algorithm queries the k-nearest training samples, and assigns the test input as the mode of this distribution. It is an algorithm which performs well on a range of classification problems and is still used today, for example in work by Chen et al. (2021a), where they use the k-nearest neighbour algorithm as computationally efficient classification head on a feature extractor. Hence, it is a useful classification algorithm which, in contrast to the perceptron, scales linearly according to the number of samples, N. This could be of use in a data-based progressive intelligence, given the intuitive scaling of N.

Support Vector Machines (SVMs) were proposed by Vapnik (1963) as a classifier which would produce optimum *margins* in the class boundaries, helping generalisation. The principle behind them is that in a linearly separable dataset there exist an infinite number of hyperplanes which separate the data, however the optimum hyperplane is one which maximally separates these points. SVMs achieve this by defining support-vectors, which are the training points closest to the separating hyperplane, and maximising their distance to the hyperplane, also known as the margin. Non-linear SVMs were enabled through the development of a *kernel trick*, developed by Boser et al. (1992), which adds an inductive bias to the algorithm in the form of an appropriate non-linear transformation on the boundary. They are still widely used in the field, including recent work which produces an early example of dynamic inference (Venkataramani et al., 2015).

The above methodologies are all *convex* optimisation problems, that is, in the error functions there is a well defined, single minima. Though, in the case of the perceptron this is dependant on the optimisation algorithm used, and for non-separable data the SVM requires additional constraints on the generated boundary. Nevertheless, this means there are guarantees in your solution that it is the best possible for your method. However, as problems spaces increase in complexity, using high dimensionality data with noisy class boundaries, the existence of a satisfactory solution which is also convexly optimised becomes less attainable. This can be addressed by increasing the parameter space of the model, thus improving its representational capacity but leading to a more complex optimisation problem, towards one that is non-convex.

Multi-layer perceptrons (MLP) are an example of such a system. Rather than using a single layer to map inputs to outputs the multi-layer perceptron, first proposed by Minsky and Papert (1969), stacks these on top of one another applying a nonlinear activation function to the intermediate outputs. The non-output/input layers are referred to as *hidden* layers. This leads to a much richer and more complex parameter space and a model which can be proven to be a universal function approximator (Hornik et al., 1989; Cybenko, 1989). Since the hidden layers of the model do not lead directly to the output, gradients need to be collected for the hidden layers with respect to the target through different means. This process is called backpropagation and is explained in more detail in section A of the appendix.

They remain the classifier of choice in the machine learning field, whether that be using features extracted by a different model (eg. He et al. (2016)), or as a standalone classifier (eg. Naumowicz et al. (2022)). In modern machine learning research they can be collected into the broader class of *neural networks*.

However, with this functional power, there are problems not least of which is the non-convex optimisation problem they present. Such optimisations suffer from a number of issues, from local-minima to multiplicity in the loss space. Furthermore, as these models increase in size, so too does their algorithmic complexity: the compute scales with the number of layers and the *width*, which in this case refers to the number of neurons in the layer.

2.1.1 Recurring Concepts

Before continuing with the literature review it is helpful to clearly define some concepts that will be used throughout the thesis.

Classification and Accuracy

Broadly speaking classification is the process of mapping an input, x, to a distribution of *classes*, C. In the simple case each input will be mapped to a single class, $c \in C$, and will have an associated target class, c_{target}.

Hence it is the goal of the classifier to perform the following mapping:

$$x \longrightarrow c_{\text{target}}$$

In practice the model output is defined as $\hat{y}$, and the target class y. Given a set of N inputs, X, we can define our outputs, $\hat{Y}$, and associated targets, Y. Then we can define our accuracy using the element-wise comparison of the output and target sets. That is:

$$\text{Accuracy} = \frac{1}{N}\sum_{i}^{N} A_i \qquad \text{Where:} \qquad A_i = \begin{cases} 1, & \text{if } \hat{Y}_i = Y_i \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Neural Networks

The perceptron can be defined as the weighted sum of the input x of dimension, D, and the weightings w of the same dimensionality. The bias term, b, is added which is a vector of length D. It can also be expressed in vector notation as shown on the right hand side in the equation below which corresponds to the linear classification boundary.

$$\text{Perceptron:} \qquad \hat{y} = f\Big(\sum_{i}^{D} w_i x_i + b_i\Big) = \mathbf{w}^{\mathsf{T}}\mathbf{x} + \mathbf{b} \tag{2.2}$$

In the MLP, or more commonly refered to as a neural network, the perceptrons are stacked into layers with the signal passing through each sequentially. Where now the input of one perceptron is the output of the one before it. Hence, we can consider a two-layer MLP with layer widths $D^{(1)}$ and $D^{(2)}$ as

$$\text{MLP (two layer):} \qquad \hat{y}_c = \sigma\Big(\sum_{j}^{D^{(2)}} w_{c,j}^{(2)} h\Big(\sum_{i}^{D^{(1)}} w_{j,i}^{(1)} x_i + w_{j,0}^{(1)}\Big) + w_{c,0}^{(2)}\Big), \tag{2.3}$$

where the output vector has c components, corresponding to C classes, h refers to the activation function, and σ the softmax function which acts to normalise the model output and is defined by

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}.$$

The output of the function produces an vector of equal length to its input. By absorbing the bias term into the weights vector we can simplify the notation to $\hat{y}_c = \sigma\Big(\sum_{j}^{D^{(2)}} w_{c,j}^{(2)} h\Big(\sum_{i}^{D^{(1)}} w_{j,i}^{(1)} x_i\Big)\Big)$. Hence, we can now generalise this to N layers

$$\hat{y}_c = \sigma\Big(\sum_{j}^{D^{(N)}} w_{c,j}^{(N)} h\Big(\dots\Big(\sum_{f}^{D^{(n)}} w_{g,f}^{(n)} h\Big(\dots\Big(\sum_{b}^{D^{(2)}} w_{c,b}^{(2)} h\Big(\sum_{a}^{D^{(1)}} w_{b,a}^{(1)} x_a\Big)\Big)\dots\Big)\Big)\dots\Big)\Big), \tag{2.4}$$

where the activation function is constant between all N layers, and the model outputs C outputs, which in a classification scenario will correspond to C classes. It can be optimised using a variety of loss functions, the most common for classification is cross entropy (CE). For a given output vector $\hat{y}$, and target vector y over C classes, these losses are:

$$\text{CE:} \qquad L = -\sum_{c}^{C} y_c \log(\hat{y}_c) \tag{2.5}$$

Where the cross entropy is calculated for a single element in the output vector and will be summed across all outputs. These errors are used to update the model weights by gradient descent through backpropagation, which is discussed in more detail in the appendix A.

Branched Neural Networks

A branched neural network consists of a neural network backbone, such as that mentioned above, with additional classification branches attached to the intermediate layers. These branches allow for classification before the end of the inference process, known as an early-exit. A diagram is shown in figure 2.1.

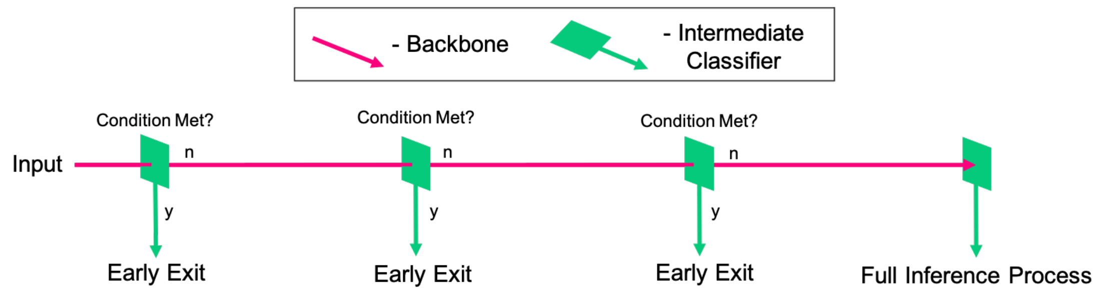


FIGURE 2.1: A diagram of a branched neural network with an early exiting algorithm annotated at the exit branches. Pink components are the backbone, and green components are the early exit enabling branches.

Hence, a branched model has B potential outputs, where B refers to the number of branches. These will each produce a corresponding output $\hat{y}_b$, where the final output $\hat{y}_B$ is that which would be produced by the unbranched neural network backbone. To perform classification with a branched neural network it therefore necessary to introduce a condition as shown in figure 2.1.

Importantly, branched neural networks and progressive intelligence systems in general, can be considered a subset of classification algorithms which require an additional condition at inference time in order to determine their output. These are discussed in more detail later in the literature review (section 2.3.2), and formalised later in thesis (section 4.3.1) where we expand on the current early exiting literature.

Computational Cost

Since progressive intelligence focuses on reducing computational cost of machine learning algorithms, it is helpful to discuss costs associated with them and to define our cost metric going forwards. Some of these such as memory usage, latency, and power are used widely in the field, though, whilst these produce tangible measurements of performance, they will be dependant on the specific implementation.

In order for the work of this thesis to generalise to as many implementations as possible we wish to identify a metric/measurement for computational cost which is independent of specific implementations. There are two popular options used in the field for this:

- The number of parameters a model uses when stored in memory.
- The number of computations a model uses when being used in inference.

Each of these are useful in specific use cases depending on the limitations a particular approach is trying to address. In progressive intelligence we aim to reduce resource usage of the inference process, hence the latter is more appropriate, since parameter counts will be static once a model is being used.

For this we can record the number of Multiply-Accumulate operations (MACs) used by a model. True resource usage will inevitably depend on the specific implementation, and observed performance and power may actually differ slightly from the MACs usage, though the ordering of costs will likely be maintained. That is, an algorithm using more MACs will likely also use more power, though fractional differences in these metrics will be different.

So, whilst this measure may not always reflect raw power usage it will provide us with an architecture agnostic approximation of it, allowing us to compare similar approaches. Hence, this is the measure used throughout the remainder of the thesis when analysing progressive intelligence inference.

We have discussed a small number of machine learning techniques, and there are a multitude of machine learning techniques available, which vary in complexity. However, with large, diverse, and often unstructured datasets becoming more available in the form of language and vision tasks, the need for complex representational ability is becoming more prevalent in contemporary machine learning research. Consequently, the less computationally complex algorithms have less utility on these datasets by themselves, due to the increased variance permitted by the more complex models such as neural networks (Kohavi and Wolpert, 1996; Neal et al., 2019).

As such, we will consider neural networks in the next section, in the context of computer vision. But also we consider the drawbacks of moving towards these algorithmically complex models, namely the practical constraints in storing and running such models.

2.2 The Cost of Success: Neural Networks for Computer Vision

When considering intelligence we need to consider the medium through which it is evaluated. The two most accessible of these are vision and language. Because vision is a key sense in humans, through our years of learning the problems presented to us are often visual in nature. For example, we might be asked to identify a picture of a farm animal, where we will match the image to the name. It follows our thoughts are arranged according to our language, providing us a structure to our intelligence whilst also providing a medium over which to share and assimilate it with others.

Hence, it is no surprise that in our pursuit of artificial intelligence these senses are at the forefront of progress. Since 2010 the advancements in the language and vision space have been at the leading edge of machine learning research, in what has been a period of unprecedented progress driven by neural network research. This is with the exception of a few prominent results, for example AlphaGo Silver et al. (2017), and AlphaFold Jumper et al. (2021), which both use deep neural networks for other tasks. The former to beat world champion Go players and the latter to solve the protein folding problem (Levinthal, 1969). In this section we start with vision, as chronologically this progress came first and is still ever-present in the literature. However, we also discuss that this progress has not come without an ever-increasing computational burden.

In the previous section we considered the MLP, which is commonly referred to as a linear neural network, since it is composed entirely of linear layers with non-linear activation functions. These can be used for a variety of tasks and it can be shown they are universal function approximators (Hornik et al., 1989; Cybenko, 1989). However, in practice they are difficult to optimise for complex problems in the vision space. To overcome this, an inductive bias needed to be developed for vision problems. This came in the form of the convolutional neural network (CNN) (Fukushima and Miyake, 1982).

Inspired by the visual cortex of the brain, the CNN architecture introduces a new type of operation in neural networks that would prove to be incredibly powerful in recognising local relationships in images, the convolution. In essence, a convolution operation in a CNN takes a $D_m \times D_n$ kernel (Often $D_m = D_n$), and applies it over an image in a 'sliding window' manner. For each available window coordinate on the image a single output value is produced. It can be represented diagrammatically as shown in figure 2.2 (Goodfellow et al., 2016).

There also exist pooling layers in convolutional networks, which pass the same kernel over the input features, but instead output the maximum value over a given selection in the case of max-pooling, or the average value in the case of average-pooling. In the example of figure 2.2 each box would be the maximum pixel value in the case of max pooling, or the average pixel value in the case of average pooling, and no weightings would be applied. Convolutional and pooling layers are often implemented in channels, for example the input layer is often a

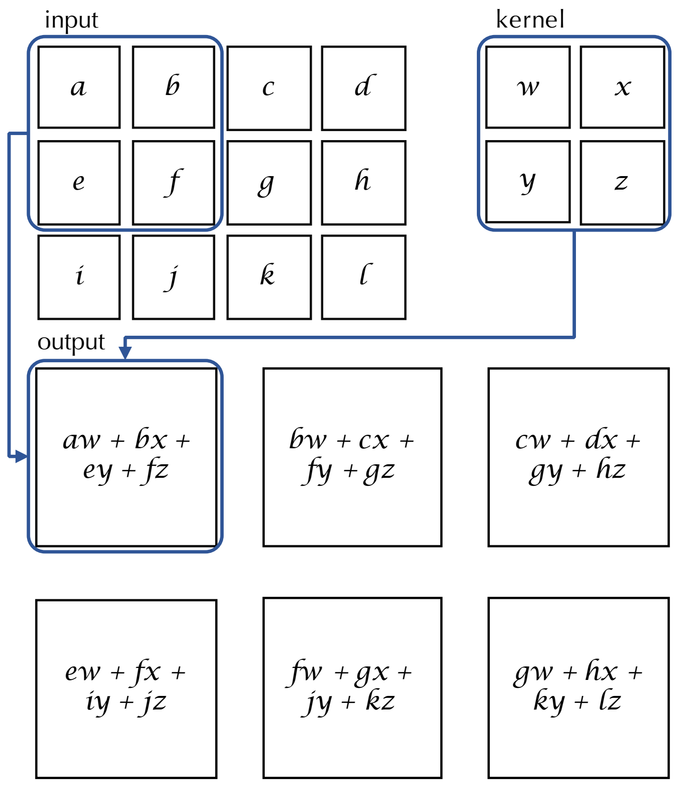


FIGURE 2.2: The convolution operation in CNN architectures inspired by the figure in 'Deep Learning' by Goodfellow et al. (2016), chapter 9, page 334. In this case the output is restricted to positions where the kernel lies entirely within the image. The boxes and arrows indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor. The act of moving the kernel outside of image is known as *padding*, where the edge of the image is supplemented with a border of a fixed value, most often zero. The values in the kernel are often referred to as its weights.

colour image composed of RGB values requiring three channels. As the network gets deeper this number is often expanded.

Whilst the convolution is a powerful tool, compared to linear layers, it can also be an expensive one. There are fewer parameters in a convolution operation, but the number of operations is much higher. The convolution for example has the following computational costs:

$$\text{Operations:} \quad D_m \times D_n \times M \times N \times D_f \times D_g \qquad \text{Parameters:} \quad D_m \times D_n \times N \times M + M$$

D_m, D_n, N, M, D_f, D_g refer to the kernel size, the number of input/output channels, and the output feature size respectively. In contrast, a linear layer takes

$$\text{Operations:} \quad D_m \times D_f + D_f \qquad \text{Parameters:} \quad D_m \times D_f + D_f$$

Where now the sizes D_m, D_f refer to the input and output size of the linear layer. Typically these layers are thousands of units in size, leading to a large number of parameters.

To illustrate this, we can consider a typical convolutional and linear layer in a neural network.

The convolutional layer has a $D_m \times\ D_n = 3 \times 3$ kernel, and operates on $N = 64$ channels, producing $M = 128$ output channels, all of feature size $D_f \times D_g = 16 \times 16$, leading to a cost of roughly 18 million operations, and a parameter cost of roughly 74 thousand. In contrast, a linear layer taking as input $D_m = 1024$ and outputting $D_f = 512$ values leads to a cost of 525 thousand operations and parameters. Hence, whilst the convolution has $\sim 15\%$ of the parameters of a linear layer, it has $\sim 34\times$ the operations cost.

However, when concepts of convolutions and pooling were first introduced by Fukushima and Miyake (1982) their cost was not the main concern, rather it was optimising them to perform the task. One of the first successful implementations of them can be found in a paper by Lecun et al. (1998), where they were used to recognise characters in documents. Following a number of convolutional layers, the output tensors are flattened to a linear representation and the final layers of the network are architecturally equivalent to a multi-layer perceptron described in section 2.1 and defined mathematically in equation A.5, in section A of the appendix. In the same paper, the most prevalent benchmark in the field was proposed, *MNIST*, which is a monochromatic digit recognition dataset and is still used as a simple proof-of-concept benchmark.

At the time of their inception, convolutional networks were very hard to optimise due to their expensive forward method, and large number of parameters. As such, their development was slow. However in the 2010s, more readily available computational power along with the introduction of the most popular benchmark in the field, *ImageNet* (Deng et al., 2009), precipitated the rapid development of CNNs.

ImageNet uses comparatively high resolution images using three channels for RGB pixel values and it is composed of 1000 classes. Hence, the ImageNet challenge, also referred to as the ILSVRC (ImageNet Large Scale Visual Recognition Challenge), presented a single, challenging, and representative benchmark for computer vision models. This meant models trained on ImageNet could be deemed effective image classification models and as such the field has entered a period of pushing performance metrics, namely accuracy, as far as it can.

At the same period of time an intermediate benchmark was developed, *CIFAR10/100*, which comprised of low-resolution RGB images with 10 and 100 classes respectively (Krizhevsky et al., 2009). Therefore, with MNIST, CIFAR10, CIFAR100, and ImageNet, there was a clearly defined experimental pipeline for computer vision research which helped facilitate rapid development. At the leading edge of this remain the models competing on the ImageNet challenge.

Hence, these datasets will be of use in developing a progressive intelligence which operates on vision tasks. However, it is also necessary to identify a starting architecture on which to apply progressive intelligence. As such, the rest of this section explores the recent advancements in neural networks for image classification.

Perhaps the most influential convolutional architecture in this challenge was AlexNet, developed by Krizhevsky et al. (2012). It massively increased upon the previous winner of the competition, improving accuracy by 9.4% on the XRCE (Xerox Research Centre Europe) submission from the year prior (Perronnin et al., 2010). The key concepts introduced by this paper were the ReLU activation function and dropout as a form of regularisation. Dropout refers to the intermittent switching off of random neurons in the output layer of the network at training time, which ensures the representations being learnt by the model are spread across its components more easily. The ReLU (Rectified Linear Unit) activation function can be defined as $\max(0, x)$. The other popular activation functions at the time were the sigmoid and the tanh function

$$\begin{aligned}
\text{ReLU:} \quad f(x) &= \max(0, x) \\
\text{Sigmoid:} \quad f(x) &= \frac{1}{1+e^{-x}} \\
\text{Tanh:} \quad f(x) &= \tanh(x) = \frac{2}{1+e^{-2x}} - 1
\end{aligned}$$

Introducing the ReLU had the effect of de-saturating the activation space, meaning the gradient could take a wider range of values. This is in contrast to previous functions which restricted the activation values between small ranges, approaching zero either side of this range. This can be seen by plotting the activation functions and their derivatives:

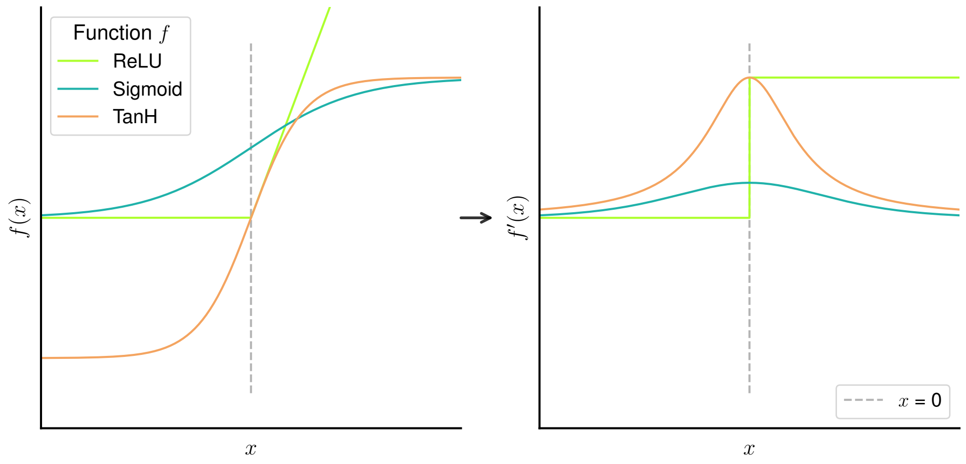


FIGURE 2.3: The common activation functions in the field, on the left-hand side, as well as their derivatives on the right. The $x = 0$ line is denoted with a dotted grey line.

To overcome the nil gradient values at $x < 0$, adaptations have been proposed such as the *leaky ReLU* (Xu et al., 2015) and the *swish* (Ramachandran et al., 2017). However, the gains are marginal and the ReLU activation function remains the most popular in the field.

The field found no issues making networks wider, however struggled to make architectures deeper. Deeper architectures would allow the model to represent more complex functions more easily and make better use of their parameters (Kidger and Lyons, 2020). As subsequent layers can make use of previous transformations to learn high level features in the data, shallower networks would have to learn such features from scratch.

One issue increasing the depth gives rise to is the downsampling of the input at each layer. Reducing the amount of downsampling will enable more information about the input to be retained, thereby providing deeper layers more granular information about the input. A key component to this problem is the initial layer of the network. For example, AlexNet uses an 11×11 kernel, massively downsampling the input in the early stages of inference thus losing information. This keeps the number of operations down, but relies on large linear layers at the end of the network leading to a large parameter cost reflected in figures 2.4 and 2.5 respectively.

In contrast, models like VGGNet introduced by Simonyan and Zisserman (2014) used a 3×3 kernel and the authors also introduced 1×1 kernels which allow the model to introduce more non-linearities, without decreasing the receptive field of the network. The receptive field of the network refers to the size of the region in the original image a given kernel is processing. As you go through the network this receptive field naturally increases as the representation of the image is compressed by the kernels. These additions allowed the model to be almost twice as deep as previous convolutional neural network architectures.

Making these networks much deeper also introduced problems with *vanishing gradients*. This is when the gradient of the model approaches zero and the initial layers of the model receive no gradient signal from the loss function. One paper solved this problem by introducing branches attached to intermediate layers during training (Szegedy et al., 2014). These provided the layers before them with a stronger gradient signal, allowing the model to be optimised. Codenamed 'Inception', now referred to as InceptionV1, this model would win the ImageNet challenge that year and present a step forward in demonstrating the power of deeper convolutional networks.

Another more recent way of solving this problem was to introduce skipped connections. This is where parts of the neural network pass the input of a given layer to a later layer in the network, without letting it be processed by the layers inbetween. This has the effect of passing the input through the identity function in parallel with standard convolutional layers. These were introduced by He et al. (2016) and were called residual connections in an architecture they called a *ResNet*. They allowed the gradient to flow directly through the skipped connections to the intial layers of the model, consequently allowing the networks to be significantly deeper. These networks could get very high accuracy on the ImageNet challenge, and still serve as the foundation for many state-of-the-art advancements in the literature. Furthermore, they have recently been reintroduced by Bello et al. (2021), who propose minor architectural changes with updated optimisation schemes and show the architecture can still achieve state-of-the-art results. Recent work by Liu et al. (2022) also adapts the ResNet architecture to compete with SoTA approaches. As such, this architecture will be of use when developing a progressive intelligence system for vision tasks.

These developments allowed neural networks to increase in size massively, both in width and in depth. However the exact depth and width of the networks was often designed based on heuristics and empirical evidence. Tan and Le (2019) introduced an optimal scaling strategy for these networks, based on a small scale model. They define a small model which is trained on a small dataset, and learn an optimised scaling method for the network. They do this by defining a parameter, γ, which through a number of relationships can be used to define the scaling ratios of the depth and width of a network, when given the resolution scaling ratio. Here the width refers to the number of channels a given layer of the network has and the depth is the number of layers. The architecture itself incorporated *blocks* of layers in its design inspired by work by Howard et al. (2017), which will be discussed in section 2.3.

The optimised depth–width configurations of these networks were able to outperform all competing architectures of similar size, as shown in figure 2.5, but figure 2.4 shows there is still a large increase in operations. However, their success has meant they are still used as a starting point in recent works, which look to optimise these networks through different training regimes (Tan and Le, 2021; Jia et al., 2021; Pham et al., 2021; Xie et al., 2020).

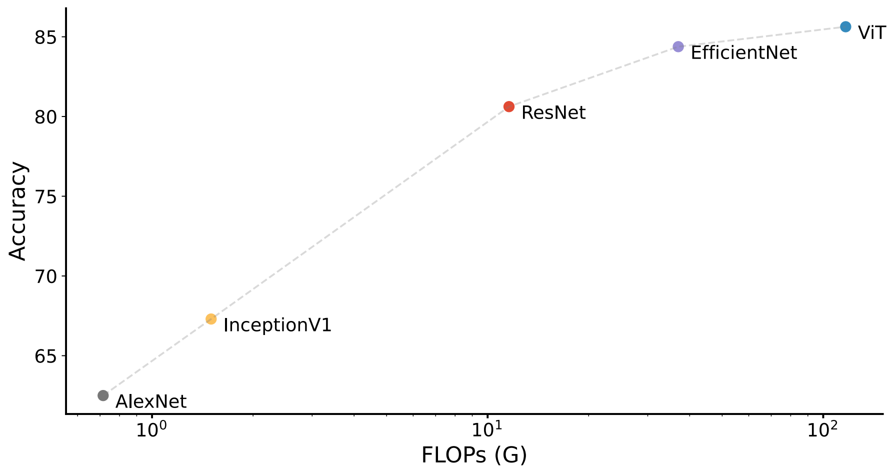


FIGURE 2.4: ImageNet accuracy against computational cost, here reported in GigaFLOPs. A dashed grey line denotes the chronological connection between the ILSVRC winning architectures discussed in this section. The performance of deep learning models has increased dramatically. However, so too has the computational cost of running them, let alone the cost of training and storing them.

Figure 2.4 demonstrates a key issue with convolutional networks for progressive intelligence, in that they operate at discrete points. Whilst the number of parameters is likely to remain static at inference time, the operations will vary dynamically in a progressive intelligence system. However, neural networks, in particular CNNs, are an appropriate starting point for progressive intelligence in vision problems. ResNet is the most popular architecture in the

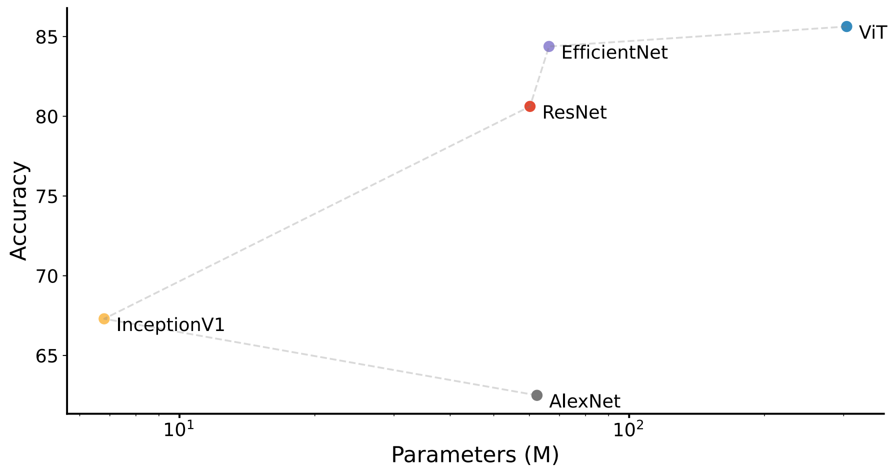


FIGURE 2.5: ImageNet accuracy against computational cost, now reported in parameter counts in millions. A dashed grey line denotes the chronological connection between the ILSVRC winning architectures discussed in this section. There is a less clear relationship between top performance and parameter counts, but generally there has been monotonicity since the advent of deep neural networks.

field and is accessible in terms of computational cost with smaller variants also performing well. But, before considering the more resource aware literature, we can briefly discuss the state-of-the-art in vision and why it is not suitable for progressive intelligence at this time.

Whilst the convolutional kernel has been massively influential in the field of computer vision, the inductive bias that made it perform so well can also be somewhat restrictive as the networks become larger. That is, the convolutional kernel can only process local information, between adjacent pixels. It cannot process and form relationships with global information until the very end of the network, at which point lots of information may be lost through the compression of the input. Hence, there is an opportunity to improve upon these architectures and that is the goal of the vision transformer.

First demonstrated by Dosovitskiy et al. (2020), the vision transformer (ViT in figures 2.4 and 2.5) makes use of the transformer model, originally designed for natural language processing (Vaswani et al., 2017). The transformer architecture introduces the concept of *self-attention.* At a high level, attention allows the model to encode global information across an input sequence, which at the time was a sentence. For example in the sentence “The animal didn’t cross the street because it was too tired", ‘it’ could refer to the animal or the street. The self-attention mechanism allows the model to analyse the sentence as a whole allowing it to associate ‘it’ with ‘animal’.

This incorporation of global information could also improve vision models, allowing global and local relationships alike to be encoded. To enable this, they split the image into patches and provide flattened, lower dimensional representations of these patches as an input to a transformer, along with their positional embeddings.

A self-attention module takes n inputs and returns n outputs, rather than compressing the representation like convolutions. The mechanism works by allowing the inputs, known as *tokens*, to interact with one another and calculates an *attention* score. As before, this score represents how 'related' two tokens are to one another and can be interpreted as the amount of attention the model should be giving to the tokens with respect to one another. The output is an $n \times n$ matrix showing the relationship between each token. These are flattened and passed to subsequent layers of the network, where the process repeats. Typically there are multiple attention heads in a given layer, which can find different relationships between the tokens.

These networks are very difficult to optimise, due to their less restrictive inductive bias. However, with enough data they have shown to be very effective on all of the computer vision benchmarks and represent the state of the art in this space (Dosovitskiy et al., 2020; Wang et al., 2021; Liu et al., 2021a). Furthermore, work by d'Ascoli et al. (2021) showed that by inducing a *soft* convolutional inductive bias, the convergence rate and performance of the transformer architecture can be improved.

Transformers are at the forefront of progress in the vision space, with image classification (eg.(Chen et al., 2023)), object detection (eg.(Zong et al., 2023)), and semantic segmentation (eg.(Wang et al., 2023)) benchmarks being dominated by transformer-based architectures. However, their size (demonstrated in 2.5) and the difficulty of optimisation make them more appropriate for future research directions in progressive intelligence, which we discuss in section 6.2.1. Thus, initial work will be better placed in convolutional neural network implementations.

Transformers were originally developed for language tasks and also prompted unprecedented improvements in this space. This has been spearheaded by OpenAI and their developments of the Generative Pretrained Transformer architecture, GPT-1/2/3/4 (Radford et al., 2018, 2019; Brown et al., 2020; OpenAI, 2023). The work pre-trains successively larger transformer models on self-supervised generative tasks, such as filling in missing words. They find that the resultant model is very powerful on common language understanding metrics. The most recent architecture being implemented in a state-of-the-art *chatbot*, ChatGPT. The performance ChatGPT is capable of achieving along with its ease of access has massively increased the public awareness of AI.

This rapid acceleration in progress and its exposure to the public eye has prompted much discussion around AI governance going forwards. The article by Baum et al. (2023) suggests this is down to the public perception being one of fear and arguably this fear is derived from our lack of understanding of neural networks. They are capable of surpassing human performance in a number of tasks but we often do not fully understand how they achieve this. Methods of understanding neural networks will be of use when trying to understand progressive intelligence, as such we briefly discuss some related topics in section 4.1.1.

So we have shown that deep neural networks are very powerful in the field of computer vision, and continue to be the most successful algorithm in the field. However, we have also seen that this performance requires increasingly large and complex architectures, this leads to the deployment of these algorithms to edge devices becoming intractable. For this to occur, they need to be compressed and the recent literature has begun to address this.

2.3 Looking Beyond Accuracy: Efficiency in Neural Networks

In figures 2.4 and 2.5 we see comparisons in accuracy vs cost of recent state-of-the-art architectures discussed in the previous section. To put these into perspective the ViT, which is a comparatively low cost transformer-based architecture, uses 0.3 Billion parameters which are stored using 32 bits. This leads to an overall storage cost of 1.13 GB at runtime, when running inference and backpropagation this number increases multiplicatively as intermediate activations need to calculated and stored in memory. This makes training infeasible on consumer devices and inference very slow. Hence, there is a need to cut down the cost of using deep learning systems.

Broadly, there are two ways to make neural networks more efficient, the architecture and its components can be made more resource aware which we refer to as *compression*. This approach can be considered static, since once the model is compressed it has a single operating mode. We briefly cover these methods first in section 2.3.1. However, due to their fixed nature, they still do not allow us to progress the amount of computation incrementally like in figure 1.1.

On the other hand, we can approach the problem dynamically and treat the inference process as such. This is more aligned with the ideas of progressive intelligence, and we discuss this in more detail in section 2.3.2. First, we consider the static approaches to compression.

2.3.1 Compressing Neural Networks

Perhaps the most obvious way to decrease the size and resource requirements of a machine learning model is to remove the components that have little to no effect on the outcome of the model. This paradigm is known as *pruning* and it has been a common technique in the area of model compression for some time, one of the first examples of which was referred to as *Skelotonization* by Mozer and Smolensky (1989). The reason for removing the connections is so that the weights for them no longer need to be stored, hence saving on computation requirements. Their criteria for doing so is that if the predictive performance of the network is unaffected by the removal of a connection, the connection can be removed. They estimate this using an additional parameter calculated in backpropagation which measures the importance of a neuron.

Pruning methods can be organised according to their structure, their criteria, and the scheduling method of the pruning.

The structure refers to whether they are structured or unstructured. Unstructured methods prune connections in a layer without taking into account their structural location in the network. This provides a less restrictive pruning scheme which can remove more connections, however hardware architectures often cannot readily exploit the sparsity of such networks. Therefore, performance gains in terms of inference cost are limited (Han et al., 2015b; Chao et al., 2020; Malach et al., 2020). The other option is to perform structured pruning, which removes entire components of the network, both reducing model size and inference cost (Li et al., 2016; He et al., 2017; Meng et al., 2020a).

When developing a pruning method, it is also important to consider the criteria for removing a component. In the paper by Mozer and Smolensky (1989), they consider the performance of the network when a parameter is removed. In the field this performance-based approach is now referred to as *greedy* pruning, in practice this is infeasible and expensive since it requires a forward pass of the model to estimate the drop in performance Li et al. (2016). The method still sees use in methods that consider parameters more holistically, like in recent work by Ye et al. (2020a). However, more popular methods estimate the importance of a given parameter. The most basic form of this, but also one of the most effective, is known as *magnitude* pruning.

Magnitude pruning uses the absolute value of a weight to determine its importance in the prediction of the model. The argument being that nodes with weights close to zero will have little effect on their inputs and thus little effect on the predictive performance of the network, these values are often normalised using the weight distributions on a given layer. Han et al. (2015b) first introduced this methodology and showed the model could take up substantially less memory without loss in predictive performance.

Whilst magnitude pruning is a relatively rudimentary form of pruning, it still sees use in the literature and remains to be a competitive criterion for pruning (Lee et al., 2021). However, finding new criteria is still an active area of interest in the community. Sanh et al. (2020) propose *movement* pruning, which uses the gradient signal during fine-tuning to determine which weights are moving away from, and towards, zero. Similarly, Ye et al. (2020b) use this principle to accelerate the training of CNNs, explaining that weights which have a small gradient signal, have little effect on training and can be removed from the procedure. Park et al. (2020) introduce an algorithm which considers the multi-layer interactions a weight imposes, incorporating the Frobenius norm to calculate the representational effect on adjacent layers when removing a connection.

Another way in which pruning methods can vary is the schedule over which the pruning takes place. The most common procedure is to first train a model, then prune it, then fine-tune the pruned model, and repeat the last two steps iteratively until the model is sufficiently compressed (Han et al., 2015b). However, other works look to prune in a single step, known as one-shot pruning and side-step the initial training altogether by pruning at initialisation (Lee et al., 2019; Wang et al., 2020; Zhang and Stadie, 2020; Hayou et al., 2021). Work by Tanaka et al. (2020) even challenges the conception that data is needed to quantify which synapses are important and instead aims to preserve *synaptic flow* in a data agnostic way.

Hence, pruning remains an active and successful area for making neural networks more efficient and even in helping us understand the way neural networks learn (Frankle and Carbin, 2018). Recent work by Blalock et al. (2020) compares all recent pruning methods, they collate results and highlight some of the issues in the neural network pruning community.

Another way in which the computational footprint of neural networks can be reduced is to decrease the precision of their parameters, by varying their bit width. This field is known as quantisation.

Quantisation has taken a number of forms in recent literature, some binarise the neural networks by using just two weight values and some work uses ternary weights, for example: -1, 0, 1. However, most work uses varying numbers of weights in their quantisation.

First proposed by Hubara et al. (2016) and Kim and Smaragdis (2016), binary neural networks naturally allow for the greatest amount of compression, with parameters theoretically being able to be represented by single bits. These values can be 1 and 0, or -1 and 1. Recent work by Qin et al. (2021b) has fully binarised a popular language model, BERT. Other work by Xu et al. (2021) has improved the performance of Binarised Neural Networks by 'reviving' weights which see very few updates during training, this moves them closer to the weight distribution centroid, lessening the effect of binarisation.

Other work increases the number of available weight values to three, in so called ternary weights. Presented in work by Zhu et al. (2016), they multiply their three values, -1 ,0, and 1, by a learned parameter at each layer of the network. This allowed the network to be quantised, whilst still maintaining the optimum weight distribution at each layer of the network.

Recently, work focuses on quantisation with slightly less restricted weight counts, due to its increased performance. Guo et al. (2022) propose a data-free approach, which approximates the Hessian matrix to essentially round the weights, but do this by decomposing it progressively into element-wise, then kernel-wise, and then output channel-wise. Wei et al. (2022) also present a new post training method which works by randomly dropping the quantisation of activations during weight reconstruction, which they show theoretically should benefit final accuracy.

Quantisation remains an active area of the field, with many papers improving the technique in recent conferences, both in the binary methods (Diffenderfer et al., 2021; Qin et al., 2021a; Bird et al., 2021; Bulat et al., 2021; Meng et al., 2020b), and otherwise (Liao et al., 2021; Wang and Scott, 2022; Liu et al., 2021b; Yamamoto, 2021; Liu et al., 2020). These methods have also been combined in some literature (Han et al., 2015a).

In progressive intelligence, pruning and quantisation techniques could be of use in developing switchable modes, indeed we discuss similar ideas in section 2.3.2. Though, whilst pruning and quantisation remain the most active areas of research for compressing neural networks, there have been other methods proposed.

Kim et al. (2019) propose a holistic approach to the compression process rather than a layer-wise one. They consider the whole network when choosing a rank configuration for kernel decomposition, they use accuracy and complexity constraints to determine which one should be used. Reagan et al. (2018) introduce a method which encodes a model using a *Bloomier* filter (Donnet et al., 2006), which can compress a data structure massively at the expense of small random errors. This allows the model to be approximated in a probabilistic way and by training the model to be resilient to this accuracy can be preserved. Another approach used, has been to avoid the explicit compression of a model, and attempt to do it implicitly by getting a model to learn the representations of a pretrained, larger model. This is known as knowledge distillation and was introduced by Hinton et al. (2015). It works by creating a co-optimisation problem on the target, or 'student' network. This encourages it to recreate the softmax output of the larger, 'teacher' network, whilst also encouraging it perform the classification task correctly. Knowledge distillation remains an active area of research, with many papers building upon it (Stanton et al., 2021; Heo et al., 2019; Park et al., 2019; Tian et al., 2019).

Despite these advances, explicitly designing bespoke architectures for deployment on edge devices is perhaps the most effective way to create compressed models. That is, the algorithm is designed from the ground up, with efficient use of computational resource intended from the outset. There are a number of papers that show this, arguably the most prevalent of which is in work by Howard et al. (2019), who introduce *MobileNetV3*.

MobileNetV3 is the third iteration of the authors' original model, MobileNet, a model that was designed in an earlier paper (Howard et al., 2017). In the original MobileNet paper, the authors propose an efficient convolutional architecture that uses depth-wise separable convolutions.

Separable convolutions are convolutional operations that are split into two parts: a kernel is factorised into two vectors, which are both applied to the image. This reduces the computational complexity of the operation while the effect of applying them to the input is the same as the original convolution. Depth-wise separable convolutions apply this idea to the channel dimension of a kernel, separating the convolutional operation into a number of single-channel operations, and then a 1×1 point-wise operation on the intermediate result. This has the effect of reducing the computational cost of the operation. Using 3×3 depth-wise separable kernels the authors can reduce the computational load by 8-9 times, we discuss this in more detail in section 4.2.1.2.

In MobileNetV2, they extend the idea but add inverted residuals and linear bottlenecks. Inverted residuals are different from traditional residual connections, in that they follow a narrow-wide-narrow approach in channel depths, as opposed to wide-narrow-wide. This has the effect of reducing the number of parameters used up by connecting wide filters. In MobileNetV3 they incorporate neural architecture search (Zoph and Le, 2016; Liu et al., 2018) to redesign the most expensive layers of the network, such as the final stage of the network. This is another architecture, which like ResNet, has been prominent in later literature (eg. Hu et al. (2020)). Hence, this particular architecture will be useful for exploring progressive intelligence.

Similarly, work by Zhang et al. (2018) leverages the performance benefits of grouped convolutions[1] whilst keeping the parameter cost of widening the network low. They achieve this by *shuffling* the filters and see similar behaviours to grouped convolutions, without incurring the computational overhead that go with them. The only network that still sees active use is the MobileNetV3, which is regularly used as a benchmark architecture.

Hence, the field of compressing neural networks is a varied space, whilst it is dominated by the parameter-centred methodologies such as pruning and quantisation, other methods and architectures have also been developed and have been shown to be successful. Compressing models at runtime can also lead to massive cost-savings in the corporate environment. When services like ChatGPT are costing an estimated $700,000 to run a day, even more moderate savings will decrease the money spent on running deep learning models considerably.

However, the most applicable approach to progressive intelligence would be one which focuses on the inference process itself. Since progressive intelligence is an approach to cost savings which focuses on incremental changes at inference time. In neural network research this field is known as *Dynamic Inference*, where the work often dynamically scales inference costs. We consider this in the next section.

2.3.2 Dynamic Neural Networks

The majority of the literature review so far has provided background information for the thesis, with some approaches being used later, such as KNN models in chapters 3 and 5 and the MobileNet and ResNet architectures being used throughout. However, dynamic neural network approaches are the most pertinent area for progressive intelligence and are most informative for the thesis.

This subset of the field can be split into two areas: those which can adapt to various operating modes, dynamic architectures, and those which extend this to the inference process in dynamic inference. Dynamic inference is the most relevant for the progressive intelligence

[1]Grouped convolutions are when multiple pathways exist for convolutions to take on a single image much like AlexNet, which has two separate paths for inference to take and its results are coupled together nearer to the classification layers. Empirical evidence suggests this allows the network to form better representations.

problem as it opens the possibility for incremental inference. However, some recent works in the dynamic architectures space will briefly be discussed first.

Works designing new architectures usually present a number of different sized architectures (Howard et al., 2017; He et al., 2016; Tan and Le, 2019; Dosovitskiy et al., 2020) which will be applicable to many different scenarios. However, all of these need to be trained separately. *Slimmable* neural networks introduced by Yu et al. (2018) was among the first to present single networks that could be used at various different operating modes. These are networks which have switchable widths, that is, their channels numbers can be varied dynamically, without having to reload those corresponding to the smaller widths. They train the network to perform with good accuracy at all widths, whilst keeping a subset of the weights consistent between the models. Yu and Huang (2019a) extend this work to be architecture agnostic, by improving some of the training techniques and show that this technique can be applied to many different architectures.

Cai et al. (2019) propose a method which trains a single architecture, and through weight sharing opens the possibility to make the slimming occur at a finer grained, weight level. This allows the network to use less memory, but performance gains at run-time are lessened. Their approach is focused more on creating single architectures after one training sequence, as opposed to adaptivity at run time.

In recent work Yang et al. (2021b) propose a method similar to *slimmable* nets, and extend the method so that it can function on a variety of input resolutions. Their method also does not require retraining. They argue that by training the model to function at a variety of different resolutions depending on resource requirements, the model learns stronger representations overall.

Whilst dynamic architectures will fulfill the requirement of multiple operating modes in progressive intelligence, if this dynamism is extended to the inference process itself, additional avenues of progressiveness become available through incremental inference and input dependence. This is the field of dynamic inference. Here, the techniques look to save on resource usage by adapting inference cost depending on input or application, in contrast to dynamic architectures which are not implemented in an input-dependent manner. In the field data-sets will often have differing levels of difficulty across the inputs which will necessitate differing levels of representational capacity on an input-wise basis.

In the machine learning domain, there are a number of different algorithms that differ in their representational capacity, for example, there are linear models which produce simple classification boundaries, and neural networks, as discussed in the previous section, which will produce more complex boundaries depending on the complexity of the data.

Therefore, a simple way to implement dynamic inference would be to use multiple models sequentially, passing to subsequent models depending on output confidence of the previous model. This idea was explored by Venkataramani et al. (2015), where the authors define an algorithm known as a *scalable effort classifier*, which is defined as a hierarchy of gradually more complex models. In the paper, they define a problem in which some inputs are more difficult to classify than others, and hence different models are more suitable than others.

Samples distant from the classification boundary can often be distinguished with a linear classifier, but those closer to the true boundary require the more complex model. Since the majority of samples may be easy to classify, using the simpler model for easier samples could produce significant power savings, as the complex model is only required on a minority of data points. This helped form the basis of their paper.

To explore this setting they identify several benchmark problems from the field, and use a variety of machine learning methodologies. For example, they implement SVMs on MNIST, and neural networks on a protein classification dataset. They construct their classifiers hierarchically, and depending on the confidence of the output from the simple models, they either output the classification or pass the input to the more complex model.

They parameterise this score using the Consensus Threshold, δ. It is called this, as the 'simple model' actually consists of two simple, but biased models. That is, each of these biased models will specialise on one class, misclassifying the more difficult samples of the other class, but seldom that of the class they specialise on. Hence, when both biased models agree with one another, the 'consensus' score between them is strong. They also use the absolute values of the confidence scores to measure the confidence in the output of the classifier, and compare it against their consensus threshold. Modulating this threshold would make the model more stringent when passing to successive models. They would range this value δ between -1 and 1. To evaluate energy consumption, they measure run-time and utilise an out-of-the-box algorithm that estimates power usage. When optimising the number of classifier stages, they found that three stages would often be optimal. This provided high accuracy, and an overall low number of operations per second.

Before this paper, most instances of this methodology were application-specific, hence it was one of the earlier examples of these ideas being generalised to the benchmark problems of the field. Whilst achieving interesting results, this paper's methodology uses many models, and many forward passes for a single inference, we call this approach the *bag-of-models* approach. Whilst they show this decreases power consumption, the use of so many models is counter-intuitive when saving on computational overheads. This approach was not shown to work on more complex tasks where power consumption is a more pressing issue. We build on this approach somewhat in section 3.3.1, creating a similar system which reuses model outputs. Though, to save on memory costs the field should move towards approaches that use much fewer models, or even a single model.

A simple and neat way to incorporate progressiveness into single models is to introduce classification branches throughout a neural network. This allows early exiting, meaning the inference can take place incrementally. Branches in neural networks were first proposed to assist with the training by Szegedy et al. (2015), where auxiliary classifiers were used to combat the vanishing gradient problem found in deep neural networks and improve convergence. In this work the auxiliary classifiers were removed at test time.

Later work however, kept these branches in at test time to reduce inference cost (Teerapittayanon et al., 2016). They also showed the branches provided regularisation and reinforced their ability to mitigate the vanishing gradient problem. These branches were trained in parallel using a weighted loss, L, shown in equation 2.6. The loss for each branch b in B total branches is given by L_b, shown in equation 2.7. Which, in this case is the cross entropy over C classes.

$$L(y, \hat{y}) = \sum_{b=1}^{B} w_b \, L_{\text{b}}(y_b, \hat{y}_b) \tag{2.6}$$

$$L_b(y_b, \hat{y}) = -\sum_{i=1}^{C} y_{b,i} \log(\hat{y}_i) \tag{2.7}$$

The weighting for branch is defined as w_b, y and $\hat{y}$ refer to the target and output vectors respectively. An entropy-based exit policy was introduced to determine when an early exit should take place at inference time. This takes the Shannon entropy (see equation 2.10) of the normalised categorical distribution at output, $\hat{y}$, which has often been normalised using the softmax equation:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}. \tag{2.8}$$

We explore inference policies for branched neural networks in more detail below. When dropout was popularised in CNNs the problem of overfitting was diminished (Srivastava et al., 2014), likewise the introduction of ReLU activation functions and residual connections into DNN architectures made the vanishing gradient problem easier to manage (Nair and Hinton, 2010). However, the field of dynamic inference took branched networks further, one paper using dense connectivity between the layers in combination with early exits allowing coarse and fine features to pervade all layers of the network (Huang et al., 2017). Recent work has implemented early exits in conjunction with layer skipping to create *dual-dynamic* network architectures (Wang et al., 2020). Layer skipping is a dynamic inference process which uses a routing controller after each layer to determine if the layer is required, it was first incorporated into ResNet architectures as the structure lends itself to the skipping of layers (Wang et al., 2017).

More recently the field has progressed to dynamic routing on neural networks (Cai et al., 2020). In this paper the authors propose an instance aware inference process that routes inputs only to necessary transformation branches of the network; the method is also generalisable across architectures. Their method outperforms the state of the art in terms of parameter counts and operation counts. They introduce a Gumbel-softmax and a re-parameterisation trick to learn the routing via gradient based methods.

Similarly, attention mechanisms have been deployed in the field of dynamic inference (Chen et al., 2020). The authors present an algorithm that for each convolutional layer an attention score is computed using a lightweight attention model to adaptively weight its kernels. These weights are passed to kernels such that only the necessary kernels are selected to transform the input. A varied temperature scheme is implemented on the attention scheme to assist the optimisation of the layers in the early stages of training.

Dynamic inference techniques, in particular branched neural networks, will play an important role in developing progressive intelligence systems. This is because dynamic inference techniques by design attempt to conform their inference process to computational constraints, whilst retaining predictive accuracy.

Branched networks also remain an integral part of the dynamic inference research community and there has been recent work investigating the optimisation of branched networks, such as that by Hu et al. (2019), which uses an adaptive weighting scheme. They found that using such a scheme could produce models that would outperform larger networks trained using a constant weighting scheme. There has also been recent work investigating the exit policies of the neural network, looking to optimise inference cost and accuracy (Dai et al., 2020).

A particularly relevant area of development to the progressive intelligence problem is the development of inference techniques. Laskaridis et al. (2020) introduce early exiting and train these classifiers simultaneously and gradually stop training the earlier classifiers to prioritise the later classifiers. They show empirical evidence that this is successful and prevents the later classifiers from being outperformed by the earlier classifiers. Their exit policy is quite simple and is the maximum confidence on the softmax output, however they tune these confidences for each exit in the network empirically.

Xin et al. (2020) introduce early exiting branches to language models. They start with a pre-trained BERT model (Devlin et al., 2018) and attach exit branches to this backbone. They use the entropy as a measure of prediction confidence, and thus use it in their exit policies. They fine-tune the thresholds at each branch.

Work by Hu et al. (2020) introduces branched neural networks under a more theoretical lens, showing that they can increase accuracy, adversarial robustness, and efficiency when compared against the backbone architectures. Here these characteristics are co-designed into their approach, by building on the architecture proposed by Teerapittayanon et al. (2016) and applying it to state-of-the-art architectures with adversarial objectives. They find their method can reduce the computational cost of robust models and is some of the first work looking to combine robustness with dynamic inference.

One thing previous works fails to do, is recognise that these branched classifiers act as a special ensemble, and predictions can be treated as such. Sun et al. (2021) address this in their work, where they create an optimisation function for language models which encourages diversity between the branches whilst also preserving their accuracy on the classification task. This is achieved by increasing the divergence between the branches, encouraging them to make different mistakes to each other. They show this creates a more successful branched model.

Wolczyk et al. (2021) introduce this ensemble approach to the inference process of vision models. They train their early exiting branches *as* an ensemble, unlike previous works in the area. They use a geometric ensemble with weighting parameters for each classifier class-specific weights for each classifier, where the parameters are learnt post-training to minimise cross-entropy loss on the training set.

Hence, much of the recent advancements in branched neural networks have focused on improving the performance of the early exit and the policies governing it. Thus, we should briefly cover the main exit policies in the field.

Generalised exit policies will take as input the information produced by the model and predict a class along with its classification confidence. For the b^{th} exit this will be κ_b, which will need to exceed a threshold, α in order for the early exit to take place. If this condition is met, the inference can halt.

$$\text{Generic Threshold:} \qquad \kappa_b > \alpha_b$$

All exit policies will differ in how they define the confidence, κ, and the threshold, α, at a given branch. Some approaches will use a learnt function which will require further optimisation, and some will act on the model outputs directly. First we will investigate those that take the outputs of the model directly.

The two most simple exit policies seen in the field are the max probability and the entropic policies. The maximum probability exit policy uses the maximum value of the output prediction vector on the branch

$$\text{Max Probablity:} \qquad \kappa_b = \max|\hat{y}_b|. \tag{2.9}$$

This has been used a number of times in the field Laskaridis et al. (2020); Dean (2020); Huang et al. (2017). Entropy instead passes the output predictions to the information entropy function

$$\text{Entropic:} \qquad e_b = -\sum_{c}^{C} \hat{y}_{b,c} \ln(\hat{y}_{b,c}), \tag{2.10}$$

where c is a given class in $\hat{y}_b$. The equality should be flipped in this case, since entropy decreases as classification confidence increases. Alternatively, we can take the confidence as $\kappa = 1 - (e/\log(C))$ in this case, since it can be shown the maximum value of entropy is $\log(C)$ which normalises the entropy. This has been the most popular exit policy in the field Teerapittayanon et al. (2017); Hu et al. (2020); Bolukbasi et al. (2017); Xin et al. (2020). In practice entropic policies and max probability policies perform similarly.

More sophisticated approaches tune their ensembles methods, increasing performance at the expense of additional complexity. Work in Huang et al. (2017) solve a constraint, which depends on a predefined batch cost and individual branch costs, which they use to calculate branch thresholds, α_b. They parameterise this confidence using a branch exit probability ψ_b, which itself is a function of a tune-able hyperparameter, ψ:

$$\psi_b = z\,(1-\psi)^{b-1}\psi. \tag{2.11}$$

Where z is a normalising constant to ensure that $\sum_b p(\psi_b) = 1$. This is solved for ψ to generate their thresholds and they use the validation set to ensure thresholds α_b produce roughly the number of exits ψ_b. Hence, this method does require optimisation but could be achieved without additional training steps if performed in parallel with model training. They use the exit policy in equation 2.9 to determine exits at inference time.

Further optimisation work by Wolczyk et al. (2021) incorporates a geometric ensemble based exit function, which they call zero-time waste and is defined by

$$\text{Zero Time Waste:} \qquad \kappa_b = \frac{1}{Z_b}\,\gamma_b^i \prod_{j\le b} \left(\hat{y}_j\right)^{\xi_j} \tag{2.12}$$

where ξ_j and γ_b^i are parameters optimised post training, to minimise cross-entropy loss on the training set. It is not explained why the geometric ensemble is used, and why it works well.

A recent paper extends the ideas of Huang et al. (2017) by incorporating a Bayesian uncertainty model into the training of the classification branches (Meronen et al., 2023). The authors perform a Laplace approximation of the network posterior distribution by taking a Taylor expansion around the maximum a posteriori (MAP) of the model, which they show can be assumed to be that of the output distribution of a model trained with a cross-entropy loss. MAP refers to a probabilistic framework for the estimation of an unknown quantity, which equals the mode of the posterior distribution. Their Bayesian model produces a probability at each branch for a given input i: $p_b(\hat{y}_i|x_i)$. It is this probability which is ensembled and passed to an exit policy, the ensemble probabilities are defined as

$$p_b^{\text{ens}}(\hat{y}_i|x_i) = \frac{1}{\sum_{l=1}^{b} w_l} \sum_{m=1}^{b} w_m\, p_b(\hat{y}_i|x_i)$$

where w_m is a weighting parameter defined by the cost requirement of a particular branch. The maximum probability of this expression is compared to a threshold, which is calculated in a similar manner to Huang et al. (2017) making their exit policy

$$\text{Fixing over confidence:} \qquad \kappa_b = \max\left|\frac{1}{\sum_{l=1}^{b} w_l} \sum_{m=1}^{b} w_m\, p_b(\hat{y}|x)\right|. \tag{2.13}$$

Extending the idea of trainable weights, work in Ilhan et al. (2023) introduces trainable functions which learn to exit efficiently for adaptive inference. They define a batch budget and inference cost, then optimise exit utility functions which output an exit probability. They also train the thresholds alongside these functions according to the batch computation requirements. Their loss function minimises a loss with two components, one is the binary cross entropy loss which predicts the correctness of the branch, the other is a exit assignment estimator, which is a combination of reducing the cost according to the budget and another which predicts the exit probability. This produces an exit assignment score defined by

$$\hat{r}_b = \mathbf{W}_b^{(h)}\,\mathbf{s}_b$$

where $\mathbf{s}_b = f_{leakyrelu}(\mathbf{W}_b^{(sh)}[\hat{y}_b, a_b, b_b])$. In this function a refers to a confidence score vector defined by the scores in the previous section. The parameter b in this function refers to a vector of *exit utility* scores, q_b. Which are themselves computed as $\hat{q}_b = f_{sigmoid}(\mathbf{W}_b^{(g)}\mathbf{s}_b)$. Definitions for the Leaky ReLU and Sigmoid can be found where activation functions are discussed in section 2.2. The $\mathbf{W}$ components are fully connected layers, and hence the trainable components of this method and $\mathbf{s}_b$ is shared between $\hat{r}_b$ and $\hat{q}_b$. At inference time, the output of $\hat{r}_b$ for a given input is used for the exit

$$\text{EENet:} \qquad \kappa_b = \hat{r}_b, \tag{2.14}$$

which outputs a single value, and depends on the specific confidence metric used; the authors use maximum probability, entropic, and agreement based scores. The thresholds are learnt similarly to Huang et al. (2017); Meronen et al. (2023).

We've discussed a number of exit policies, some of which are simple like maximum probability and entropy, and others use additional optimisation. We find that the most popular in the field remains to be entropy, possibly due to its simplicity and the marginal gains further optimisation brings. Hence, it may be beneficial to propose an information sharing exit policy which does not require additional optimisation, starting from an entropic policy. In this section we've also considered other dynamic approaches to inference in the field, some of which more applicable to others. To summarise we consider the potential gaps in the field which can be investigated in this work.

2.4 Next Steps: Gaps in the Literature for Progressive Intelligence

We have seen in this chapter there is a a wealth of active research in and around the areas concerning progressive intelligence, however the area itself still has unanswered questions and there are opportunities to deepen our understanding of existing systems and develop new ideas in the field.

Quantifying Progressiveness A key issue in the field, is that there is no unifying way of quantifying progressiveness. Hence, this can be developed allowing models to be compared with one another from a progressive intelligence perspective.

Exploring Progressive Intelligence Systems Progressive Intelligence has yet to be defined in the context of machine learning. Hence, the types of progressive intelligence and working examples should be proposed. This will allow the most impactful directions to be identified and the open problems within those directions.

Representational Effect of Classification Branches In branched neural networks there has been little work investigating the representational effect of efficient inference strategies, namely classification branches. It will be beneficial to the field to better understand the effect of these branches, when the branches are optimised in parallel with the backbone network and when they are optimised independently of the backbone.

Optimising Architectures for Progressive Intelligence How does scaling affect progressive intelligence systems and can efficient inference techniques/architectures be incorporated with progressive intelligence techniques.

Improving Early Exit Policies Exit policies in the branched neural network field have predominantly been quite simple, being based only on classification confidence at a given exit, and ensemble methods have required additional optimisation. There is an opportunity to develop parameter-free exit policies which exploit the ensemble nature of branched neural networks.

Introducing Epistemic Uncertainty to Progressive Intelligence Systems One thing that is yet to be explored in the literature is how resource-aware neural network designs, in particular branched neural networks, interact with OOD and adversarial data. Furthermore, it is unclear how they might handle samples which have additional information available such as hierarchical label structure. To this end, the field may benefit from exploration of this area.

Introducing New Early Exiting Mechanisms Following from the previous observation, the additional information provided by such inputs may allow new means of early exiting in progressive intelligence systems. Branched networks offer a unique opportunity when encountering OOD and adversarial inputs. Specifically, if they can be detected, inference can be halted to prevent further time wastage. This possibility remains an open question in the progressive intelligence space.

Hence, there are a number of opportunities to explore new ideas in progressive intelligence. We begin by exploring progressive intelligent systems and developing a way of quantifying their performance in chapter 3. This is followed in chapter 4 by an exploration of branched neural networks. Namely we concentrate on the representational effects of classification branches, how the architectures can be optimised, and how classification policies can be improved. Finally, in chapter 5 we consider epistemic uncertainty and additional information in classification policies.

Chapter 3

Data-Space vs Model-Space

Consider a machine learning model, if this model is to be incorporated into a progressive intelligence system it needs to be scalable. In this context, much like machine learning systems, progressive intelligence can be considered from two aspects: The *model space*, and the *data space*.

In the model space, the learning machine itself is scaled to achieve progressive intelligence. That is, the resources used by the model are independent of the amount of data it is trained on. Progressive intelligence applies only to the inference modes of machine learning systems. Hence, once trained, a model-space progressive intelligence system uses a predefined amount of power on a given data point. In the data space, we consider machine learning models where there is a direct link to the computational expense of the model and the amount of data it is trained on. There is also another aspect of this space, whereby the number of features in the data can be scaled. However, for now only the former will be considered.

This more theoretically focused chapter seeks to address the first gap in the literature: *Exploring Progressive Intelligence Systems*. We make the following contributions:

Quantifying Progressiveness We develop a way of quantifying *progressiveness*, that can be used where the performance of competing methods overlap in their range of computational cost.

Explore Upper Bounds in Data-Space Systems Data-space progressive intelligence is explored theoretically from the perspective of sample counts. We develop expected error estimates in this domain, which when compared to prototype KNN algorithms we find to represent an upper bound for the error.

Prototype Model-Space Systems We move towards obtaining empirical results in the model based domain, where the complexity of tasks are often greater. Ensembles and branched neural networks are presented as prototypes of model-based systems, we settle on branched neural networks for future work due to their increased progressiveness and performance.

Before investigating these types of progressive intelligence ideologies, it is necessary to define a metric that can be used across these systems. This will provide a means of quantifying their performance in the progressive intelligence space. That is, we wish to define their *progressiveness.*

3.1 Quantifying Progressiveness

We will look to investigate systems that take pareto-optimised solutions as in figure 1.1 and use them in their design methodology. As such they will be represented by curves in the pareto-efficiency space, as opposed to individual points in figure 1.1. For example, these solutions might make fast gains in accuracy early in the inference stage and slowly increase the performance when using more computational resources. However, one solution may be preferred over the other as resource constraints change. Hence, a goal of this work will be to find ways of quantifying the *pareto-optimality* of a system.

So we have two goals for our progressive intelligence, we want monotonicity in performance improvements as we increase the computational resources used, whilst also prioritising early gains in this performance. Hence, an initial measure of pareto-optimality that will be incorporated into this work is the area underneath a pareto-curve. We can show that this measure is equivalent to linearly weighting gradients earlier in the curve, and integrating this over all values of x.

Consider a pareto-curve $f(x)$ measuring performance where x is the resource usage. We can take the area under this curve as our measure of performance, M:

$$M(f(x)) = \int_0^1 f(x)\ dx = \text{AUC}$$

Early improvements are also desired, that is for low values of x we wish for $\frac{d}{dx}f(x)$ to be high, and as x increases we care less about the value of $\frac{d}{dx}f(x)$, and more on the value of $f(x)$ itself. Following this, we can construct a simple point-wise measure of performance, m

$$m(f(x)) = (1-x)\frac{d}{dx}f(x). \tag{3.1}$$

To get our complete measure of performance, we need to integrate this expression over all values of x:

$$M(f(x)) = \int_0^1 (1-x)\frac{d}{dx}f(x)\,dx = \Big[(1-x)f(x)\Big]_0^1 - \int_0^1 f(x)\Big(\frac{d}{dx}(1-x)\Big)\,dx$$
$$= -f(0) + \int_0^1 f(x)\,dx$$

If we now make the reasonable assumption that $f(0) = 0$, that is when the cost is zero, our performance is the same, we find we arrive back at the AUC, giving us

$$M(f(x)) \quad = \quad \int_0^1 (1-x)\frac{d}{dx}f(x)\,dx \quad = \quad \int_0^1 f(x)\,dx \quad = \quad \text{AUC}. \tag{3.2}$$

Hence, equation 3.2 shows us that the area under the pareto-curve is a more powerful measure of progressiveness than it might first appear. The AUC will be increased if there are early rises in performance and also if these are maintained later in the performance–cost curve, thereby encouraging monotonicity. That is, the AUC encourages monotonicity and weights this early in the cost distribution.

However, this measure does not consider improvements at later areas of the curve, two models may have equal areas, but perform very differently across different power ranges. This is particularly true in later power ranges, where the highest accuracies will be desired. This could be improved in some way by changing the weighting factor in m shown in equation 3.1, from $(1-x)$ to something more complex. Though, for the time being the AUC will suffice and future work can consider more complex measurements of progressiveness. We can now use this method when evaluating progressive intelligence systems, and first we can consider data-space progressive intelligence.

3.2 Data-Space Progressive Intelligence

In the data space, we consider the progressive intelligence systems which use models that are dependent on the size of their training set. For example KNN algorithms and SVMs. Here, there is an innate dependence on the number of samples used to train the system, and the power used by it is directly proportional to the number of training samples, N.

In this section, we analyse this relationship analytically with the aim of understanding how it affects the error achieved by a classification model. We wish to understand how the estimation of the decision boundary, t, converges with the number of samples, N. This will give an estimate on the performance of a classifier on this data, as the number of training samples is increased.

To that end, we start with a simple problem and make the assumption that the data for each class is drawn from a Gaussian distribution. We analyse the problem with varying levels of complexity in the following sections, starting with equal 1-dimensional standard deviations, ending in the general case with multidimensional Gaussians of differing covariances.

In each case we form a theoretical relationship between our estimation of the decision boundary and the number of samples, starting with a 1-dimensional Gaussian distribution

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}.$$

we consider how the performance of a binary classifier behaves as the training set size varies. Assuming that both classes are equally probable, it can be shown the decision boundary, t, will occur where the two distributions overlap. That is, the decision boundary will lie where both classes are equally probable, when $\mathbb{P}\left(x|y=1\right) = \mathbb{P}\left(x|y=2\right)$. Since the probability distributions are Guassians we therefore have:

$$f_1(x) = f_2(x) \quad \rightarrow \quad \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{1}{2}(\frac{t-\mu_1}{\sigma_1})^2} = \frac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{1}{2}(\frac{t-\mu_2}{\sigma_2})^2}$$

By taking the natural log of both sides and removing shared terms we get:

$$-\ln(\sigma_1) + \frac{1}{2}\left(\frac{t-\mu_1}{\sigma_1}\right)^2 = -\ln(\sigma_2) + \frac{1}{2}\left(\frac{t-\mu_2}{\sigma_2}\right)^2 \tag{3.3}$$

3.2.1 Equal σ

We start in the simplest case. If we assume the two distributions have identical standard deviations, σ, this gives us

$$t = \frac{\mu_2^2 - \mu_1^2}{2(\mu_2-\mu_1)} = \frac{(\mu_2-\mu_1)(\mu_2+\mu_1)}{2(\mu_2-\mu_1)} = \frac{\mu_1+\mu_2}{2}. \tag{3.4}$$

Hence, our optimal solution for t is entirely dependent on the means, μ_1 and μ_2. When estimating the mean of a distribution, the standard error of the mean, $\Delta\mu$, is defined as:

$$\Delta\mu = \frac{\sigma}{\sqrt{N}} \tag{3.5}$$

Using this we can derive an expected value of Δt, as we increase N, and assuming the means are independent of each other. We can use the following formula to obtain our uncertainty in the value for t.

$$\Delta t = \sqrt{\left(\frac{\partial t}{\partial\mu_1}\right)^2(\Delta\mu_1)^2 + \left(\frac{\partial t}{\partial\mu_2}\right)^2(\Delta\mu_2)^2} \tag{3.6}$$

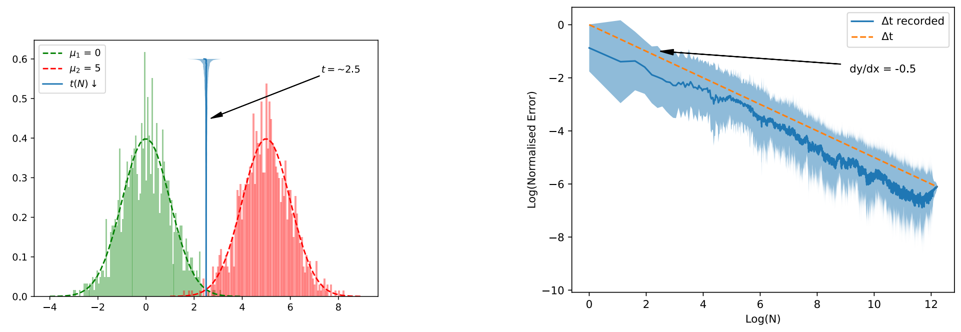
We use equations 3.4 to 3.6 to derive our expected error in t. Finding $\frac{\partial t}{\partial\mu_i}$ in equation 3.4 is trivial, $\frac{\partial t}{\partial\mu_i} = \frac{1}{2}$. Applying this, along with our error in μ_i, to equation 3.6 we arrive at:

$$\Delta t = \sqrt{\left(\frac{1}{2}\right)^2\left(\frac{\sigma}{\sqrt{n_1}}\right)^2 + \left(\frac{1}{2}\right)^2\left(\frac{\sigma}{\sqrt{n_2}}\right)^2} \rightarrow \Delta t = \frac{\sigma}{2}\sqrt{\frac{1}{n_1}+\frac{1}{n_2}}$$

Where n_c refers to the number of samples taken from class distribution c, and $\sum_i^C n_c = N$. If the classes are evenly distributed, we can assume even sampling from each as N becomes increasingly large and n_c becomes N/C. Hence, for the two class problem we have:

$$\Delta t = \frac{\sigma}{\sqrt{N}} \tag{3.7}$$

Indeed this is what we observe when simulating the optimal decision boundary, as shown in figure 3.1.



(a) Two distributions with different means and identical standard deviations. The estimation in t with N over 50 runs is shown.

(b) Log of expected error with Log of N, the gradient is -0.5 showing that $\Delta t \propto N^{-1/2}$. The shaded area shows the standard deviation over 50 runs.

FIGURE 3.1: Experimental verification of the our uncertainty in t is shown, in the case where $\mu_1 = \mu_2$. In both figures the dashed lines show the theoretical models, and the solid lines show the experimental verification.

Hence, we have an understanding of our error as a function of N in a data-based system in the simplest case. Allowing us to provide confidence estimates in predictions made by such a system. However, this was derived in the simplest case, to ensure this holds as problems become more challenging we can derive this relationship in the more complex cases.

3.2.2 Unequal σ

Now we consider the case where the standard deviations are not equal. This makes equation 3.3 more involved to solve, and collecting the terms reveals a quadratic in t:

$$t^2\frac{1}{2}\left(\frac{1}{\sigma_1^2}+\frac{1}{\sigma_2^2}\right)+t\left(\frac{\mu_2}{\sigma_2^2}-\frac{\mu_1}{\sigma_1^2}\right)+\frac{1}{2}\left(\frac{\mu_1^2}{\sigma_1^2}-\frac{\mu_2^2}{\sigma_2^2}\right)+\ln\left(\frac{\sigma_2}{\sigma_1}\right)=0$$

Which we can solve by completing the square, utilising the quadratic equation:

$$ax^2+bx+c=0 \qquad \rightarrow \qquad x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

where:

$$a = \frac{1}{2}\left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}\right) \qquad b = \frac{\mu_2}{\sigma_2^2} - \frac{\mu_1}{\sigma_1^2} \qquad c = \frac{1}{2}\left(\frac{\mu_1^2}{\sigma_1^2} - \frac{\mu_2^2}{\sigma_2^2}\right) + \ln\left(\frac{\sigma_2}{\sigma_1}\right) \tag{3.8}$$

Plugging the coefficients of equation 3.8 into the quadratic equation gives us:

$$t = \frac{1}{(\sigma_1^2 - \sigma_2^2)}\left(\mu_2\sigma_1^2 - \mu_1\sigma_2^2 \pm \sqrt{\mu_1^2\sigma_1^2\sigma_2^2 - 2\mu_1\mu_2\sigma_1^2\sigma_2^2 + \mu_2^2\sigma_1^2\sigma_2^2 - 2\sigma_1^4\sigma_2^2\ln\left(\frac{\sigma_2}{\sigma_1}\right) + 2\sigma_1^2\sigma_2^4\ln\left(\frac{\sigma_2}{\sigma_1}\right)}\right)$$

Which can be simplified to:

$$t = \frac{1}{(\sigma_1^2 - \sigma_2^2)}\left(\mu_2\sigma_1^2 - \mu_1\sigma_2^2 \pm \sigma_1\sigma_2\sqrt{(\mu_1 - \mu_2)^2 + 2(\sigma_2^2 - \sigma_1^2)\ln\left(\frac{\sigma_2}{\sigma_1}\right)}\right) \tag{3.9}$$

Once again, in order to get the expected error in t we need to use equation 3.6. The partial derivatives for equation 3.9 are as such:

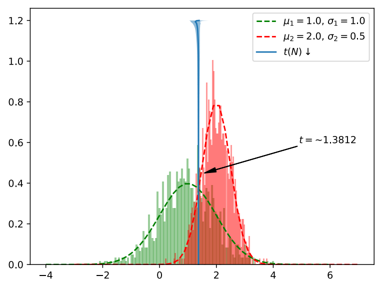
$$\begin{aligned} \frac{\partial t}{\partial \mu_1} &= \frac{-\sigma_2^2 \pm \dfrac{(\mu_1\sigma_1^2\sigma_2^2 - \mu_2\sigma_1^2\sigma_2^2)}{\sqrt{\mu_1^2\sigma_1^2\sigma_2^2 - 2\mu_1\mu_2\sigma_1^2\sigma_2^2 + \mu_2^2\sigma_1^2\sigma_2^2 - 2\sigma_1^4\sigma_2^2\ln(\frac{\sigma_2}{\sigma_1}) + 2\sigma_1^2\sigma_2^4\ln(\frac{\sigma_2}{\sigma_1})}}}{(\sigma_1^2 - \sigma_2^2)} \\ \frac{\partial t}{\partial \mu_2} &= \frac{\sigma_1^2 \pm \dfrac{(-\mu_1\sigma_1^2\sigma_2^2 + \mu_2\sigma_1^2\sigma_2^2)}{\sqrt{\mu_1^2\sigma_1^2\sigma_2^2 - 2\mu_1\mu_2\sigma_1^2\sigma_2^2 + \mu_2^2\sigma_1^2\sigma_2^2 - 2\sigma_1^4\sigma_2^2\ln(\frac{\sigma_2}{\sigma_1}) + 2\sigma_1^2\sigma_2^4\ln(\frac{\sigma_2}{\sigma_1})}}}{(\sigma_1^2 - \sigma_2^2)} \end{aligned} \tag{3.10}$$

Equation 3.9 produces two boundaries each with slightly different partial derivatives. The correct boundary, and thus correct partial derivatives, can be found by selecting that which lies between the two means. Similarly to figure 3.1, we present experimental validation of this in figure 3.2. This shows that when applying equations 3.10 and 3.5 to the expression for uncertainty in t, equation 3.6, we arrive at the the same convergence behaviour with N, $\Delta t \propto N^{-1/2}$.

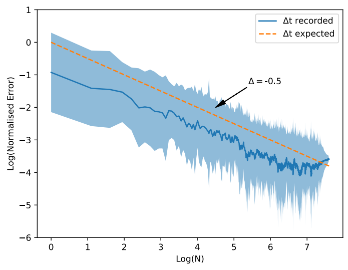
So we arrive at the same finding, with unequal standard deviations. Suggesting at least in the 1-dimensional case we have a theoretical measure of uncertainty in a data-based classifier. In the next section we analyse this problem in the most general case.

3.2.3 The d-Dimensional Case

Finally, in the general case we lift the 1-dimensional restriction and can no longer consider the standard deviation, we instead need to consider the distributions as being defined by a mean vector, $\boldsymbol{\mu}$, and Covariance matrix, $\boldsymbol{\Sigma}$. However, much of the expressions derived in the previous sections still hold in this case, but are instead derived using the multi-dimensional



(a) Two distributions with different means and standard deviations. The estimation in t with N over 50 runs is shown.



(b) Log of expected error with Log of N, the gradient is -0.5 showing that $\Delta t \propto N^{-1/2}$. The shaded area shows the standard deviation over 50 runs.

FIGURE 3.2: Experimental verification of the our uncertainty in t is shown, in the case where $\mu_1 \neq \mu_2$. In both figures the dashed lines show the theoretical models, and the solid lines show the experimental verification.

Gaussian equation,

$$p\big(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}\big) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})},$$

where we now consider a probability density. Once again we can equate these distributions, and if equal probabilities between them are assumed, any priors will cancel out leaving

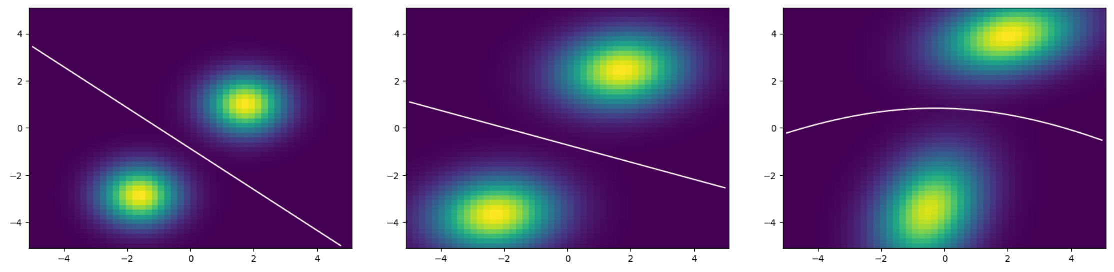
$$-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{1}(\boldsymbol{x}-\boldsymbol{\mu}) - \frac{1}{2}\ln|\boldsymbol{\Sigma}| - \frac{d}{2}\ln(2\pi),$$

on either side. We will find the decision boundary, $\boldsymbol{t}$, at points where these two distributions are equal:

$$-\frac{1}{2}(\boldsymbol{t}-\boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\boldsymbol{t}-\boldsymbol{\mu}_1) - \frac{1}{2}\ln|\boldsymbol{\Sigma}_1| = -\frac{1}{2}(\boldsymbol{t}-\boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{t}-\boldsymbol{\mu}_2) - \frac{1}{2}\ln|\boldsymbol{\Sigma}_2|. \tag{3.11}$$

Solving this equation produces a hyperquadratic boundary in d dimensions, however we note that the aspects of dimensionality drop out of our expression for the boundary. The quadratic expression will have similar terms to our expressions in equation 3.8, only using the vector and matrix form for the mean and covariance respectively. Each side consists of two terms, the square of the Mahalanobis distance to the mean, and a constant which is a function of the covariance of the distribution, we can equate these to find our boundary numerically:

Since we are only interested in the convergence behaviour with N, we wish to understand if the added dimensions add complexity to the $\Delta t \propto N^{-1/2}$ we see in the 1-dimensional case. Our expression for the boundary in equation 3.11 has no dependence on the dimensionality, d, and we can assume the quadratic has similar roots to its scalar equivalent in equation 3.8, and thus a similar $\frac{\partial t}{\partial \mu_i}$. This means the only dependence of Δt on N will come from the expression for the uncertainty in the mean.



(a) The simple case where distributions have the same covariance matrix, with independent variables.

(b) The simple case where distributions have the same covariance matrix, with dependent variables.

(c) More complex case where distributions have different covariance matrices.

FIGURE 3.3: Numerical solutions to find the optimum decision boundary between two 2-dimensional Gaussian distributions. In the simple case (a), the boundary can be defined once again using the distance of the means. In the dependent variable case (b), the linear boundary is modified by the covariance matrix. Finally the complex case (c) lifts all assumptions and a quadratic boundary is required.

In the d-dimensional case, the uncertainty in the mean becomes a vector, since the mean itself is also a vector, $\boldsymbol{\mu} = (x_1, .., x_d)$. We assume the variance in x_i as σ_i^2 and the covariance between x_i and x_j is σ_{ij}. Our expected value of the mean, $\overline{\boldsymbol{\mu}}$, will have a covariance matrix, with diagonal elements defined with σ_i^2/n and off-diagonal elements

$$\mathrm{Cov}(\bar{x_i}, \bar{x_j}) = \frac{1}{n^2}\sum_i^n \sum_{j\neq i}^n \mathrm{Cov}(x_i, x_j) \rightarrow \frac{\sigma_{ij}}{n},$$

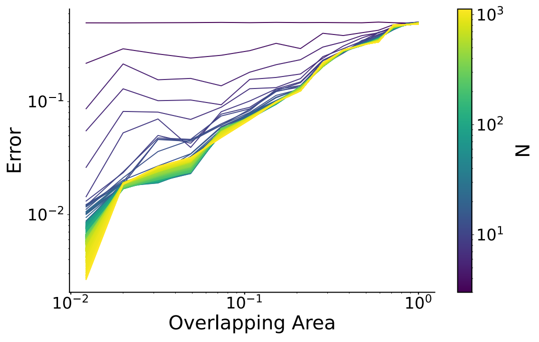
where i and j refer to the dimension of the distribution. Hence, the covariance of $\overline{\boldsymbol{\mu}}$ is the covariance of the original matrix, divided by the number of samples, n. Our standard error in $\boldsymbol{\mu}$ is the root of this expression, leaving us with $\Delta\boldsymbol{\mu}_i \propto (\frac{N}{2})^{-1/2}$ in equation 3.6 and thus $\Delta \boldsymbol{t} \propto N^{-1/2}$.

Hence, we've shown that in a simple Gaussian classification problem, it can be shown the error in our estimation of the decision boundary is proportional to the inverse of the square root of N. In the 1-dimensional case we've verified this by simulating a distance between the means classifier. This will allow us to estimate the accuracy and thus confidence of data-based progressive intelligence systems as we increase the sample size available. Now we move to problems closer to those in the real world, this will allow us to verify this behaviour with N. We can also prototype a data-space progressive intelligence classifier which can be used incrementally at inference time.

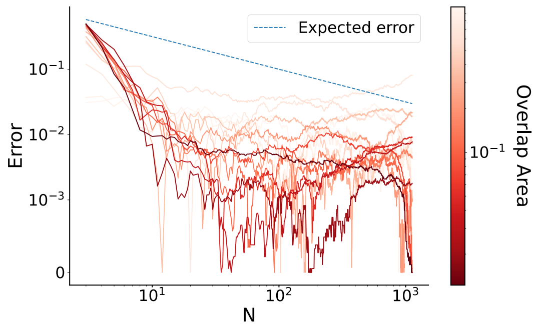
3.2.4 Prototyping Data-Space Progressive Intelligence

We can first evaluate the ideas of the previous section on more difficult problems by shifting the two Gaussian distributions. To quantify the difficulty of the classification between the two distributions we can use their overlapping area, since the distance between the means alone does not reflect the potential for differing standard deviations.

By shifting these distributions and evaluating the calculated threshold against the theoretical threshold, we can observe the error as overlapping area increases, as well as the error with increasing number of samples, N. These are shown in figures 3.4 (a) and (b) respectively. This allows us to see the behaviour as both vary.



(a) Log of Error against log of overlapping area for a variety of sample sizes.



(b) Log of error with log of N for a variety of overlapping fractions, the theoretical behaviour is shown with a dashed line.

FIGURE 3.4: Relationships showing error with classification difficulty and sample sizes, the shading of the lines denotes the other variable in both plots.

We find from figure 3.4 (a) that for very small sample sizes the overlapping area has little effect on the recorded error. However, as the sample size increases this quickly converges to a linear relationship in the log-log space. We see this sharp decrease in error in figure 3.4 (b) as N increases to an order of 10. However, then roughly the expected relationship with N is observed for smaller overlapping fractions, though the recorded values for this are noisy.

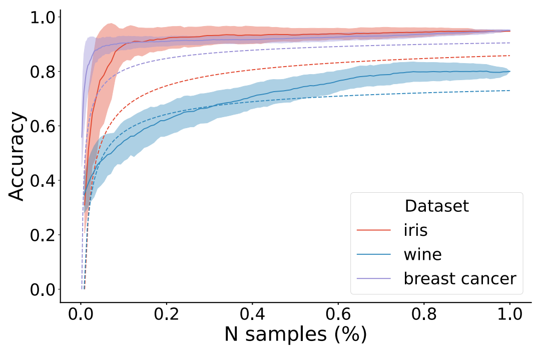
While this allows us to analyse the theoretical performance of a progressive intelligence classifier, the problem proposed is not necessarily representative of a real-world problem. Furthermore, this classification regime relies on a binary problem and for multi-class problems a more general classification algorithm is needed, such as a KNN classifier.

We can use a KNN classifier progressively by incrementally sampling the validation distribution at run-time in a progressive manner, thereby increasing the confidence in the classification as the sample size over which to calculate the KNN distance increases. We use 3 real-world tabular datasets:

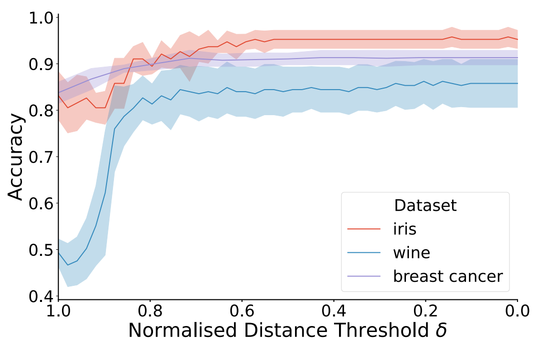
iris	Four-class flower classification dataset with 4 dimensions denoting measurements of the flower (Fisher, 1988).
wine	Multi-class classification dataset with 13 different chemical and colour measurements for identifying three wine types (Aeberhard and Forina, 1991).
breast cancer	Binary classification dataset using measurements of 30 different biological markers to predict the presence of breast cancer (Wolberg and Street, 1995).

We first assess the importance of N by analysing the accuracy of the KNN algorithm as the number of samples in the training set increases. This is shown in figure 3.5 (a). This rise is rapid in the iris and breast cancer datasets, and more gradual in the wine dataset. Furthermore, the peak accuracy is lower in the wine dataset, indicating the dataset being of high difficulty.

In each case the theoretical estimate is shown with a dashed line of the same colour. Since 100% classification accuracy is not always possible, we normalise our error to the peak classification accuracy of the classifier. We find that in all cases our estimate is close to, or worse than, the true relationship with N. Hence, in these real-world examples our theoretical expectation of error represents an upper bound on the true-error of the classifier.



(a) Accuracy against normalised sample counts, N. Our expectation of the error is shown with the dashed lines of the same colour.



(b) Accuracy against distance threshold for adaptive KNN classification.

FIGURE 3.5: Accuracy against the number of samples used for KNN classification in (a) and in (b) the distance threshold for adaptive KNN inference against the accuracy achieved.

Whilst this graph does reflect the relationship between the accuracy of a KNN classifier and the number of reference samples for training, it does not reflect how the model would be used for progressive intelligence. This is because progressive intelligence operates at inference time so the number of training points being used should vary between each test sample.

We call the criteria on which the number of samples should stop increasing the *exit policy*, which is described further for branched neural networks in section 2.3.2. In the case of the KNN model we can use the mean distance of the k-nearest neighbours for our condition to halt inference, as the relative proximity to our nearest k samples will indicate the reliability of the prediction. For this we denote the *distance threshold*, δ. The policy is described in algorithm 1.

The modal value of the y labels corresponding to the training data points x in the nearest neighbours d_k is taken for the final output, as this value will correspond to the most common neighbour. In the event of a tie, the closest of the tied classes is taken, but k is typically odd and greater than the number of classes which helps avoid ties. This is repeated for each test sample, x. We simulate this inference process, varying the normalised distance threshold δ between 1 and 0, corresponding to minimum and maximum confidence respectively. The results of which are shown in figure 3.5 (b).

Algorithm 1 Adaptive KNN early exiting algorithm

```
1:  procedure EARLY EXITING INFERENCE WITH KNN($X$, $\delta$)
2:      Read Test Data: $D$

3:      for $x \in D$ do                                    ▷ For all inputs in the test set
4:          for k $\in K|X$ do          ▷ For all available values of $k$ in the training set, $X$
5:              $d_k = \min(\text{dist}(x, D))_k$                        ▷ minimum $k$ distances
6:              if $\bar{d}_k \leq \delta$ then          ▷ Check mean distance against threshold
7:                  Break
8:              end if
9:          end for                                 ▷ Full inference used without early exit
10:         $y = \text{mode}\big(y : (x, y) \in d_k\big)$   ▷ Output is modal value of $y$ given (x':y) in $d_k$
11:         return $y$                                                     ▷ Return output
12:     end for                                                      ▷ Move to next input

13: end procedure
```

We find that lowering the normalised distance threshold increases the accuracy rapidly, particularly in the breast cancer dataset. It has little effect past the peak accuracy ~ 0.8. In contrast to the behaviour seen in the Accuracy–N plot, we find that the peak accuracy is reached more quickly in the adaptive inference experiment. We also find that the variance, denoted by the shaded area, is relatively consistent across the threshold range, whereas in the Accuracy–N analysis, this reduces to negligible values towards the end of the range. This is because when all of the samples are used, the inference outcome will be identical regardless of the order they were added across runs. In the adaptive inference plot however, the test set is randomly sampled between 5 runs, whereas it is constant between runs in the former plot.

Finally, to quantify the performance of the methods we analyse the AUC of all of the curves in line with the discussion in section 3.1. This is shown in table 3.1. We find that the increased

Dataset	AUC	
	Accuracy – N	Adaptive Inference
Iris	0.909 ± 0.035	0.926 ± 0.026
Wine	0.700 ± 0.046	0.806 ± 0.054
Breast Cancer	0.917 ± 0.018	0.904 ± 0.015

TABLE 3.1: AUC measurements for the KNN algorithm on the datasets, left column shows this as sample size increases, the right column shows that as the distance threshold increases in the adaptive inference method.

performance shown in the iris dataset leads to greater AUC performance in the adaptive inference results, despite the breast cancer dataset showing a sharper rise in accuracy the AUC is limited by peak accuracy. However, in the Accuracy-N results, the sharp rise in the breast cancer dataset leads to a greater AUC value. The wine dataset records lower AUC values due to the poor performance of the underlying algorithm and the difficulty of the dataset.

In this section we have discussed data space progressive intelligence, theoretical bounds, and proposed a prototype of a progressive intelligence system in this space. Although, this method is performant as a progressive intelligence, the resource usage in this problem space is not an issue due to the relative low cost of the classifiers compared to more modern approaches. We hence move towards model-space progressive intelligence where there is a wider range of application areas, such as vision, text, and audio. We focus on the vision space as it has more accessible benchmarks at the time of writing.

3.3 Model-Space Progressive Intelligence

Data-space progressive intelligence is one way to approach the problem of progressive intelligence. However, in real-world scenarios the underlying machine learning models are not necessarily that computationally intensive to begin with, meaning the resource saving benefits of using progressive intelligence approaches may be less important.

In model-based progressive intelligence systems it is the model itself which is adapted at run-time to save power. As such, model-based progressive intelligence is more applicable to algorithmically complex machine learning approaches such as neural networks, as well as less complex approaches such as random forests.

To explore model-based systems we consider the progressive intelligence system made up of a sequence of N functions

$$F(x)_N = [f_1(x), f_2(x), ..., f_N(x)].$$

In this paradigm the inference mode of F uses each of the functions f_n sequentially; we consider the scenario where they are used in ascending order, and the cost of using F sequentially, $E(F_n)$, should always satisfy

$$E(F_n) < E(F_{n+1}).$$

Furthermore, as n increases the output of the model $\hat{Y}$ should increase in confidence and ideally accuracy. Since in this scenario the amount of power being used is input-dependent, it is possible to halt inference as confidence increases. In order to ensure computation is not wasted, the outputs of f_n can be combined as an ensemble according to a combination function Ψ, like so

$$\hat{Y}_n(x) = \Psi[\hat{y}_1 \cdots \hat{y}_n].$$

Where $\hat{y}_n$ represents output probabilities of each model, defined using the softmax function (see equation 2.8). Hence, in model-based systems the output $\hat{y}_n$ is developed incrementally, likewise the system itself increases in complexity incrementally. They can represent individual models that combine outputs as we consider above, or they can also represent subsets of an overarching function. That is, F might represent a unified model that can operate at varying levels of complexity, for example branched neural networks which we consider in section 3.3.2, and were discussed in detail in section 2.3.2.

The former has been explored by Venkataramani et al. (2015), and while it can be considered a progressive intelligence system, it does not optimise the use of computation as outputs are discarded when confidence requirements are not met. For this reason we start our study by building on this work, with a prototype progressive intelligence system in the ensemble scenario where outputs are combined with one-another.

3.3.1 Prototyping a Model-Based Progressive Intelligence

As mentioned at the end of section 3.2, we now consider vision problems, specifically image classification. For this reason we choose a simple convolutional neural network for the classifier to be ensembled. Work by Springenberg et al. (2015) describes the *all_conv* neural network, a model composed entirely of convolutional layers, thus reducing the memory overhead of parameter-rich linear layers. Though, as discussed in section 2.2 convolutional layers are expensive in regard to MACs, unlike linear layers. These networks have shown to be capable of achieving reasonable performance in baseline benchmarks such as MNIST and CIFAR10, and have been used in many prominent papers since, for example that by Frankle and Carbin (2018).

We select a number of identical, 4-layer all_conv architectures, and reduce the number of convolutional channels to employ them as an ensemble. As a simple starting point, we train the model with a voting based loss function. That is the ensemble output is calculated using an evenly weighted sum of each member of the ensemble,

$$\Psi_N = \frac{1}{N}\sum_{n}^{N} \hat{y}_n,$$

is the corresponding combination function. We find that the model successfully trains on the dataset, though not achieving state-of-the-art accuracy the ensemble was capable of achieving accuracy similar to that in previous papers using the same architecture (Frankle and Carbin, 2018). This is shown in the training curves in the left hand graph of figure 3.6.

There is nearly a 10% disparity in accuracy between the train and validation sets, so the ensemble is overfitting to some extent. However, it does optimise fairly quickly relative to the total training time. To better understand the role of each model in the ensemble we analyse the accuracy curves of the individual models in the ensemble, shown in the right-hand graph in figure 3.6.

We find that when trained in this manner one classifier from the selection of identical classifiers dominates the learning, with other classifiers in the ensemble providing little contribution. Furthermore this behaviour is consistent across the training process. In the initial

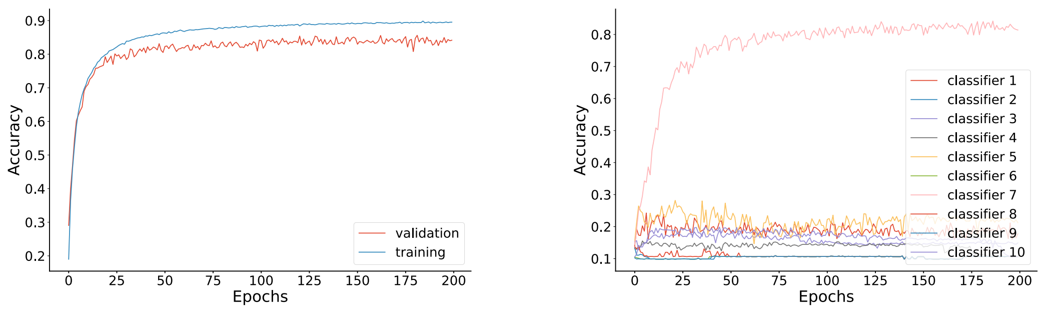


FIGURE 3.6: Training curves for a 10 model ensemble composed of all_conv4 convolutional neural networks. Accuracy is shown on the *y* axis, and epochs on the *x* axis. Data is collected on the CIFAR10 dataset (left). Training curves for all 10 models of the ensemble composed (right), data is collected on the CIFAR10 dataset.

stage of learning, it appears a number of the classifiers are being optimised, notably the pink, yellow, and red lines. However, the classifier denoted by the pink line is completely dominant by 15 epochs. This may be due to how the voting loss is formulated in this scenario, the output of each model is aggregated before being passed to the loss function. Meaning if one model performs well on the classification task there is no benefit to the learning process in minimising the loss of other classifiers.

To test this principle we re-formulate the loss function, and instead take the loss of each classifier and take the mean of the losses. We find that the model performs similarly but with more of the classifiers performing well individually. The results are shown in figure 3.7.

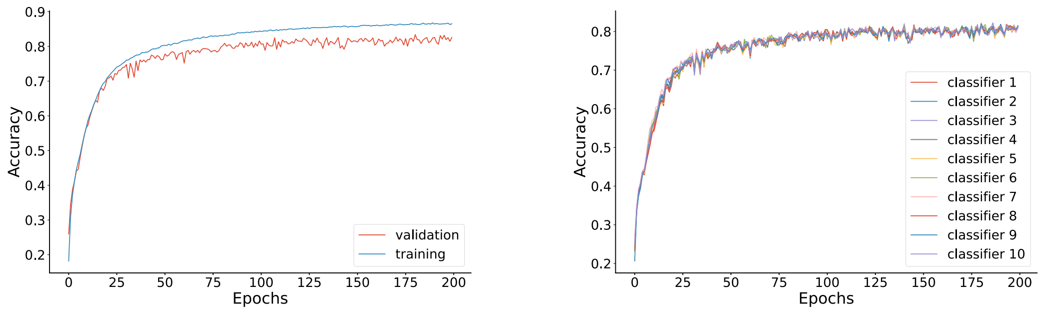


FIGURE 3.7: Training curves for a 10 model ensemble composed of all_conv4 convolutional neural networks. Accuracy is shown on the *y* axis, and epochs on the *x* axis. Data is collected on the CIFAR10 dataset (left). Training curves for all 10 models of the ensemble composed (right). This time the losses are taken before summation in the voting loss, data is collected on the CIFAR10 dataset.

Much like the KNN classifiers in the previous section, we can treat the ensemble as a progressive intelligence system by gradually combining their outputs until a confidence requirement is met. Since each classifier incurs the same cost in this case, we can order our classifiers in reverse order of performance for the fastest gains. This is similar to Venkataramani et al. (2015), only we examine a system which operates as an ensemble at inference time and does not discard outputs.

Early exiting inference in this case uses the intermediate output of the system to determine whether the classification is confident enough. The intermediate output, n will come from the mean output of the n^{th} ensemble, calculated using Ψ_n.

The general procedure for early exiting inference is shown in algorithm 2, where each model in the ensemble is evaluated sequentially on a given input and the mean output is used to calculate a confidence value. If the confidence requirement is reached at a given ensemble the inference process can be halted. When the inference process is halted, the inference cost of subsequent models is saved, making the system more computationally efficient than using the full ensemble on the input sample.

An integral part of the early exiting inference is the criteria determining the confidence of classification. The most commonly used metric for confidence is the entropy of the output distribution (see section 2.3.2). Since the final layer output of a neural network is raw logits, this is typically passed through a softmax function to ensure the output is normalised. This allows us to consider it as a categorical distribution of class probabilities.

Entropy based early exits use entropy as a measure of uncertainty. Entropy is defined as information entropy, formalised in equation 2.10.

When the probability of a single class approaches one, the entropy will approach zero. When all of the class probabilities are equal, total uncertainty, the distribution approaches the maximum value, $\log C$. Consequently this flips the condition in algorithm 2, thus entropy can be considered a measure of uncertainty and the inverse a measure of confidence

$$\kappa(x) = 1 - \frac{e(x)}{\log C}. \tag{3.12}$$

Hence, entropy represents a simple measure of confidence and is widely used in the field. We recreate this exit policy in algorithm 2, and vary the threshold T to present a model that can operate under varying confidence thresholds.

Using the models trained in this section we present the results of this inference on the CIFAR10 dataset in figure 3.8. Here, the confidence is varied from its minimum value of zero, and maximum value of one. This produces a curve which represents the operating range of the model. We find that the model trained using the voting loss, whilst having much higher variance in its performance largely outperforms the weighted training loss. However, it is worth noting the small range in accuracy, as such the tangible performance benefits in accuracy are limited.

Since both ensembles will operate in identical power ranges, we can compare their utility for progressive intelligence by taking the AUC measurements. This is shown in table 3.2. We find, as expected, from the graph that the voting model performs better than the weighted model. However, the variance in this model does mean the worst case voting model is outperformed by all weighted models.

Algorithm 2 Early exiting inference algorithm for the progressive ensemble

```
1: procedure EARLY EXITING INFERENCE WITH ENSEMBLE($f, X, \alpha$)
2:     Load model: $f$
3:     Read data, Threshold: $X, \alpha$
4:     for $x \in X$ do                                      ▷ For all inputs in the test set
5:         for $n \in N$ do                                  ▷ For all models in ensemble
6:             $\Psi_n(x) = \frac{1}{n}\sum_{n'}^{n} f_{n'}(x)$                 ▷ Get ensemble output
7:             $\hat{y}_n = \arg\max(\Psi_n(x))$                 ▷ Calculate ensemble prediction
8:             $\kappa_n = \text{confidence}(\Psi_n(x))$            ▷ Calculate confidence
9:             if $\kappa_n \geq \alpha$ then                    ▷ Check confidence threshold exceeded
10:                return $\hat{y}_n$                           ▷ Return classification if confident
11:            end if
12:        end for                                        ▷ Full inference used without early exit
13:        $\Psi_N(x) = \frac{1}{N}\sum_{n'}^{N} f_{n'}(x)$                    ▷ Get final output
14:        $\hat{y}_N = \arg\max(\Psi_N(x))$
15:        $\kappa_N = \text{confidence}(\Psi_N(x))$
16:        return $\hat{y}_N$                                   ▷ Return final output
17:    end for                                               ▷ Move to next input
18: end procedure
```

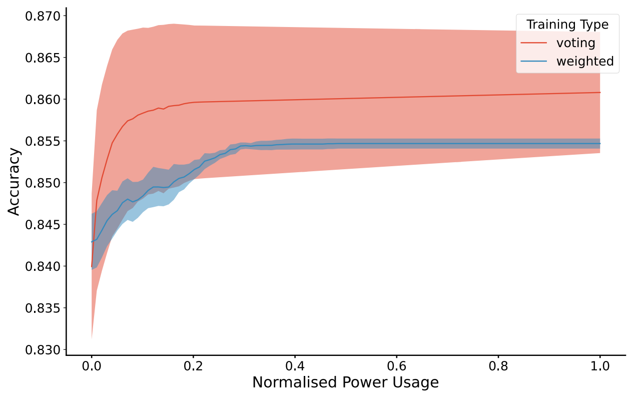


FIGURE 3.8: Operating range for ensemble classifiers, with both training regimes. Solid lines show the mean performance in their power range, the shaded area denotes the standard deviation across five runs.

AUC	
Voting	Weighted
0.860 ± 0.008	0.854 ± 0.001

TABLE 3.2: AUC measurements for both training techniques, mean and standard deviation are shown across 5 runs.

The progressive ensemble systems do present a good proof-of-concept for model-based progressive intelligence, and achieve the levels of accuracy seen in work by Frankle and Carbin (2018) using the same architecture. However more powerful architectures do reach a higher level accuracy, in an effort to move towards this we can use a more performant technique seen in the field and discussed in detail in section 2.3.2, branched neural networks.

3.3.2 Branched Neural Networks for Model-Based Progressive Intelligence

Branches can take a variety of forms, when first proposed by Teerapittayanon et al. (2016) the branches consisted of a number of layers including convolutional and pooling layers, which are discussed in the next chapter. This was echoed in following work, but we find that similar accuracies are achieved when simply using an average pooling layer and a fully connected (MLP) layer, thus reducing the operational cost of the branches, this is also shown in work by Sun et al. (2021).

It is recommended in the branch network literature (eg. Teerapittayanon et al. (2016), Hu et al. (2020)) to position the branches at equidistant points throughout the network, hence we follow this approach by placing the branches after each block of the network.

Adding branches to the network will give the optimisation task more degrees of freedom. This can still be represented simply, by summing the losses from each branch and applying weighting functions where necessary. The resultant total loss function L is shown in equation 3.13, as formalised by Teerapittayanon et al. (2016). The individual branch losses are shown in equation 3.14, which is the cross-entropy loss. However, there is scope to make these loss functions more sophisticated (Wolczyk et al., 2021; Sun et al., 2021).

$$\begin{aligned} L(y,\hat{y}) &= \sum_{b\in B} w_b\, L_{\mathrm{b}}(y_b,\hat{y}) && (3.13)\\ L_b(y_b,\hat{y}) &= -\sum_{i=1}^{K} y_{b,i}\, \log(\hat{y}_i) && (3.14) \end{aligned}$$

Here L_b refers to the loss function at any given exit, with respect to the prediction $\hat{y}_b$ and ground truth y, which is subject to the model parameters θ. These losses are then summed according to their weightings w_b, which should satisfy $\sum_{b=1}^{B} w_b = 1$.

In order to compare our approach with the ensemble based approach, we select a backbone that operates in a similar range to the ensemble. For this we can use the MobileNet architecture (Howard et al., 2019). We discuss the design of the MobileNet in section4.2.1. For now we will just discuss the inference process in the branched neural network and compare the operating range against our ensemble based approach.

Early exiting inference uses the intermediate output of the network to determine whether the classification is confident enough. This intermediate output will come from the branches which are co-optimised with the backbone, or can be trained independently to classify on the backbone network.

The general procedure for early exiting inference in this system is shown in algorithm 3, where each branch is evaluated sequentially on a given input; if the confidence requirement is reached on a given branch, the inference process can be halted. When the inference process is halted, the inference cost of subsequent layers is saved, making the network more computationally efficient on the test set.

Algorithm 3 Early exiting inference algorithm for branched neural networks

```
1:  procedure EARLY EXITING INFERENCE IN BRANCHED NETWORKS($f, X, \alpha$)
2:      Load model: $f$
3:      Read data, Threshold: $X, \alpha$
4:      for $x \in X$ do                                    ▷ For all inputs in the test set
5:          for branch $n \in N$ do                          ▷ For all branches in model
6:              $\hat{y}_n = \arg\max(f_n(x))$               ▷ Calculate branch prediction
7:              $\kappa_n = \text{confidence}(f_n(x))$        ▷ Calculate confidence
8:              if $\kappa_n \geq \alpha$ then                ▷ Check confidence threshold exceeded
9:                  return $\hat{y}_n$                        ▷ Return classification if confident
10:             end if
11:         end for                                         ▷ Full inference used without early exit
12:         $\kappa_N = \text{confidence}(f_N(x))$            ▷ Calculate final entropy for completeness
13:         $\hat{y}_N = \arg\max(f_N(x))$
14:         return $\hat{y}_N$                                ▷ Return final output
15:     end for                                             ▷ Move to next input
16: end procedure
```

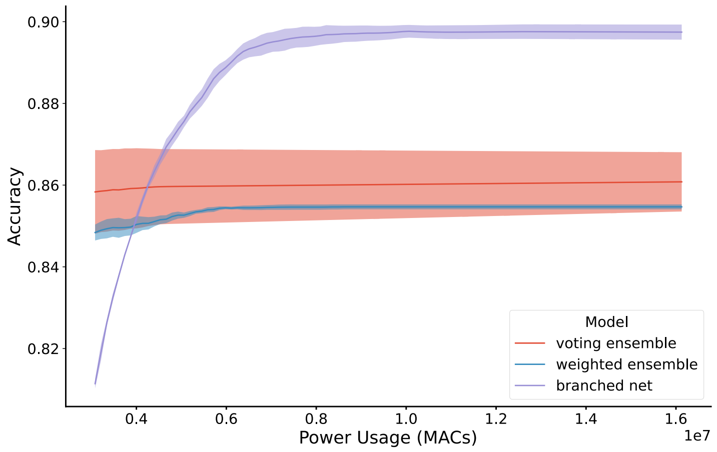


FIGURE 3.9: Operating range for ensemble classifiers, with both training regimes. Solid lines show the mean performance in their power range, the shaded area denotes the standard deviation across 5 runs.

Once again, as in equation 3.12, entropy is the basis for our confidence measure. We recreate this exit policy in algorithm 3, and vary the threshold α to present a branched model operating under varying confidence thresholds. Using models trained using the loss in equation 3.13 on the CIFAR10 dataset, we compare the operating range of a branched network against that of our ensemble approaches. This is shown in figure 3.9.

We find that the branched network, whilst performing slightly worse at the beginning of its operating range, quickly rises in performance as confidence and MACs are increased, achieving an accuracy of ~90% at ~25% power usage. The variance in the performance is also a lot lower in the branched network than the ensemble trained with a voting loss. Furthermore, the range in accuracy is more indicative of a progressive intelligence. That is, whilst the ensemble models do outperform the branched net at the lowest power range, their accuracy changes very little throughout the operating range, whereas the branched network improves significantly. To understand this further we examine the AUC measurements of all 3 methods in table 3.3.

AUC		
Voting ensemble	Weighted ensemble	Branched net
0.860 ± 0.008	0.854 ± 0.001	0.889 ± 0.002

TABLE 3.3: AUC measurements for both training techniques, mean and standard deviation are shown across 5 runs. Then we compare with the branched network, with mean and standard deviation shown across 3 runs.

We find that in the model space, the branched network outperforms the ensemble methods, including in AUC measurements. Hence, for the next chapter we will take this method of model-based progressive intelligence forward in order to understand it more deeply.

3.4 Summary

In this chapter we have explored progressive intelligence as a concept, and how it can be quantified. We have also introduced data and model-based systems, and found theoretical values of the expected error in the former. For model-based systems we have taken a more empirical approach, looking towards vision problems as a test-bed for progressive intelligence.

In the first section we find that when analysing the performance–cost space, maximising early gains in performance with early costs can be likened to the area under the curve (AUC). Hence, with the AUC we have a measure of performance that is generalisable to any measure of performance or cost, and assuming equal cost bounds, a measure of performance that can be used to compare two approaches.

The second section considers data-space progressive intelligence, taking a theoretical approach and considering the fundamentals of such a system, the number of samples. We find that the expected error should decrease according to $N^{-\frac{1}{2}}$. We find this relationship holds in practice when analysing a boundary on Gaussian distributions and when using real-world data we see it becomes an upper bound.

In the final section we move to model-based systems, taking a more pragmatic approach we move towards vision problems and neural networks. We use a CNN-based ensemble as a prototype progressive intelligence system and find performance benefits are limited when using them. More advanced losses may improve these systems, however, we choose to look at branched neural networks for model-based progressive intelligence. This is because we find these perform much better than the ensemble approach, likely due to the increased depth and complexity of the branched model. Hence, in the next section branched networks will be explored. In particular, how they are designed, trained, and used at inference time.

Chapter 4

Exploring Branched Networks for Progressive Intelligence

The previous chapter discussed the different types of progressive intelligence, data-based methods are effective in saving power, but we find more tangible performance benefits are seen in the larger machine learning models which are often computationally expensive to operate. Hence, it is better to consider model-based progressive intelligence, whereby the underlying model is adapted at runtime. In the previous chapter we found that branched neural networks not only perform well, but also offer a neat, single component solution to the problem of producing progressive intelligence. Hence, this chapter presents an exploration of progressive intelligence using branched networks. The majority of the work in this chapter has been published a conference paper (Dymond et al., 2022).

In section 2.2 we discussed a variety of neural network architectures, all of which could be applied to this research, and in the previous section we explored the MobileNet architecture due to its overlap in power range with our ensemble approach. However, no neural network architecture has been more successful or pervasive than the ResNet architecture (He et al., 2016). Its modular design makes it easy to experiment with in terms of depth, width, and branch positioning. Furthermore, its prevalence in the field ensures the findings will be applicable to a wider bredth of related works: they were first used in branched architectures by Teerapittayanon et al. (2017), and are still being used in recent work by Hu et al. (2020) and Wolczyk et al. (2021). Hence, this architecture will constitute the backbone in the experiments undertaken and will have branches applied to it.

In this chapter the training processes will be considered, and how it can affect properties relating to progressive intelligence. The next section will examine the importance of the loss function and how it can affect the pareto-optimisation of the branched network representations. The second section will discuss the importance of the backbone architecture and how it can influence the power consumption of the system. Finally, the last section of the chapter will discuss the exit policy, and what changes can be made to improve early exit power savings.

4.1 The Effect of the Loss Function

Branched neural networks are perhaps the simplest form of dynamic inference in neural networks, but they are arguably also the most complete solution to the progressive intelligence problem. This is because the architecture itself implicitly allows for the inference to take place incrementally and allows for the halting of inference at chosen points. As such, branched networks have consistently found their place in recent dynamic inference literature (Hu et al., 2020; Wolczyk et al., 2021; Sun et al., 2021; Tang et al., 2023).

However, branched networks have yet to be investigated from a representational perspective. Here, an insight into the representational benefits of neural networks is made, making the following contributions and addressing the gap in the field: *Representational Effect of Classification Branches* in section 2.4.

Pareto-Optimising the Representations of Branched Neural Networks Previous work has investigated the optimisation of branched neural networks, but at the time of writing, no work has taken a detailed investigation to the effect early exit branches have on intermediate representations when they are incorporated into the loss function. This work has coupled state-of-the-art explainability techniques with branched neural networks, allowing a greater insight into their effect. It has then used this insight to propose training regimes and identified one which allows the benefits of the branched network to persist, whilst retaining the performance of the backbone network. Furthermore, we investigate analytically the effect of branch positioning.

Investigating Different Loss Objectives in Branches Similarly, the effect of different training objectives have not been prevalent in the branched neural network space. This work has briefly introduced two objectives that may be relevant in the progressive intelligence paradigm and also investigated these objectives using explainability metrics. Future work in this space will better enable progressively intelligent deployed systems which have specialised roles, or which have a multi-modal approach to the inference problem in the case of hierarchical labelling, which is also investigated in chapter 5.

4.1.1 Techniques for Comparing Neural Network Representations

In order to understand the inner workings of the neural network it would be useful to know three things. One is how the data is separated in the latent space, as this allows an understanding of how the network separates and clusters the classes in the latent space. Next, linear separability can also be inferred which is different to separation, as the classes do not necessarily need to be separated from each other in the latent space for this to occur. This will directly indicate the performance in that area of the backbone. Lastly we wish to compare representations between different neural networks to understand the effect branches have on the way a model processes its input. As such it will be beneficial to have a similarity measure between the latent representations of different layers between different neural networks.

Class Separation An early insight to the inner representations of neural networks was provided using t-SNE projections, which project high dimensional vectors such as neural network embeddings, into two or three dimensional spaces (Van der Maaten and Hinton, 2008). The overlap of different classes when projected into this space allowed the discriminatory power of the network to be estimated. However, t-SNE projections have drawbacks, namely that representations between layers of different dimensionality cannot be compared using the same projection in this setting. Furthermore, due to the flexibility of the approach such projections can often be misleading, for example showing clusters in randomly sampled data (Wattenberg et al., 2016).

Motivated by the drawbacks of projection based approaches, Euclidean distances were proposed as a means of measuring class separability, in a measure which the authors call general discrimination value (GDV) (Schilling et al., 2018). The measure is defined as the difference between the mean intra-cluster variability and the mean inter-cluster separation for labelled vectors in the embedding space. Hence, a lower GDV corresponds to a greater separation of classes, it is also parameter free unlike projection-based approaches. Using this measure, the authors found that on simple problems the GDV would fall monotonically and on more challenging problems transformations are necessary in the early layers before class separation can occur in the later layers.

More recent work has used cosine similarity to measure class separability, where the authors use the ratio of intra-class cosine distance over the inter-class cosine distance and take one minus this quantity as their measure (Kornblith et al., 2020). The metric, R^2, is defined as

$$R^2 = 1 - \frac{\sum_{c=1}^{C} \sum_{m=1}^{N_c} \sum_{n=1}^{N_c} (1 - \mathrm{sim}(\boldsymbol{X}_{c,m}, \boldsymbol{X}_{c,n}))/(CN_c^2)}{\sum_{j=1}^{C} \sum_{c=1}^{C} \sum_{m=1}^{N_j} \sum_{n=1}^{N_c} (1 - \mathrm{sim}(\boldsymbol{X}_{j,m}, \boldsymbol{X}_{c,n}))/(CN_j N_c)}. \tag{4.1}$$

Here, $\boldsymbol{X}$ refers to the activation of an input, n and m refer to a datapoint within the class c and j. N_c and N_j refers to the number of samples within the class and C the total number of classes. Finally, sim(.,.) refers to the cosine similarity between the two activation vectors, which quantifies the similarity between the activations as they are passed through the model. By taking the ratio of the similarity within a class and that across all classes, the class separation can be quantified. The metric takes one minus this value to ensure a greater value signifies greater class separation. Similar to GDV, this cosine similarity score also revealed class separation is concentrated in the final layers of the network for more complex problems. Hence, the R^2 metric provides a suitable measure of class-separation.

Class Separability Perhaps the most appropriate way to measure class separation is to attempt classification on the embedding space itself using a linear classifier, the accuracy denoting the class separability. This was first proposed in a paper referring to these intermediate classifiers as *linear probes* (Alain and Bengio, 2017). Unlike other separability measures this work found class separability, the linear probe accuracy, would increase monotonically. They also found greater complexity in the classification problem would make this monotonic increase slower. This suggests that while the networks do not separate classes geometrically, they do learn a representation that allows discrimination between them when using simple linear classifiers. The authors also consider classification branches, which they refer to as "auxiliary classifiers". However, they restrict their analysis to how these assist final layer classification and do not asses their impact on the intermediate representations. Further work has investigated linear probe accuracy on intermediate layers, suggesting that a greater linear separability is indicative of, and important for, better generalisation performance (Belcher et al., 2020).

We can define the linear probe as the mapping between the activation features, $\mathbf{X}_l$, to a categorical distribution $[0,1]^C$: $\mathbf{X}_l \to [0,1]^C$. In practice, f_l takes the flattened embedding vector of that layer, x_l. Giving us

$$f_l = \sigma(W_l \cdot x_l + b_l). \tag{4.2}$$

Where $\mathbf{X}_l$ is of the form $\mathbb{R}^D$, with D denoting the dimensionality of the activation at that layer. These features are passed to the lth linear probe classifier with weight and bias terms W_l and b_l respectively which are of shape $D \times C$ and $1 \times C$ respectively. These parameters are optimsed to minimise the cross-entropy loss function. Finally, $[0,1]^C$ refers to the categorical distribution across the C target classes, to which the features x_l belong to.

Thus, to measure class separability we use linear classifiers applied to the embedding space. It is possible to use this metric to evaluate the linear probe accuracy on either the training data or the validation/testing data. To understand the generalisation performance, we focus our evaluation on the validation data.

Representational Analysis An effective method of visualising the similarity of representations between different network architectures is needed to compare activation distributions between the layers. Activations from a layer are preferred over the weights of the layer, as weight distributions do not necessarily reflect the effect a specific layer has on the input, whereas activations can provide some insight since they are input dependent.

A number of metrics have been proposed in the field, among which has been Canonical Correlation Analysis (CCA) which has taken a number of forms, for example, single vector CCA

(SVCCA) which alleviates the sensitivity of standard CCA to perturbations (Raghu et al., 2017). However, a considerable oversight with these approaches is that they often fail to pass a simple sanity check: "Given a pair of architecturally identical networks trained from different random initialisations, for each layer in the first network, the most similar layer in the second network should be the architecturally corresponding layer" (Kornblith et al., 2019).

Thus, motivated by the inadequacy of CCA, the same research group that proposed the R^2 metric presented Centred Kernel Alignment (CKA) (Kornblith et al., 2019). This is a method of analysing representational similarities between networks. CKA is essentially a pairwise comparison between the intermediate output of every layer of one neural network and every layer of another neural network. It is defined by

$$\mathrm{CKA}(\mathbf{X}_i, \mathbf{Y}_j) = \frac{\mathrm{HSIC}(\mathbf{X}_i^\top \mathbf{X}_i, \mathbf{Y}_j^\top \mathbf{Y}_j)}{\sqrt{\mathrm{HSIC}(\mathbf{X}_i^\top \mathbf{X}_i, \mathbf{X}_i^\top \mathbf{X}_i)}\sqrt{\mathrm{HSIC}(\mathbf{Y}_j^\top \mathbf{Y}_j, \mathbf{Y}_j^\top \mathbf{Y}_j)}}, \tag{4.3}$$

where $\mathbf{X}_i$ and $\mathbf{Y}_j$ are the activations from layers i and j respectively. HSIC refers to the Hilbert-Schmid-Independence-Criterion (HSIC) (Song et al., 2012). The HSIC measures the statistical dependence between two distributions, which in this work will be the activations of the model. Let $\mathbf{K} = k(\mathbf{X}_i, \mathbf{X}_i)$ and $\mathbf{L} = l(\mathbf{Y}_j, \mathbf{Y}_j)$ where k and l are two kernels. The HSIC is defined by

$$\mathrm{HSIC}(\mathbf{K}, \mathbf{L}) = \frac{1}{(n-1)^2} \mathrm{trace}(\mathbf{K}\,\mathbf{H}\,\mathbf{L}\,\mathbf{H}), \tag{4.4}$$

where $\mathbf{H}$ is the centering matrix defined as $\mathbf{H}_i = \mathbf{I}_i - \frac{1}{i}\mathbf{1}\mathbf{1}^\top$ which centers the two matrices in kernel space. The HSIC calculates the inner product between all the elements in these two centered projections. Thus, the CKA sums the statistical dependence between the individual activations and normalises the value. This allows the similarity between activations produced by two models to be compared. In this work, we use a linear kernel as shown to be effective in the original work by (Kornblith et al., 2019).

This method was later used to analyse the effect of width and depth and how it changes what neural networks learn, by comparing *self-similarity* of such networks (Nguyen et al., 2021). They found that the wider or deeper a neural network and thus the more overparameterised it is, the *blockier* the representations in the the CKA plot would become. That is, once overparameterised, any given layer would produce representations that are similar to the layers around it. This can be qualitatively explained as it being beneficial for network to maintain the transformations it has already applied, as further transformation of the input would adversely affect model performance.

This same metric was also used to analyse different loss functions, finding that in the later layers closer to the classification layers, network representations would begin to differ depending on their training objective (Kornblith et al., 2020). Furthermore, recent work has extended this analysis to vision transformers, to compare the representations between them and CNNs (Raghu et al., 2021). Hence, it is a prevalent metric in the field and will be an effective measure of representational similarity when comparing networks.

A number of publications also show equivalence between the R^2 metric and CKA (Kornblith et al., 2020; Cristianini et al., 2002; Cortes et al., 2012). Namely, that when using a balanced dataset and a cosine kernel, the CKA score between the embeddings of a layer and a one-hot encoded label matrix, is equivalent to the R^2 score in the same layer.

This presents us with three methods which will be of use in this work: we measure class-separation with the R^2 metric, model similarity with CKA score, and class separability, or generalisability, using linear probes.

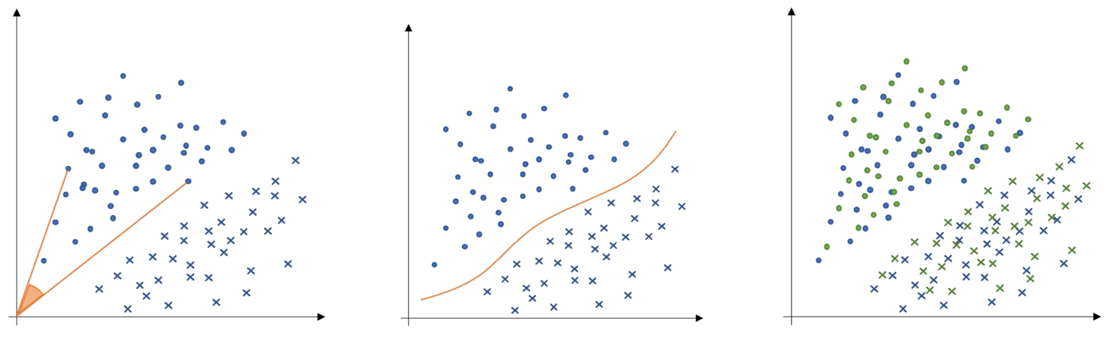

(a) Cosine Similarity (b) Linear Probe (c) CKA

FIGURE 4.1: A comparison between the different techniques presented. Here a hypothetical two dimensional latent space is presented, with two different classes shown with circles and crosses. Cosine similarity from R^2 between two data points is shown in (a), a linear probe boundary in (b), and two latent space distributions for CKA is shown in (c).

For these metrics the activations will be taken from all layers. A visual comparison between these metrics is presented in figure 4.1. The figure 4.1 (a) shows the cosine similarity between two data points, these similarity measures can be calculated and summed between all data points in the latent space to understand class separation at each intermediate layer. In 4.1 (b) a classification boundary as drawn by a linear probe is shown in the latent space, separating the classes in the latent space. Finally, in 4.1 (c) two distributions are shown for different networks at any given layer, one in blue and one in green, the CKA score is then a measure of statistical dependence between these two distributions.

The field has taken other directions in the pursuit of understanding neural networks, however these are not the main focus of this work. A richer survey in this area investigating our understanding of the inner workings of neural networks has recently been written by Räuker et al. (2023).

Confusion Matrices Finally, confusion matrices are a common technique which allow for the cross-examination of classification performance, through the statistics of mis-classification. This is achieved by plotting the actual class along the rows of a table and the predicted class along its columns. The resultant matrix is shown below in table 4.1, it follows that the diagonal of the matrix is populated with the correct predictions, assuming the class order is preserved between the two axes.

Actual \ Predicted	Cat	Dog
Cat	**6**	4
Dog	2	**8**

TABLE 4.1: Confusion matrix between a cat and a dog.

Three metrics have been presented which will allow the class separation, class separability, and representational similarity to be compared across architectures. We will also use confusion matrices, a common method of cross-examining errors in classification tasks. Now the experimental procedure is addressed.

4.1.2 Opening the Black-Box

We wish to understand the effects of branches in the neural network and we do this by exploring their weighting and placement. Following this, we explore how the training strategies and optimisation objectives in the branches can be varied. An important aspect is to compare the representational effect of including the branches in the loss function during training with when the branches are added post training. This section will detail and justify the experiments undertaken in this work.

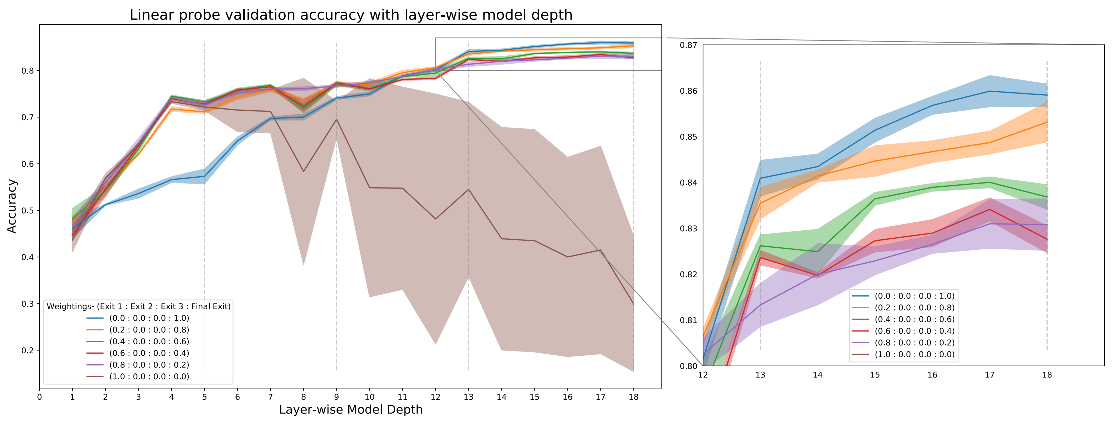
4.1.2.1 Varied branch weighting

First the effects of varying branch weightings are analysed using class separation and linear probe accuracy. This is the simplest way to change the outcome of a branched system. That is, to vary the parameter w_n in equation 3.13

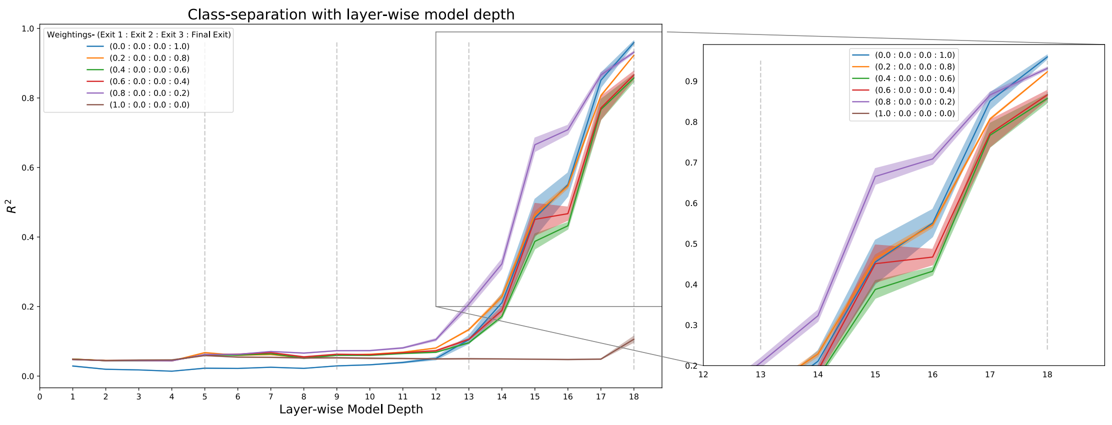
$$L_{total}(\hat{y}, y; \theta) = \sum_{b=1}^{B} w_b L_b(\hat{y}_b, y; \theta),$$

where the weights should satisfy the following condition $\sum_{b=1}^{B} w_b = 1$. When using a single branched exit we vary the weighting of the first branch from 0.0 to 1.0, adjusting the second exit accordingly. When using multiple branches we keep one branch at a fixed weighting, vary a second branch in the system, and adjust the final exit according to the summation condition. This allows us to sample from the partition of unity and gain an understanding of how the behaviour can be changed with branch positions and weightings. The optimisation of branch weightings in the current literature has focused on output accuracies as opposed to maximising representational objectives such as class separation and separability. The goal of this experiment is to pareto-optimise the intermediate layer performance of the network according to linear probe accuracy, we will use equation 3.2 to help quantify this. It is when looking at progressive intelligence that these weightings show more dramatic changes and by varying them we can aim to pareto-optimise the intermediate layer performance.

To investigate the effects of branch weighting we consider a single interior branch at layer 5, and vary its weighting between 0 and 1. Linear probe results can be found in figure 4.2 (a) and class separation results can be found in figure 4.2 (b) using the R^2 metric defined in equation 4.1.



(a) Linear probe accuracy



(b) Class-Separation

FIGURE 4.2: Layer-wise metrics with varying branch weightings, linear probe accuracy in (a) and class separation using the R^2 metric in (b). The unbranched network is shown in blue, with 100% of the loss weighting being allocated to the final exit. The grey dashed lines here indicate the exit positions in the network, including the final exit at layer 18. The shaded regions denote the standard deviation across 3 runs.

From figure 4.2 (a) it is evident that the branches have a significant effect on linear probe accuracy and thus class separability throughout the backbone, when they are incorporated in the training of the network. This is demonstrated in the layers preceding the early exit, where there is a ~20% increase in linear probe accuracy between the unbranched model, shown in blue, and the branched models. Furthermore, this increased separability is predominantly maintained after the early exiting, suggesting the transformations induced by the early exiting objective are beneficial to the final classification layer. However, as is shown in the right-hand element of the plots, this earlier separation is at the expense of slightly reduced class separation at the classification layer. This suggests that early exit objectives increases class separability earlier in the layers at the slight expense of reduced separability later in the network.

It is also clear that these effects become more pronounced with branch weighting. That is, a greater weighting increases the class separability earlier in the network, but slightly reduces it at the final classification layer. A weighting as low as 0.2 is enough to see the positive effects of the branch earlier in the network, with only very slightly reduced separability later in the network.

The brown line in this graph denotes a branch with a weighting of 1.0, meaning the later layers remain at their initialisation values. In this system the linear probe accuracy increases quickly and lowers slowly following the exit, on average falling to a minimum accuracy of ~30% at the final exit. This suggests that the class separability incurred by the transformations made by the model is resistant to the random convolutions following them. Indeed literature has investigated the effectiveness of random weights in the network and found that only select layers in a convolutional network need to be trained to achieve reasonable results and that random convolutions do not negate the work of the model (Rosenfeld and Tsotsos, 2019). Evidently, this observation holds true to some extent when using branches in the neural network.

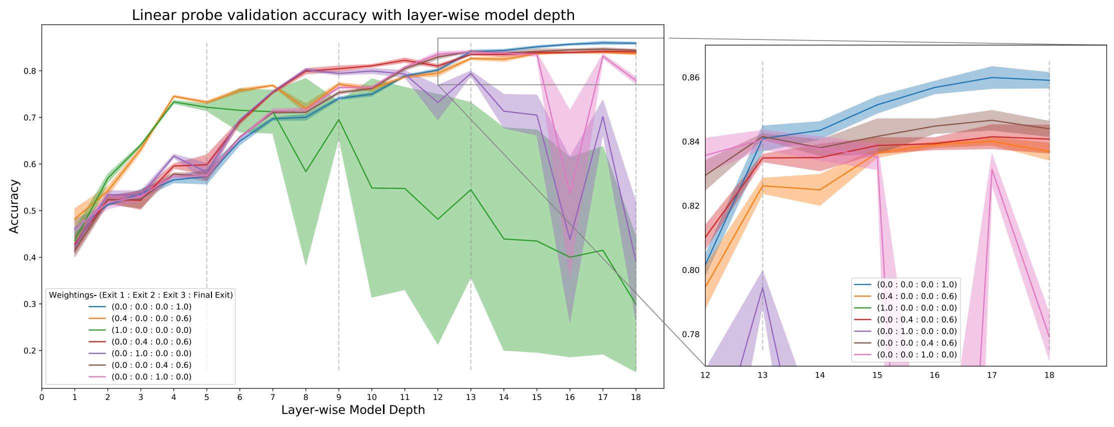
From figure 4.2 (b) it is clear that branches earlier in the network have a less significant effect on actual class separation than they do on class separability. However, there is a small increase throughout the layers when the branch is introduced, and the late increase in class separation occurs earlier in the branched networks, this happening earlier with increased branch weighting. Much like in the linear probe accuracies, the maximum class separation achieved is dependent on the branch weighting, with lower branch weightings often corresponding to greater class separation at the final layer. Although this is not true with the 0.8 weighted branch, the purple line, which matches the performance of the 0.2 weighted branch at the final layer.

Once again, the brown line in this plot indicates a network with a branch weighting of 1.0. It is clear from the very slow drop in class separation that the random convolutions following the branch have little effect on class separation. The sharp rise in class separation coincides with a compression of the embedding space, likely causing classes to cluster slightly more within the layer. As it is evident from the linear probe plot that it is still possible to classify in these embedding spaces, with greater than random levels of accuracy. Furthermore, the variance seen in this line is much less than the equivalent in the linear probe plot.

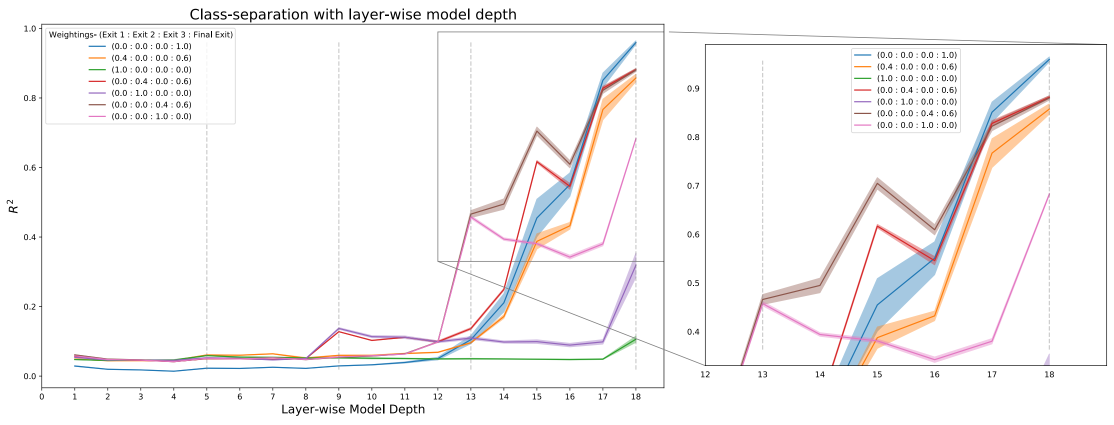
4.1.2.2 Varied branch positioning

Following the varied branch weighting experiment, the branch positions are varied to understand their effect on the separation of classes. The literature often places branches in an *ad-hoc* fashion, giving qualitative explanations for their positioning. For example, spacing branches in an equidistant manner has been proven to be successful in efficient classification (Teerapittayanon et al., 2016; Hu et al., 2020).

Hence, we follow the recent branched network literature and position our branches at architecturally convenient points, such as between blocks which are distributed evenly throughout the network. This experiment aims to understand how the intermediate layers are affected when using branches and to build pareto-optimality into the system.



(a) Linear probe accuracy



(b) Class-Separation

FIGURE 4.3: Layer-wise metrics with varying branch weightings and positioning, linear probe accuracy in (a) and class separation using the R^2 metric in (b). The unbranched network is shown in blue, with 100% of the loss weighting being allocated to the final exit. The grey dashed lines here indicate the exit positions in the network, including the final exit at layer 18. The shaded regions denoting the standard deviation across 3 runs.

The branches were positioned between the blocks in the backbone architecture design, meaning there were three possible positions at which inference could be halted early. Here the analysis of the weight varying experiment is repeated, we select a weighting of 0.4 and 1.0 at each branch and compare the impact of branch positioning on linear probe accuracy in figure 4.3 (a) and class separation in figure 4.3 (b).

From figure 4.3 (a) it is evident that the branches have diminishing returns on class separability the later they are incorporated into the network. With branch 2 providing only ~10% improvement to linear probe accuracy in the layers preceding it. Furthermore, the third branch only provides an additional ~5% linear probe accuracy.

We find similar effects across the fully weighted branches at each position, that is a gradual decrease in the linear probe accuracy is seen. The closer the branch to the final exit, the less this decrease. Interestingly in the randomly initialised layers preceding the fully weighted branches, there appears to be a drop in linear probe accuracy in the second to last layer of each block. Since the number of channels is consistent, this drop in accuracy corresponds to the random transformation of the representation space due to the convolution, which is then amended by the re-introduction of the more effective activations from the layer preceding the skipped connection. Also, when comparing these fully weighted branches, we find that variance in linear probe accuracy decreases in the unweighted regions, the closer the branch is to the final exit.

Finally, we see that in the third branch position there is little effect on linear probe accuracy. Furthermore, the fully weighted branch performs surprisingly well after the branch, excluding the drop in the penultimate block layer.

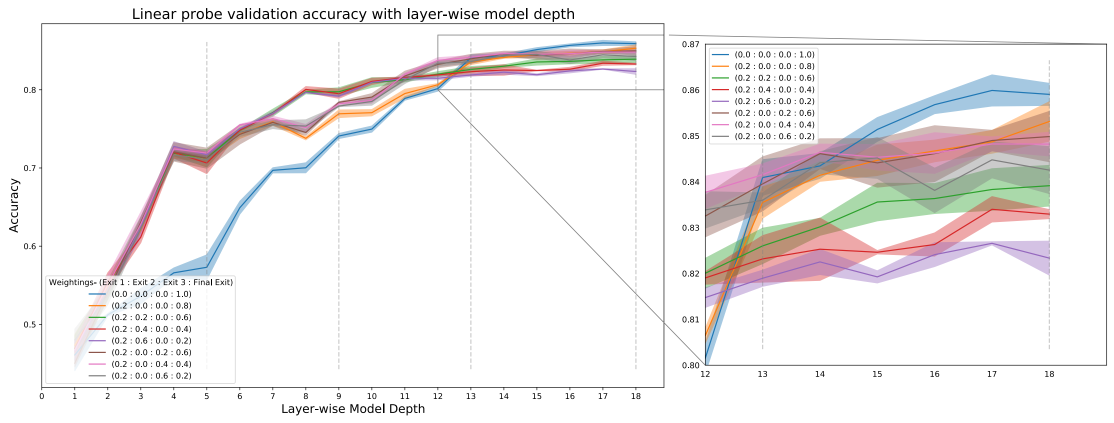
From figure 4.3 (b), the opposite conclusions can be drawn: The change in class separation is increased the closer the branch is to the final exit. There is little change between the different branch weightings in the class separation values at the final exit. However, when the final exit is unweighted, the final exit class separation appears to increase with branch proximity to the final exit. This is likely due to these representations encountering fewer random convolutions, thus having less effect on the class separation that has already taken place. Furthermore, in all of the fully weighted branches there is a jump in class separation at the final exit. This could be due to the compressed representation space caused by the average pooling layer, moving similar classes closer to one another and emphasising the class separation that took place in the early exit.

From these experiments, it is clear that the branches have a positive effect on the class separability and class separation, wherever they are placed.

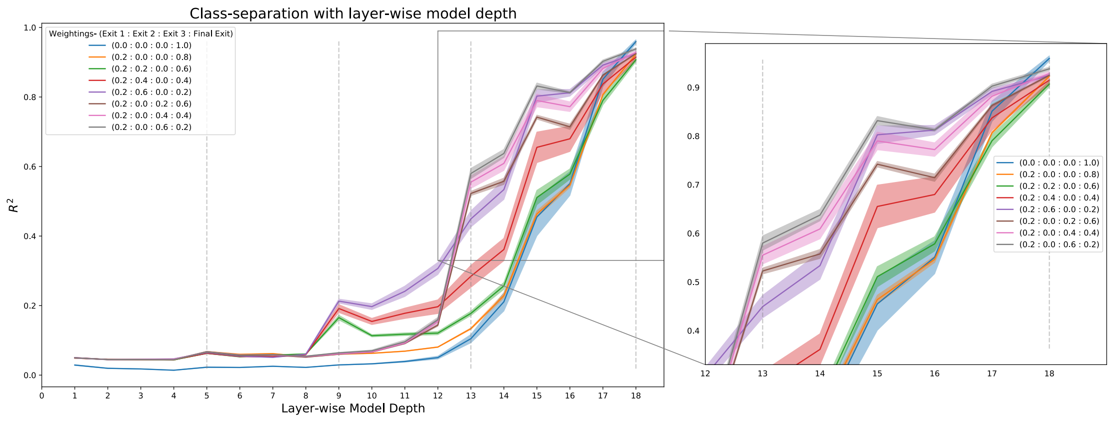
4.1.2.3 Branch combinations

With each branch affecting different layers of the network and having diminished effect elsewhere, the question is raised as to whether combinations of branches might be more beneficial. Indeed, when branches have been used in the field, they have seldom been used in isolation. Hence, we will discuss the effect combinations of branches have on these metrics.

Since the minimum weighting needed to see the benefits of the first branch is 0.2, we fix the first branch at this weighting and vary those of the others. First we look at combinations of just two branches, linear probe accuracy being shown in figure 4.4 (a) and class separation in figure 4.4 (b).



(a) Linear probe accuracy



(b) Class-Separation

FIGURE 4.4: Layer-wise metrics with varying branch weightings and positioning with multiple branches, linear probe accuracy in (a) and class separation using the R^2 metric in (b). The unbranched network is shown in blue, with 100% of the loss weighting being allocated to the final exit. The grey dashed lines here indicate the exit positions in the network, including the final exit at layer 18. The shaded regions denotie the standard deviation across 3 runs.

In figure 4.4 (a) we see that the combination of multiple branches further increases the improvements on linear probe accuracy earlier in the network, without diminishing the improvement of the previous branch. This is evident from the improvements over the orange line in the plot, which is the single weighted comparison. However, it is also clear that adding a second branch, namely in the second position, has further negative effects on final layer linear probe accuracy. As in figure 4.3 (a), we find the branch in the third position has less of an effect in linear probe accuracy in the layers preceding it, however, we do find that the final layer linear probe accuracy matches that of the single branch network.

Figure 4.4 (a) shows similar results to figure 4.4 (b) in that the benefits of the branches are combined when used in conjunction, as opposed to being diminished by competing objectives. Furthermore, all of the networks here have very similar final layer class separation values.

Finally we test combinations of three branches, in these ranges there is little change, both in linear probe accuracy and in class separation. However, it appears that equally weighting all three branches is marginally better, although the improvements are often within the variance of the other two combinations. Next, different training regimes will be discussed, analysing the effect of altering weightings during the training progress.

4.1.2.4 Scheduling Regimes

The previous sections have analysed the effect of branch weighting and position as well as the effects of combining multiple branches in the network. Optimising branched structures has seldom been studied as an adaptive learning system on the branches, although recent literature has used adaptive weights to optimise branched structures such as taking the inverse of the branch loss and using it as a weighting (Hu et al., 2019). Other work in the field thresholds the loss contributions from the branches and decreases this as training continues (Kaya et al., 2019). So there has been work in the field varying the training regimes, however the metric of success in this body of work has often been the accuracy of the branches. In this work we look beyond accuracy and focus on the class separability at various stages, aiming to pareto-optimise this property in the intermediate layers.

Due to the branches, there is a multi-objective optimisation problem, which presents challenges when trying to optimise the class separability throughout the network. Hence, we wish to vary the training regimes and identify the approach that best optimises class separability and class separation throughout the network. In doing so we present multiple regimes: a simple one in which the branches are optimised jointly with the final exit throughout training, as used in the previous sections; another scheme adds these exits onto a pre-trained backbone, and jointly optimises the branches along with the backbone, after the backbone architecture has been optimised independently, which we term *pre-trained*; finally, we present a finetuning scheme, where the branches and final exit are jointly optimised and then the final exit is fine-tuned to preserve final exit accuracy, which here is called *fine-tuned*.

The intuition of the pre-trained method is to understand if any pre-trained architecture can be taken and have the same representational benefits applied through attaching branches, also it is useful to understand if the final layer accuracy persists. In the finetuning experiment we want to understand the opposite. That is, in an architecture that has the branches co-optimised, is it possible to boost the final layer performance using an additional training step. Both methods use an additional training step and aim to reproduce the benefits of the branched network, whilst maintaining the final output performance of the unbranched network. We show linear probe results for these experiments in figure 4.5.

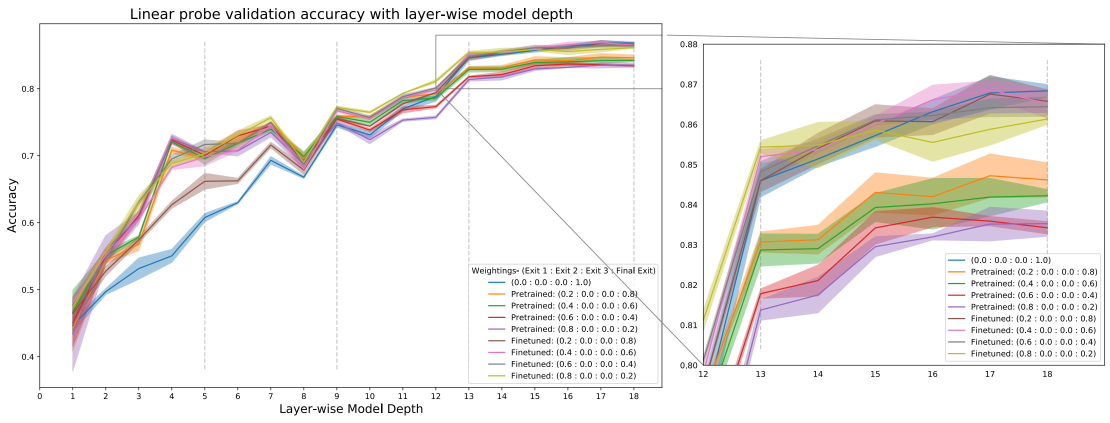


FIGURE 4.5: Layer-wise linear probe accuracy with varying training regimes. Here the unbranched network is shown in blue, with 100% of the loss weighting being allocated to the final exit. The grey dashed lines here indicate the exit positions in the network, including the final exit at layer 18. The shaded regions denote the standard deviation in results across 3 runs.

It is clear that all training regimes improve on the unbranched network. The pre-trained networks appear to improve over the fine-tuned networks in the earlier layers. However, later in the network the fine-tuned networks outperform the pre-trained networks. This make sense, as in each of these networks the layers at which one outperforms the other are the layers which were most recently optimised. The fine-tuned networks match the performance of the unbranched model in the final layer linear probe accuracy and improve upon it throughout the entire network. Here we compare against an unbranched model trained for the same total number of epochs as both scheduling regimes, 400 epochs. This suggests that optimising the branches first and then fine-tuning the final exit is the most effective way of learning the representations in a branched network. However, the lower the weighting in the fine-tuned networks the more the early benefits are removed, hence, stronger weighted branches are required at the earlier stages of training.

4.1.2.5 Varying Training objectives

Finally, at the time of writing, the effect of different training objectives between branches has yet to be explored in the field. To that end, we vary the training objectives in the branches and present two schemes that are concordant with progressive intelligence ideas. This is a novel investigation in the field of branched neural networks, particularly when analysed from the perspective on intermediate layer performance, looking to maximise the progressive intelligence qualities of a system. This is most relevant to the first research question, looking to understand progressive intelligence systems.

Firstly, a coarse-grained objective is used in the branches which incorporates hierarchical labels and the fine-grained labels for the final exit. Some datasets such as CIFAR100 have coarse-grained labels and fine-grained labels, however they are not incorporated into solutions as frequently as the fine-grained labels. Hence, one experiment in this work uses the coarse-grained labels in the early-exit branch and uses the fine-grained labels in the final branch. This aims to understand if the coarse-grained classification task earlier in the network can assist the fine-grained classification task.

The second objective used in the weighted loss function which focuses on specific classes in the dataset which are more pertinent for downstream tasks. In some application scenarios such as object detection, certain objects in the task space may be more important than others. This scenario is considered in chapter 5. Therefore, weighting operationally pertinent classes earlier in the network may better enable the confident classification of these objects and also allow them to be classified with lower latency. Here the classes weighted are *Plane, Truck, Automobile*, and *Ship*, which are given a weighting double that of the other classes in the dataset.

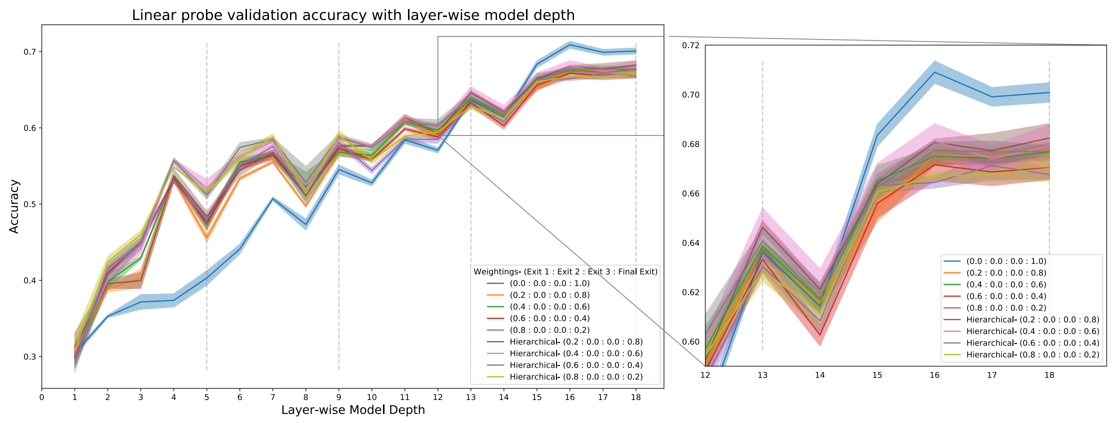
Hence, we have a number of experiments which will both investigate the effect branches have on neural network representations and their applicability to the progressive intelligence problems and their nuances. We also have a simple measure of performance for these experiments, as well as any qualitative results. Here, two objectives are analysed.

To analyse hierarchical classes, two linear probe plots are produced. One which tracks accuracy on the coarse-grained classes shown in figure 4.6 (a) and another which tracks the fine-grained accuracy in figure 4.6 (b).

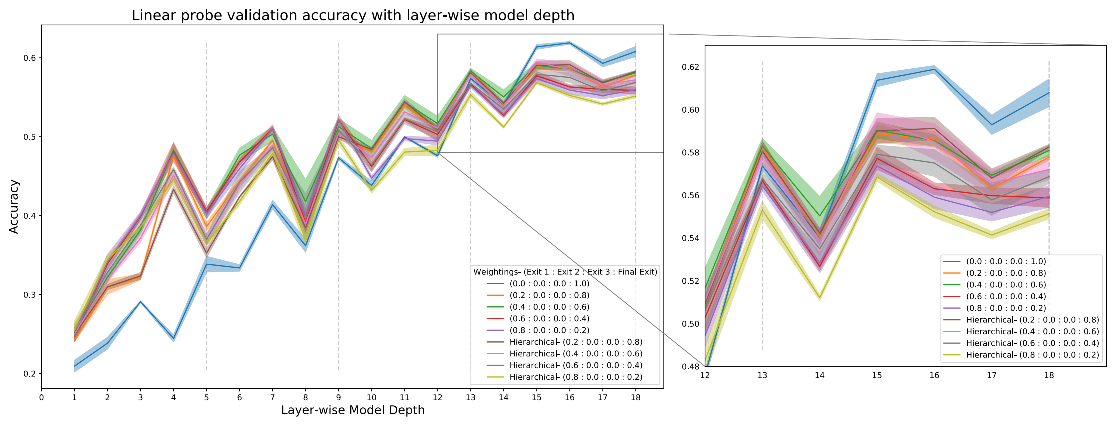
Figure 4.6 (a) shows a ~20% improvement on the unbranched model early in the network and the hierarchically trained models perform better than the the models trained using the fine labels. This is to be expected since the branches trained using the fine labels have no access to the coarse-grained labels, hence the hierarchically trained model will naturally perform better by this metric. However, at the final exit, all models perform worse than the unbranched network.

In figure 4.6 (b) the opposite is true as the model trained using the fine-labels outperforms the hierarchically trained model. However, both models still outperform the unbranched model on the fine-grained labels. At the final exit the hierarchically trained model performs worse than the fine-grained model and both training objectives perform worse than the unbranched model.

These plots also allow the performance of the branched ResNet18 to be assessed on the CIFAR100 dataset, shown in the non-hierarchical models in figures 4.6 (a) and (b). The branched networks still present an improvement over their unbranched counterpart, in some places an improvement of ~20%.



(a) Coarse-grained accuracy

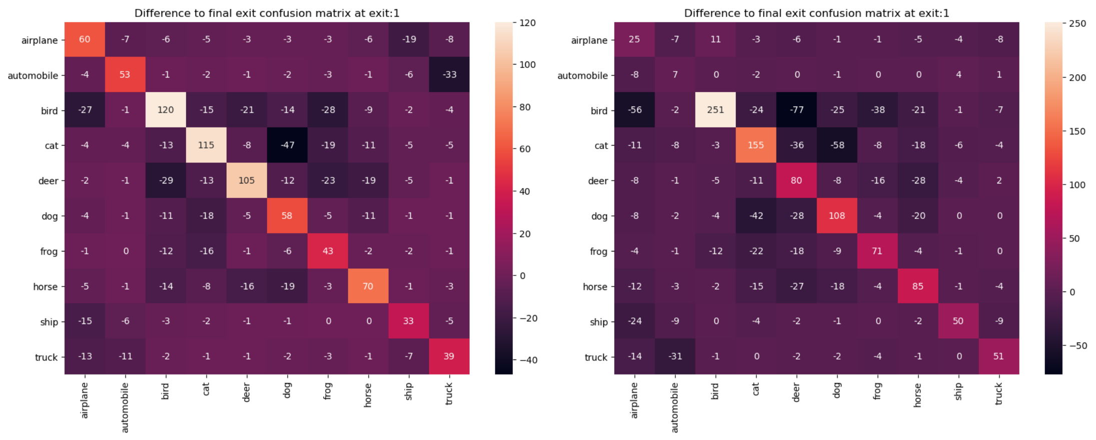


(b) Fine-grained accuracy

FIGURE 4.6: Layer-wise metrics with varying branch weightings and positioning with multiple branches, linear probe accuracy in (a) and class separation using the R^2 metric in (b). The unbranched network is shown in blue, with 100% of the loss weighting being allocated to the final exit. The grey dashed lines here indicate the exit positions in the network, including the final exit at layer 18. The shaded regions denoting the standard deviation across 3 runs.

The linear probe accuracy in general appears to be much lower on the CIFAR100 dataset, this is likely due to the increased class number and thus increased complexity of the task. We also find there is increased volatility in the linear probe accuracy, when shifting to the more complex dataset. The structure of the ReNet18 can explain volatility in the linear probe plots, in that the periodic behaviour in the plots happens every 2 layers which is the number of layers in single resnet *block*. This means, the increase in linear probe accuracy occurs at the concatenation of embeddings before the branch and after the branch this increase occurs between concatenations.

Finally weighted losses are implemented to prioritise specific classes in the task, here confusion matrices can be used to analyse the effect on individual class accuracy. To better visualise the difference between the branch and the final exit, plotted is the difference in confusion matrices. Therefore, positive numbers on the diagonal indicate worse performance, as do negative numbers off the diagonal. In figure 4.7 (a) the confusion matrix using the unweighted loss function is presented and in figure 4.7 (b) that of the weighted loss function is presented. Both networks are plotted used a branch weighting of 0.2.



(a) Unweighted (b) Weighted

FIGURE 4.7: Difference between confusion matrices in the first and final exit for a class-wise weighting in the classification branches. The unweighted model is shown in (a) the weighted in (b).

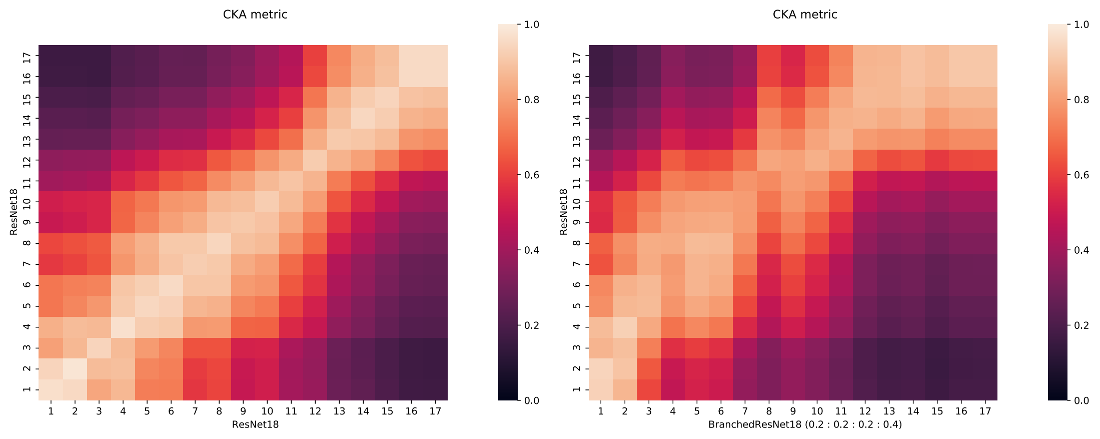
As expected the weighted classes are classified more easily earlier in the network, in some cases performing nearly as well as the final exit, such as under the *automobile* class. This is significantly better than the network without class weighting in some classes, however in others the network seemed to outperform the class-weighted model. Something about the dataset may inherently make these classes easier to classify and it is possible the branches obscure this. Finally, in some scenarios the branch in the class-weighted model outperformed the final exit, for example in classifying the *airplane* class as the *bird* class. This pattern continues with stronger branch weighting however there was little improvement over the *ship* and *truck* classes, suggesting high performance on these particular instances requires more representational power than what is available in the first few layers of the network.

4.1.2.6 What are the networks learning?

Linear probe accuracy, class separation plots, and confusion matrices can only go so far in understanding the inner workings of these networks, another method that can illuminate some of the representational changes are centred kernel alignment plots. Hence, these will be presented and used to analyse the similarities and differences between what the networks are learning.

First as a sanity check, two ResNet18s are compared against one another to ensure similarity is maximised in architecturally identical layers. This is presented in figure 4.8 (a), with the x and y axes denoting the layer of the network and the cell in the heat-map denoting the centred kernel alignment score between those layers. This similarity is maximised along the diagonal of the matrix with pockets of self-similarity evident within the blocks of the networks. The similarity between the first and final layers of each network are minimised, due to learning different representations.

Figure 4.8 (b) shows the CKA scores between layers of an unbranched network and a branched network. A three branched network selected as this appeared to be most successful network in optimising class separation throughout the network.



(a) Control (b) Unbranched vs Branched

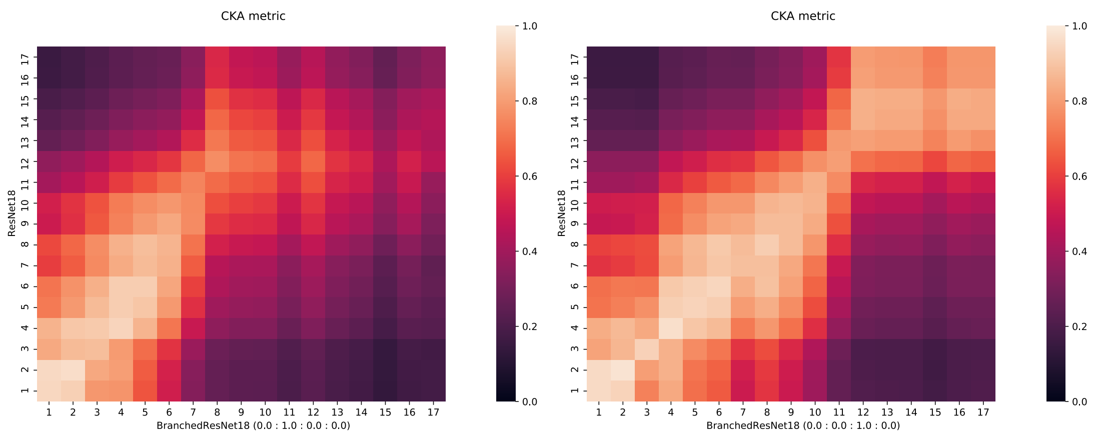
FIGURE 4.8: CKA scores between layers of two models. Two ResNet18s trained from different intialisations in (a) and a ResNet18 against a three branched counterpart in (b).

From the CKA plot, it is clear the similarity between the two networks never reaches the maximum values, peaking at ~0.95 in the early layers, throughout the rest of the network the maximum similarity at a given layer lies between 0.8 and 0.9. In the branch positions there is a clear shift in the similarities of the earlier layers of the branched networks towards the later layers of the unbranched networks. These are particularly noticeable at the 4th and 8th layers, the layers preceding the first and second exits respectively.

The shift in representational similarity to the later layers of the unbranched networks is not isolated to the branch layers, although it is more prevalent in them. This is evident from the departure from the diagonal and suggests that the branched model is learning the same representations of the data to those in the later layers of the unbranched model.

This distance of the shift from the diagonal is dependent on the depth at which the early exit takes place. This is highlighted in figure 4.9 (a) which shows the same plot for a fully singly weighted branched network, with its branch in the second position. The CKA score is maximised along a diagonal towards the exit position in the 9th layer, but it is off of the main diagonal of the heat-map and shares the most similarity in the 12th layer.

In figure 4.9 (b) this pattern is closer to the diagonal of the heatmap. Here we also see the CKA invokes a response from the layers after the exit, which are untrained and never updated. This is seen most clearly in 4.9 (b) where the branch is seen in the third position, but also



(a) Second branch fully weighted (b) Third branch fully weighted

FIGURE 4.9: CKA scores between layers of a ResNet18 and a single weighted branched counterpart, (a) shows a model with weighting in the second output position, and (b) a model with its weighting in the second to last position.

noticeable in figure 4.9 (a). This can explain the consistent linear probe accuracy, and class separation scores after the branches, as the representations after the exit are similar to those in the final layers of the unbranched model, meaning the activations will be effective. This is indicated by the block of high similarity in the top right corner of figure 4.9 (b).

4.1.3 Using Area Under Curve to Compare Models

We utilise the formulations in equations 3.1 to 3.2 and take the area under normalised linear probe accuracy curves. This is shown in table 4.2, where area under curve (AUC), raw improvement, and fractional improvement is presented.

It is clear from table 4.2 that weighting the early branches is most effective when using a single branch. As these branches increase performance early in the network where there are larger gains to be made. Hence the area under these curves is improved the most, up to 5.7%. When branches are positioned in the third position there is very little improvement over the baseline model, which is reflected in improvements that are less than 1%. There is very little difference in the AUC measurements for each branch position, suggesting that branch positioning has a greater effect than weighting when using this metric.

Below in table 4.3 the same results are presented for pre-trained and fine-tuned models when using the most effective branch position. Then multiple branch configurations are presented in table 4.4.

From these results it is clear that the fine-tuning process is more effective with the more heavily weighted branches, however it fails to outperform the models trained with the same weightings in table 4.2, according to the AUC metric. However, from the linear probe plots it is clear that these models perform better in final layer classification, thus highlighting a

w - (Exit 1 : Exit 2 : Exit 3 : Final Exit)	AUC	Raw ↑ (%)	Frac. ↑ (%)
Baseline: (0.0 : 0.0 : 0.0 : 1.0)	0.6878	-	-
(0.2 : 0.0 : 0.0 : 0.8)	**0.7276**	**3.98**	**5.78**
(0.4 : 0.0 : 0.0 : 0.6)	0.7266	3.88	5.64
(0.6 : 0.0 : 0.0 : 0.4)	0.7217	3.39	4.93
(0.8 : 0.0 : 0.0 : 0.2)	0.7262	3.84	5.58
(0.0 : 0.2 : 0.0 : 0.8)	0.7096	2.18	3.16
(0.0 : 0.4 : 0.0 : 0.6)	0.7093	2.15	3.13
(0.0 : 0.6 : 0.0 : 0.4)	0.7099	2.21	3.21
(0.0 : 0.8 : 0.0 : 0.2)	0.7078	2.0	2.91
(0.0 : 0.0 : 0.2 : 0.8)	0.6926	0.48	0.69
(0.0 : 0.0 : 0.4 : 0.6)	0.6932	0.54	0.78
(0.0 : 0.0 : 0.6 : 0.4)	0.6919	0.41	0.6
(0.0 : 0.0 : 0.8 : 0.2)	0.6937	0.59	0.86

TABLE 4.2: A table of area under curve (AUC) measurements of single weighted linear probe accuracy plots. Branch weighting is shown in the first column, raw AUC shown in the second, the third and fourth show raw and fractional improvement upon the baseline respectively.

drawback to this evaluation metric. Finally we present multiple weighted models in table 4.4.

w - (Exit 1 : Exit 2 : Exit 3 : Final Exit)	AUC	Raw ↑ (%)	Frac. ↑ (%)
Baseline: (0.0 : 0.0 : 0.0 : 1.0)	0.6878	-	-
Pretrained: (0.2 : 0.0 : 0.0 : 0.8)	0.7133	2.55	3.71
Pretrained: (0.4 : 0.0 : 0.0 : 0.6)	0.7137	2.59	3.77
Pretrained: (0.6 : 0.0 : 0.0 : 0.4)	0.7111	2.33	3.39
Pretrained: (0.8 : 0.0 : 0.0 : 0.2)	0.705	1.72	2.5
Finetuned: (0.2 : 0.0 : 0.0 : 0.8)	0.7059	1.81	2.64
Finetuned: (0.4 : 0.0 : 0.0 : 0.6)	0.7218	3.4	4.94
Finetuned: (0.6 : 0.0 : 0.0 : 0.4)	0.7255	3.77	5.48
Finetuned: (0.8 : 0.0 : 0.0 : 0.2)	**0.7269**	**3.91**	**5.69**

TABLE 4.3: A table of area under curve (AUC) measurements of linear probe accuracy plots with pre-tained and fine-tuned models. Training type and branch weighting is shown in the first column, raw AUC shown in the second, the third and fourth show raw and fractional improvement upon the baseline respectively. The top performing model is shown in bold.

w - (Exit 1 : Exit 2 : Exit 3 : Final Exit)	AUC	Raw ↑ (%)	Frac. ↑ (%)
Baseline: (0.0 : 0.0 : 0.0 : 1.0)	0.6878	-	-
(0.2 : 0.3 : 0.1 : 0.4)	0.7308	4.3	6.26
(0.2 : 0.2 : 0.2 : 0.4)	**0.7345**	**4.67**	**6.79**
(0.2 : 0.1 : 0.3 : 0.4)	0.7317	4.39	6.39

TABLE 4.4: A table of area under curve (AUC) measurements of linear probe accuracy plots for multiple branch configurations. Branch weighting is shown in the first column, raw AUC shown in the second, the third and fourth show raw and fractional improvement upon the baseline respectively. The top performing model is shown in bold.

These models perform the best since improvements are made over the baseline throughout the network, as opposed to individual locations. This is reflected in the larger fractional improvements, a maximum of 6.8%. Hence, combining these multi-weighted models with the finetuned training approach could yield good results.

From these results there is a clear change in intermediate layer performance when incorporating branches into the loss function, at the expense of final layer performance. This is most true in the linear probe accuracy results where the unbranched networks would show ~1% improvement over most of the branched networks in the final layers. This is also reflected in the class separation plots, which demonstrate class separation taking place earlier in the networks with branches in their optimisation. However, if these branched networks are first optimised to have good intermediate layer performance and then fine-tuned for final layer performance, the branched networks have good final layer performance as well as optimised, progressively intelligent intermediate layers. These models are all compared using AUC, showing that certain weight configurations make improvements of up to ~6.79% over the unbranched models.

We also investigated different training objectives and analysed hierarchically trained models using class hierarchy data. It was found that models given coarser grained objectives, could better distinguish class hierarchy earlier in the inference process but would perform worse than the unbranched model on the same coarse labels at the final exit. This suggests that models naturally distinguish class hierarchy when learning data, provided there are visual similarities within each coarse class. Furthermore, class weighting is investigated: using confusion matrices, it is evident that even with a low branch weighting it is possible to classify specific classes with high accuracy earlier in the network. Chapter 5 follows on from this, incorporating hierarchical information into the exit policy for improved early exit savings.

Finally CKA analysis was used to investigate the representations throughout the networks. It is evident from these plots that when incorporated into the loss function, branches shift representations that are pertinent in classification earlier in the network to allow early classification. However, these representations are changed to some degree, not matching exactly those from the unbranched networks. This suggests that branches learn an intermediate representation of the data, that allows for classes to be distinguished between one another, whilst also allowing the transformations necessary for final layer performance.

These results suggest that classification branches help models increase generalisability throughout the network and focus this increase earlier in the network rather than the end. It is also clear that only a low weighting is required for these benefits to take hold, a weighting of 0.2 in the earlier branch position being sufficient for earlier layer performance. Adding additional branches in the later positions can also improve this performance and enable more progressive generalisation throughout the model. Furthermore, results from figure 4.5 show that the training regime can play an important role in final layer performance and improve upon the early performance of branched networks, with fine-tuning the networks proving to be the most effective method.

This work has the allowed the inner workings of branched networks to be investigated in more detail, however it is not clear that these performance gains are tangible as currently they are limited to abstract concepts. To realise the benefit of branched neural networks, their resource usage at run-time need to be considered. Hence, the next section will consider this in the context of the neural network backbone.

4.2 The Effect of the Backbone on the Operating Range

The pareto-optimisation of the representations of a model allows its generalisability to be pushed earlier in the network and the activations it produces become more similar to later layers in models that are trained without branches. However, it is not clear that these representational benefits translate to tangible, efficiency benefits. This section will investigate this, through the lens of the model backbone and its impact on such efficiency benefits. This section makes the contributions listed below and addresses the literature review gap, *Optimising Architectures for Progressive Intelligence.*

Building an understanding of model scaling and its operating range We vary backbone scaling and design to understand how they affect the inference modes, allowing us to understand how to best use model design to influence the performance of a branched neural network across its operating range.

Forward hook method for simulating branched neural networks We introduce the forward hook method for easily simulating branched neural network inference. The forward hook method is possible in both PyTorch and Tensorflow libraries. We also integrate a script which easily creates custom ResNet architectures for simulation. Available at: `github.com/J-Dymond/adapting-branched-networks`.

Prototyping models with adaptive width at run-time We formalise and prototype an approach for using branched neural networks which have an adaptive width at run-time.

First it is necessary to cover the mode by which branched networks reduce resource usage: Early exiting inference. Using the inference procedure defined in algorithm 3 of section 3.3.2, we simulate inference on the peak performing AUC models of the previous section, these are shown in figure 4.10.

We find that there is not much to separate the branched models at their peak accuracies, as most results lie on or within their uncertainty bounds, except the early weighted model with a lower weighting on the middle branch (orange). All models appear to follow a similar distribution when the confidence requirement is varied, and the entropy threshold is increased.

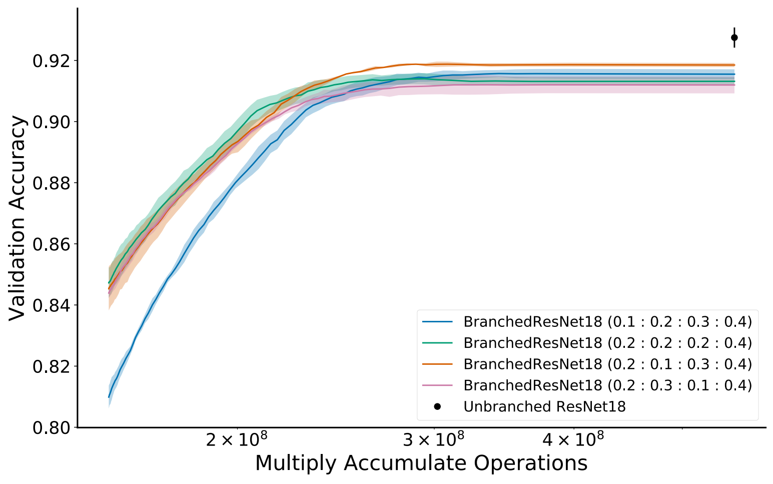


FIGURE 4.10: Multiply accumulate operations (MACs) against validation accuracy for various models. A distribution is given for the branched models, where the rightmost datapoint is the lowest entropy threshold, and the leftmost the highest entropy threshold.

The branch with the lowest weight attributed to the first early exit branch performs poorly at the low confidence threshold range, whereas the other models appear to fall within each others uncertainty bounds. Once again the orange model performs slightly better than the other models (0.5-1%). All models perform worse than their unbranched counterpart at peak accuracy, however, these models can perform in a wide range of power modes, their lowest power mode being $\sim 5\times$ more efficient than their highest in terms of MAC operations.

Whilst the optimal weighting configuration appears to be the orange configuration, we recognise that this result may not translate to other datasets, and will not translate to models with more than 3 early exits. Hence, the remainder of the analysis uses an equally weighted early-exit approach, which prioritises the final branch. This setup performed best in the pareto-optimality work of the previous chapter and is also the setup most commonly used in the literature. Hence, the findings will permeate furthest outside of this work. However, this work shows that optimising the branch weighting during training can marginally improve performance.

We further analyse the performance of the individual branches when the exit threshold is varied, by plotting the exit percentage and exit accuracy we can observe the behaviours of the branches as entropy thresholds are varied. This is shown in figure 4.11.

This shows that as the entropy threshold decreases the exit percentage, shown in the shaded regions, increases accordingly for the later branches, and decreases for the earlier branches. Furthermore the exit accuracy, shown with the lines of the same colour, increases across all branches as the entropy threshold decreases. This shows that as the entropy threshold decreases, the confidence required in classification increases making sure the model is more

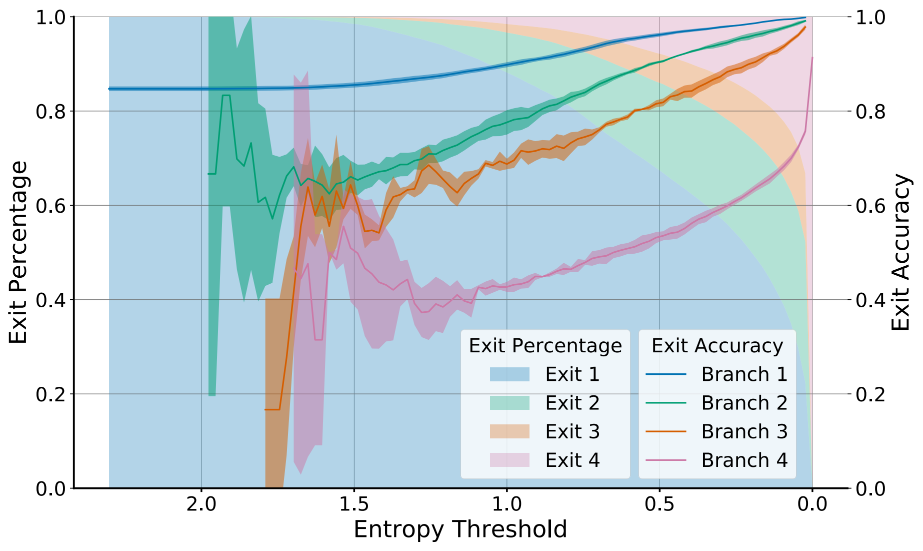


FIGURE 4.11: Entropy threshold against branch validation accuracy and branch exit percentages for an equally weighted branched model.

certain in an early exit, hence making the branches more accurate. Furthermore, at low entropy thresholds the early exits are volatile in their predictive accuracy, we call this the *activation point* of the early exit branch. This suggests that more stable exit mechanisms should be used to give more predictable results surrounding the activation point when tuning confidence thresholds.

There are two ways in which such an algorithms can be made more computationally efficient: The model backbone, $f_{1:N}(x)$ can be made more efficient, or the exit policy confidence$(f_{1:N}(x))$ should choose exits appropriately such that it can exit more frequently without incorrectly classifying inputs. Whilst the ResNet architecture is very successful in the field in terms of performance, it is very computationally intensive. Hence a more efficient backbone architecture should be identified, this section will discuss these ideas.

4.2.1 Efficient Neural Network Backbones

As discussed in section 2.3, efficient neural network architectures can be obtained in a number of ways, but ultimately there are two approaches: larger networks can be compressed, or bespoke networks can be designed that are computationally efficient. Initially, we choose the latter approach and vary the width and depth of the existing architecture to find the optimal combinations of architectures. We then look towards loading more efficient backbones from external sources, specifically the MobileNetV3 architecture.

4.2.1.1 Optimal Scaling of the ResNet

Convolutional neural networks can be scaled in two ways. One option is to make them wider, that is to increase the number of channels in each convolutional layer. The other option is to make them deeper, that is to increase the number of layers in the network.

To run this experiment, a generalised ResNet architecture was created, which as an argument would take depth and width multipliers. The width multiplier was simply passed to the block generating algorithm, which created the convolutional channels. Whereas the depth multiplier would be passed to a block allocation algorithm. Blocks in this context refer to a collection of layers, which act as building 'blocks' for the model itself. There are two distinct blocks in the ResNet paper, those are the *basic* blocks and the *bottleneck* block He et al. (2016).

The basic block is simply two 3×3 convolution layers placed one after another, whereas the bottleneck block uses a 3×3 convolution layer sandwiched between two 1×1 convolution layers. The 1×1 convolutions are used to compress the representation[1] of the input before the 3×3 convolution and then expand it again afterwards, hence forming an information bottleneck. This makes the skipped connections particularly important in such architectures, allowing information to bypass the bottleneck. Furthermore, this makes each block more lightweight computationally than if it were a basic block using the same number of input/output channels.

So far, this work has only considered networks using the basic block of the ResNet paper. However the original authors suggest using the bottleneck block for the architecture when moving beyond 34 layers (He et al., 2016). When investigating varying depth, we remove this degree of freedom in architecture design at first and use the basic block. We can then introduce the bottleneck block in the design to better realise its benefits when creating larger architectures.

We present MACs vs accuracy plots for ResNet architectures of varied width, depth, and number of branches. Since only one model of each were trained, we represent uncertainty using standard deviations from the mean accuracy value. By assuming the accuracy of the model represents a point of a Gaussian, we can assume the confidence interval will take the following form,

$$\Delta = \sigma \sqrt{\frac{A(1-A)}{n}}, \tag{4.5}$$

where Δ is the confidence interval, σ the number of standard deviations from the mean, n the number of samples, and A the mean accuracy. For the following results a σ of 3 is used, corresponding to a certainty of 99.7%.

[1]Here, compress refers to the reduction of channels in the layers, the second 1×1 convolution expands the representation by restoring the number of channels.

First we analyse the effect of width in the ResNet18 architecture, this is shown in figure 4.12 (a).

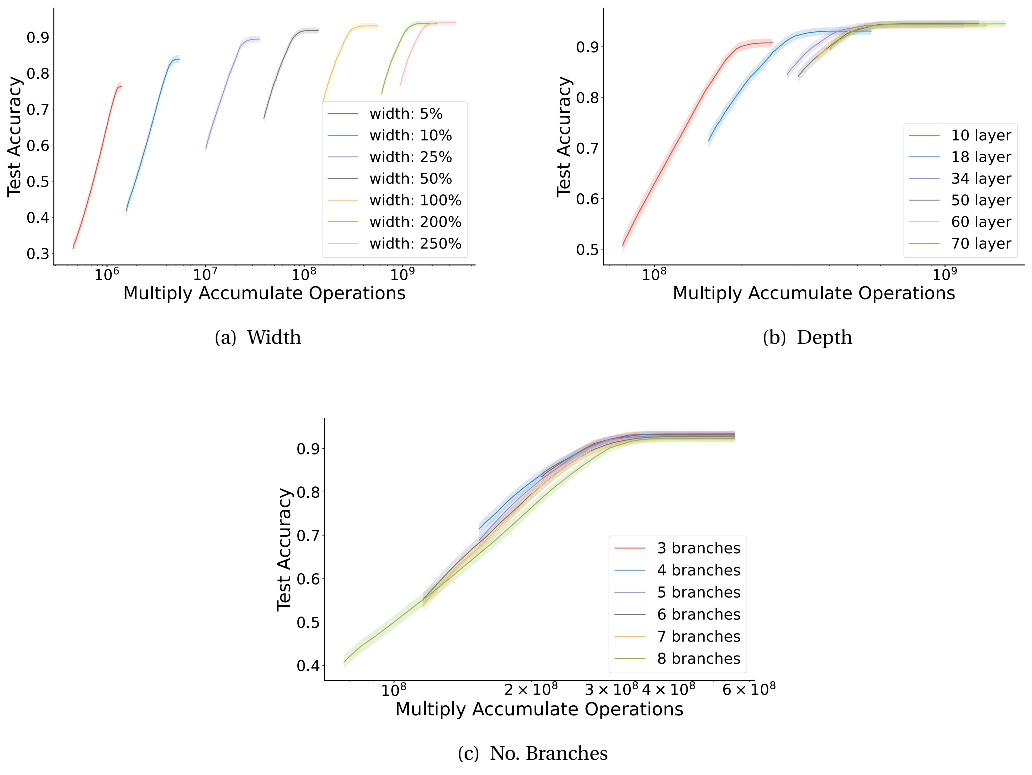


FIGURE 4.12: Validation accuracy against MACs for equally weighted branched ResNet18 models of varied width (a), depth (b), and number of branches (c). The bounds on each point represent the confidence interval at 3σ.

The width of a network has a consistent effect on the MAC usage of the model, without drastically effecting the classification performance of the model when trained under the same conditions. The model can be compressed to up to 5% of the original width without drastically reducing the performance of the model, considering the cost drop. It appears the performance drop-off starts between a width of 50% and 25%, as the performance of the 50% width model appears to match the full size models within the confidence bound. It is clear that the larger widths do not increase the accuracy outside of the confidence intervals for this dataset. The MACs benefits roughly follow what would be expected, since the time complexity of the convolutional operation amounts to that as shown in equation 4.6.

$$\mathcal{O}(D_m\, D_n\, N\, M\, D_f\, D_g) \tag{4.6}$$

Here, D_m, D_n, N, M, D_f, D_g refer to the kernel size, the number of input/output channels, and the output feature size respectively. Thus when everything is kept consistent, except a constant scaling parameter on the number of channels, there is an operations change of factor $(W_1/W_2)^2$ between two models. Where W refers to their respective widths. We see this behaviour in figure 4.12 (a), between the peak operating modes of the models suggesting the power calculation is a good estimate. It is evident that varying the width of the network is much more effective in reducing the computational cost of the network, than varying the early exit aggressiveness. The early exiting can at most save $\sim 50\%$ of the inference cost (in the far right hand model), with minimal loss in accuracy, whereas the network thinning has the potential to save $\sim 87.5\%$ (between the $2.5\times$ model and standard width model). Hence, the same accuracy values can be attained through reducing the width of the network. However, as this happens the benefit of the branching is reduced, with the 5% width network dropping in performance rapidly when early exiting aggressiveness is increased. The wider networks maintain the accuracy for longer and have a shallower drop. Section 4.2.3 investigates a shared weight allocation, using *slimmable* branched networks, which will allow for run-time switching of these width modes.

The depth of these networks is also analysed, whilst keeping the width consistent. This is shown in in figure 4.12 (b).

We find that the shallower model has a dramatically reduced minimum accuracy, but does operate over a decreased power range. As with increasing the width, there are diminishing returns when using deeper networks. The peak operating powers roughly follow what is expected, there should be a linear relationship between the power used and the depth. However, this will likely drop off as the model becomes bigger, as the input will be downsampled more, thus reducing the computational cost of the convolution operation. It appears on this particular dataset, the ResNet18 is the architecture which allows the greatest power range, without compromising the performance of the network greatly.

Finally we analyse the effect on the number of branches in the branched ResNet18 architecture, shown in figure 4.12 (c). The number of branches is varied between 3 and 8, each branch is equally weighted during training except the final branch which was always given a weight of 0.4, the others were hence weighted $0.6/n_{\text{branches}}$. We start at three branches to avoid training an early exit branch with a greater weighting than the final exit branch.

We find that an increased number of branches extends the operating range of the neural network, however this is at a dramatically reduced accuracy. It appears the eight branched model performs slightly worse throughout its operating range than the other models. However, the six and seven branched models extend this operating range without drastically compromising the performance at the greater confidence ranges. There seems to be no computational benefit of using any less than four branches, and the four branched model performs better than or matches that, of all of the other models in its MAC operating range. Furthermore, the discrete nature of the minimum power values is likely due to the limited number of branch connection points, which will also depend on the number of branches the model has. The benefits of having more branches could be compounded when additional exit policies are developed, hence, this work shifts focus on improving exit policies in chapter 5. However, the last way in which the backbone architectures can be improved is by using bespoke architectures which are computationally efficient. Hence, we investigate this in the next section.

4.2.1.2 The MobileNet Architecture as a Backbone

Many efficient neural network architectures have been developed in the field, but few have remained active in the field except the MobileNet. The architecture, first developed by Howard et al. (2017), was introduced to move inference towards the edge and introduced *depth-wise separable convolutions.*

In these networks they replace standard convolutions with the combination of a depth-wise convolution, one which processes each channel individually, and a point-wise convolution, one which combines these outputs to change the number of channels and create new feature maps. The result is a module which produces the same output as a standard convolutional filter, but with a reduced computational cost. The cost of a standard convolutional filter is

$$D_k \times D_k \times M \times N \times D_f \times D_f,$$

where D_k is the kernel size, M the number of input channels, N the number of output channels, and D_f the size of the feature map being passed to the module.

Whereas the depth-wise separable convolution can be split into two terms, one for each stage,

$$D_k \times D_k \times M \times D_f \times D_f + M \times N \times D_f \times D_f.$$

In a depth-wise separable convolutional filter, the spatial convolutional kernel is separated from the dimension change. This acts to reduce the computational cost by

$$\frac{D_k \times D_k \times M \times D_f \times D_f + M \times N \times D_f \times D_f}{D_k \times D_k \times M \times N \times D_f \times D_f} = \frac{1}{N} + \frac{1}{D_k^2}.$$

This cost can be up to 9 times less when using a 3×3 convolutional filter. This architecture is selected due to its computational efficiency, its performance, and its relevancy in state-of-the-art papers as a benchmark architecture, namely in branched neural network research Hu et al. (2020).

MobileNetV3 (Howard et al., 2019) differs from the initial implementation of MobileNet in that it incorporates inverted residual bottlenecks and neural architecture search in model design to boost performance.

Inverted residual bottlenecks are much like the residual units in the ResNet architecture (He et al., 2016), only they use lightweight convolutions with linear activations before and after the skipped connection. Between the two they incorporate a more expensive 3×3 convolution, with a ReLU activation function. This way, the representational power of the large convolutional kernel and its non-linearity is preserved, but information in the neural network predominantly passes through lightweight components.

The authors also incorporate neural architecture search (Zoph and Le, 2016; Liu et al., 2018) to redesign the most expensive layers of the network. Specifically they tune the number of filters in a given layer, and redesign the combination of blocks to make the model more computationally efficient without impacting the performance. They also introduce squeeze-excite layers within bottlenecks, which compress the activations into a channel-wise representation and concatenate this to the output convolution. They find these improve performance, without impacting computational expense. Hence, we implement a MobileNetV3 architecture for our work.

However, designing such architectures from scratch is involved, due to the ever-increasing complexity of the architectures that are successful. Furthermore, with the resources surrounding deep learning research becoming increasingly accessible, there are a multitude of *out-of-the-box* architectures that can be used in the field. Hence, there is an opportunity to develop a prototyping methodology, that can load these architectures as backbones and simply attach branches to the architecture at selected points. We call this the forward hook method.

Forward Hooks for Branched Networks Forward hooks refer to a method whereby the intermediate activations are extracted from a model during inference and used in some way. We name them as such because in the PyTorch deep learning library forward hooks are used to achieve this (Paszke et al., 2019). Once the activations are extracted, they can be passed to the branch which is designed manually. Whilst forward hooks do not allow for early exiting inference, they can be used to train the model effectively, and they can also be used to emulate early exiting inference provided the resource usage of the model can be calculated. Hence, we take this approach in designing branched networks, and evaluating them against one another. A diagram is shown in figure 4.13. Here the green components represent the

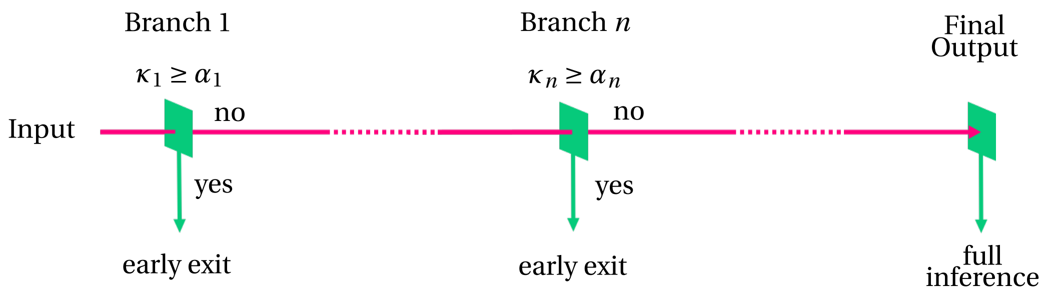


FIGURE 4.13: A diagram of the forward hook method with an early exiting algorithm annotated at the exit branches. Pink components are loaded externally, and green components are designed manually.

activation extraction, and the early exit classification branches. Whereas the pink components are the backbone network, which can be loaded from an external source. We use this method to incorporate the MobileNetV3 architecture into the work, and compare the cost-performance readings against the ResNet architecture.

4.2.2 ResNet vs MobileNet

To compare these architectures we analyse their accuracy vs MACs for varied entropy thresholds, the results of which are shown in figure 4.14. Both networks were trained under the same conditions, and trained for the same number of epochs. The results as with the previous experiments are averaged over three runs.

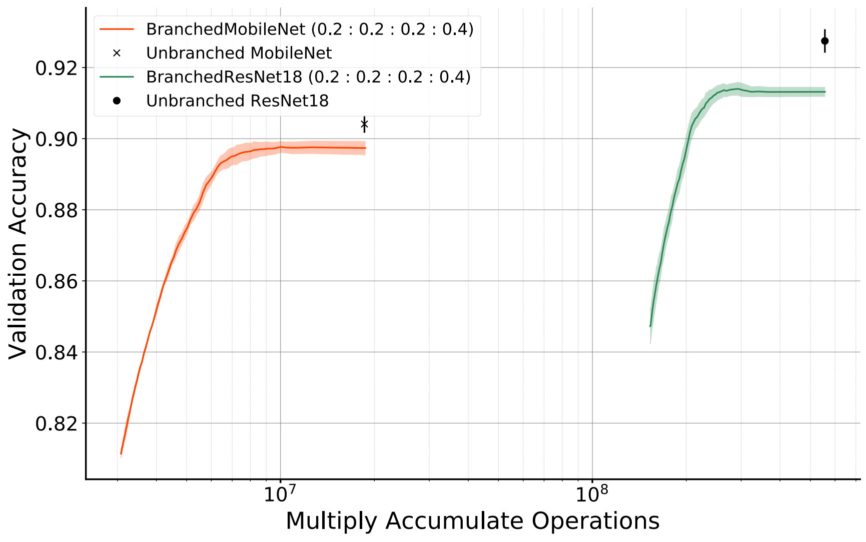


FIGURE 4.14: MACs against validation accuracy for varied entropy thresholds. The MobileNet architecture is shown in orange, and the ResNet in green. The unbranched architectures, of each are shown with the cross and circle respectively.

The MobileNet architecture is shown in orange and evidently it is much more efficient than the ResNet, ~15× more efficient than the ResNet at each entropy threshold. However the MobileNet also performs about ~2% worse than the ResNet, suggesting the peak operating modes cannot compete with the larger ResNet architecture. Both models perform worse than their unbranched counterparts, the Mobilenet slightly closer than the ResNet. This further suggests that there is a trade-off when co-optimising branches in an early exiting system and it is architecture agnostic. Furthermore, the pattern shown between the MACs and validation accuracy with varying entropy appears to be shared across the architectures. In the ResNet18 model we find that the mean accuracy actually peaks at a lower entropy value, before levelling off at a slightly lower value. This phenomenon has been recorded in the field before, and it described as ‘overthinking’ in a paper by Laskaridis et al. (2020).

We also compare the exit percentages and exit accuracy in the branched MobileNet architecture. These are shown in figure 4.15, which can be compared against those in figure 4.11.

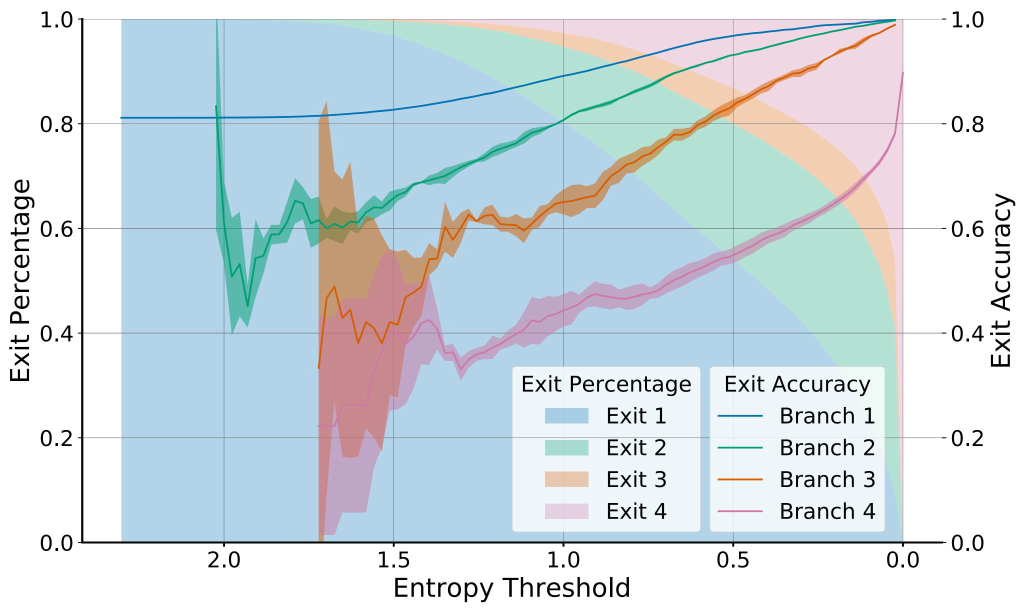


FIGURE 4.15: Entropy threshold against branch validation accuracy and branch exit percentages for an equally weighted branched MobileNet model.

We find that the MobileNet architecture uses the later exit branches at a lower entropy threshold. The accuracy approaches 100% in the first two branches, unlike the ResNet which performs slightly worse than 100% accuracy on the branches. However, it is worth noting that the model performs worse overall, also suggesting the peak operating modes hinder the overall performance of the model. The early exiting looks slightly less unpredictable than the ResNet, however the transition is still not as smooth and predictable as would be preferred. The branch accuracies are very noisy at their activation point, reinforcing the notion that early exit mechanisms should be wary of activation point noise when making their predictions.

4.2.3 Adaptive Width at Runtime

The previous sections investigated the notion of adaptive depth through inference time early exiting, in this section, the idea of run-time adaptive width is investigated. To do this we investigate a more recent development in the deep learning literature of *slimmable* models, that is models that have varying operating modes which can be adapted a run-time. In this section, we apply this idea to branched networks.

***Slimmable* branched networks** Slimmable networks can operate in a variety of *modes* which use varying amounts of the network components. Crucially, in these modes the weights of the network are shared. This means that operationally the modes can be varied with little latency, as weights do not need to be transferred. Theoretically, this principle of design could be extended to an infinite number of width modes as the following summation will always be satisfied.

$$\frac{1}{2^W} + \sum_{w=2}^{W} \left(\frac{1}{2}\right)^w = 1 \tag{4.7}$$

Here W refers to the number of width switches, w, available. The summation is an example of a geometric series: $\sum_{n=1}^{\infty} \left(\frac{1}{2}\right)^n$, which can be shown sums to $1-\frac{1}{2^n}$, and hence we add the term $\frac{1}{2^W}$ to ensure our model is the target width. The number of switches available is limited by the number of channels at any given point in the backbone, in the ResNet architecture this is 64. However, training such a network is challenging, with $W \times B$ outputs to optimise. Hence, when presenting the prototype we limit the number of width switches to three, training a model of switches $w \in \{1.0, 0.5, 0.25\}$, co-optimising 3B linear classifiers when doing so.

To test this concept in branched neural networks, three branched networks are stacked width-wise and trained in unison, of widths: {0.25×,0.25×,0.50×}. The outputs of these networks are stacked together cumulatively to produce networks of widths: {0.25×,0.50×,1.0×}, this concept is demonstrated in figure 4.16. The resultant network, with 4 branches, now has the potential to operate at 12 operating points on a given inference.

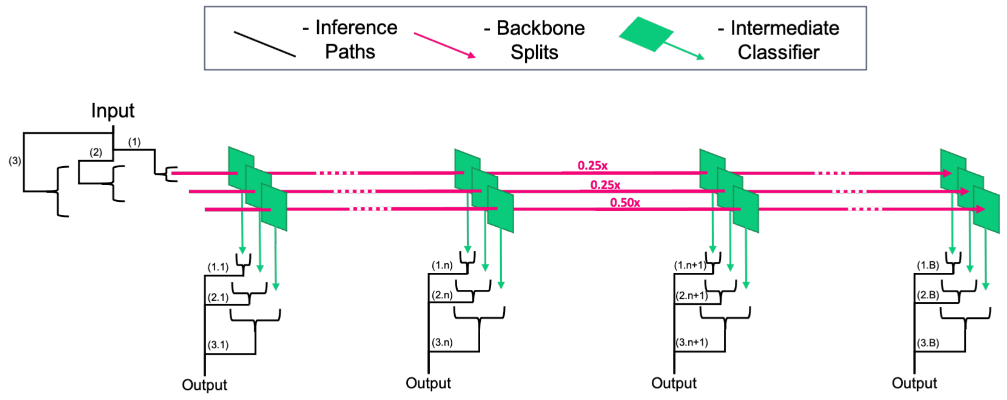


FIGURE 4.16: Representation of the prototype slimmable branched network. The three width modes combine the outputs of one or more networks at each branch. The width labels denote the width of a backbone split in relation to the full with of the slimmable network.

We compare our model with adaptive runtime width against standalone models of the same widths, the results are shown in figure 4.17.

We find that the slimmable model is not able to match the performance of the standalone models, and that the performance is limited when using the early exiting functionality of the model, in all cases the slimmable counterpart performs worse. We hence find that the early exiting on a conventional backbone network is more easily optimised than a slimmable

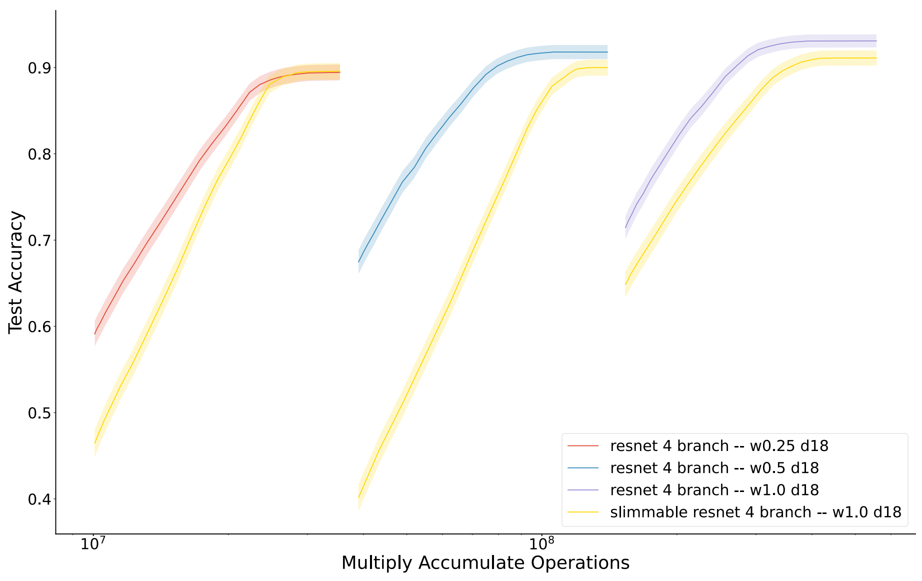


FIGURE 4.17: Validation accuracy against MACs for equally weighted branched ResNet18 models of varied width. The bounds on each point represent the confidence interval at 3σ. A slimmable model with 3 switches at {1.0,0.5,0.25} is shown in yellow, as well as the accuracy–MACs performance it can achieve.

branched network in this form. However, this work does present a working solution to a progressive intelligence model that operates with adaptive width and depth.

This work has looked to improve the resource usage of branched neural networks, as a means of improving the efficiency benefits that can be attained using branched neural networks.

First early exiting inference was presented on the pareto-optimised networks, it was shown that certain training configurations could marginally improve the cost–performance distribution on the branched ResNet18, however these results were likely dataset specific, and further results will need to be attained to find more generalisable results. However, it was shown that incorporating branches into the training could improve the cost–performance distribution of the network.

Following this, the neural network backbones were identified as a place where branched neural network inference could be made more efficient. To investigate this, the depth and width of ResNets were varied, it was found that increasing the width gave diminishing returns, and that decreasing it had little effect on the predictive performance down to a scaling of 0.5×. Similar results were found when scaling the depth, a decreased depth allowed for a dramatically increased operating ranges in terms of MACs.

The number of branches were also changed, to assess its effect on the operating range. It was found that more branches would extend the MACs operating range of the network, albeit at a reduced performance. Anything fewer than four branches did not increase the peak performance of the model, hence four branches are the minimum that should be used in these systems. Furthermore, the model using four branches outperformed all other models in terms of performance across the power modes it could operate in.

The work then turned its attention to loading bespoke architectures as opposed to scaling existing ones. For this the MobileNetV3 architecture was used as a backbone (Howard et al., 2019). This was shown to dramatically increase the MACs savings that could be attained, at the cost of a slightly reduced performance. These architectures can be loaded 'out-of-the-box' and a branch attaching mechanism was developed for prototyping these networks.

Finally we investigate models that have adaptive width at run-time, which we call slimmable branched networks, and present a prototype implementation of such networks. We find that these networks are overly complicated to optimise in the conventional manner. However, we do demonstrate a proof of concept which can reduce memory usage whilst maintaining a reasonable level of performance in the progressive intelligence domain, whilst also offering increased levels of adaptability.

While it is evident the structure of the backbone network can influence the position, and to some extent the shape, of the operating range curves the exit policy is what defines them. Hence, it is necessary to examine the exit policies and how they can be improved in progressive intelligence systems. This is considered in the next section.

4.3 Improving Early Exit Policies

We now move to the final step in the pipeline of the branched network as a progressive intelligence: inference. In this section the inference process itself is optimised, finding an easy to implement method which utilises the special ensemble nature of the branched neural network. Specifically we address the gap of *Improving Early Exit Policies* in section 2.4, making the following contribution.

A new ensemble based inference policy is developed We introduce a new early-exit policy for branched networks resulting in up to 44% MAC operations improvement in inference over conventional approaches, without the need for additional optimisation.

To improve early exit functions, it is important to formalise the early exit function itself, and how it interacts with the output of the model. Hence, the we will first formalise some basic concepts in the inference process of branched neural networks, construct some metrics, and formulate the early exits themselves.

4.3.1 Formalising Early Exits

To start, consider the following definitions:

$$\begin{aligned} f &= \text{branched model}, \quad b = \text{branch}, \quad B = \text{total branches}, \quad \theta = \text{model parameters}, \\ x &= \text{input}, \quad \sigma = \text{softmax} \quad N = \text{Number of training patterns}, \quad C = \text{Number of classes} \end{aligned}$$

Where softmax refers to that defined earlier in equation 2.8. Then we can use this with individual branch logits, to obtain branch outputs for a given input,

$$\text{Branch Logits:} \qquad z_b(x) = f_{\theta_b}(x)$$

$$\text{Branch Prediction:} \qquad \hat{y}_b(x) = \sigma(z_b(x)).$$

A target array can also be defined:

$$\text{Targets:} \qquad Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_N \end{bmatrix}.$$

This will be of shape $1 \times N$, however, if this is one-hot encoded it will become shape $C \times N$. Traditionally data is represented using a one dimensional array with class indexes representing the classification. When using one-hot encoding there is a $C \times N$ dimension array where for a given input x, there is a $1 \times C$ array where all values are zero except for the index that corresponds to the class, c, which is given the value of one. That is for a given input x, of class c, the one-hot encoded label array Y is defined:

$$Y_{i,x} = \begin{cases} 1 & \text{if } i = c \\ 0 & \text{otherwise} \end{cases}$$

The same structure can be applied to the output predictions of the model, which are obtained by taking the index of the maximum argument of the branch outputs, $\hat{Y}$,

$$\text{Branch Predictions:} \qquad \hat{Y} = \begin{bmatrix} \arg\max(\hat{y}_1(x_1)) & \arg\max(\hat{y}_2(x_1)) & \cdots & \arg\max(\hat{y}_B(x_1)) \\ \arg\max(\hat{y}_1(x_2)) & \arg\max(\hat{y}_2(x_2)) & \cdots & \arg\max(\hat{y}_B(x_2)) \\ \vdots & \vdots & \ddots & \vdots \\ \arg\max(\hat{y}_1(x_N)) & \arg\max(\hat{y}_2(x_N)) & \cdots & \arg\max(\hat{y}_B(x_N)) \end{bmatrix}.$$

This array will be of shape $B \times N$, but if it is one-hot encoded, it can take the shape $B \times C \times N$.

In the case where both Y and $\hat{Y}$ are one-hot encoded, the row-wise dot product of them, $\hat{Y}_{b,x} \cdot Y_x$, for a given input x, returns the binary accuracy value of that branch on the input. Which, if carried out for all inputs and branches will produce a one-hot encoded branch accuracy array,

$$\text{Branch accuracy:} \qquad \hat{A} = \begin{bmatrix} \hat{Y}_{1,1}^{\top} Y_1 & \hat{Y}_{2,1}^{\top} Y_1 & \cdots & \hat{Y}_{B,1}^{\top} Y_1 \\ \hat{Y}_{1,2}^{\top} Y_2 & \hat{Y}_{2,2}^{\top} Y_2 & \cdots & \hat{Y}_{B,2}^{\top} Y_2 \\ \vdots & \vdots & \ddots & \vdots \\ \hat{Y}_{1,N}^{\top} Y_N & \hat{Y}_{2,N}^{\top} Y_N & \cdots & \hat{Y}_{B,N}^{\top} Y_N \end{bmatrix}.$$

This will be of shape $B \times N$, and can be represented more efficiently when written as in equation 4.8. This property is presented in figures 4.11 and 4.15.

$$\hat{A}_{i,j} = \hat{Y}_{i,j}^{\top} Y_i \tag{4.8}$$

Finally, if the one-hot encoded class prediction matrix is integrated across all training inputs, we can obtain a branch-class accuracy matrix,

$$Acc_{i,j} = \sum_{n}^{N} \hat{A}_{i,j,n}.$$

Generalised exit policies will take as input the information produced by the model and predict a class along with its classification confidence. For the ith exit this will be C, which will need to exceed a threshold, α in order for the early exit to take place.

We now formalise a number of exit policies. First we define the entropic exit function as

$$\text{Entropic:} \qquad e_i(x) = -\sum_{c}^{C} \hat{y}_{i,c}(x) \ln\left(\hat{y}_{i,c}(x)\right), \tag{4.9}$$

where c is a given class in $\hat{y}_i$. The confidence is given by $\kappa_i = 1 - e_i / C$, making the exit policy:

$$\text{Inference on input } x \begin{cases} \kappa_i(x) \geq \alpha_i & \text{Exit} \\ \text{otherwise} & \text{Continue} \end{cases} \tag{4.10}$$

One issue with such a policy is that the only information governing an exit is taken from the current branch. Consequently, if an exit is not deemed suitable any information obtained is discarded when the inference is continued. To overcome this, predictions may be *ensembled* and combined in a way which improves the performance over an entropic policy (Sun et al., 2021; Wolczyk et al., 2021). A very simple option for ensembling is to compare a branch output to the output before it, and exit early if they match. For this work, such an exit is referred to as *mutual agreement.* As shown in equation 4.11 this can be represented as the dot product of the one-hot encoded prediction vectors between the two branches.

$$\text{Mutual Agreement:} \qquad m_i(x) = \hat{Y}_i(x)^{\top} \hat{Y}_{i-1}(x) \tag{4.11}$$

Then the exit policy is given by

$$\text{Inference on input } x \begin{cases} m_i(x) = 1 & \text{Exit} \\ \text{otherwise} & \text{Continue} \end{cases} \tag{4.12}$$

One issue with this policy, is that it is not tune-able. Hence, the performance–cost point of this approach will be fixed for a given architecture. Furthermore, in the mutual agreement paradigm, inference can never halt at the first early exit branch, as there is no previous branch to compare against. To overcome this, such an approach can be given an entropic element to its decision policy, in what can be called *entropic mutual agreement* as

$$\text{Inference on input } x \begin{cases} (\kappa_i(x) \geq \alpha) \vee (m_i(x) = 1) & \text{Exit} \\ \text{otherwise} & \text{Continue} \end{cases} \tag{4.13}$$

Three exit policies have been defined, which can be applied to the output predictions of any model, these are now compared in the next section.

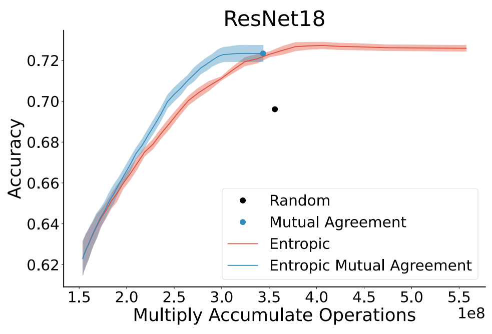
4.3.2 Comparing Early Exits

All three of these exit policies can be compared against one another in the MACs vs accuracy space for varied entropy thresholds. To do this we take a trained model, and apply the exit policies to the softmax output of the model to determine when the exits would have taken place. This allows the accuracy and power usage to be calculated. This is shown below in figure 4.18 (a) for the ResNet18 architecture on CIFAR100.

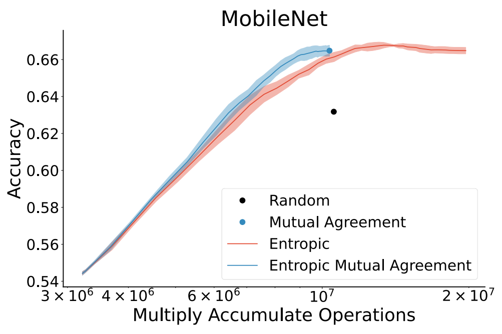
It is clear that the mutual agreement policy can perform to a high accuracy, within the error bounds of the original exit policy. However, it is also performing at an average power value that is $\sim 1.5\times$ more efficient than the maximum power mode of the entropic policy. When the entropic condition from equation 4.13 is applied in combination to this condition, the model can match this performance at a lower power mode, before the performance drops and moves towards the entropic condition. It is also worth noting that the entropic mutual exit condition never performs worse than the entropic policy at a given power usage value. It also dramatically reduces the length of the 'tail' at the peak accuracy value, meaning there is less redundant resource usage when reaching the peak accuracy value. Furthermore the small peak performance drop on the mutual exit policy is within the uncertainty bounds of both models. Finally, a random exit policy is shown with the black dot on the graph, as a sanity check to ensure the policies perform better than random selection.

We then show the same results for the MobileNet architecture in figure 4.18 (b).

The benefits are shared between the two architectures on the CIFAR100 dataset. The marginal drop in peak performance is lessened in this architecture, however it is worth noting that the



(a) ResNet18



(b) MobileNet

FIGURE 4.18: Entropy threshold against branch validation accuracy against MACs for the exit policies on different backbone architectures, tested on the CIFAR100 dataset. In (a) the ResNet18 model is shown, in (b) the MobileNet model.

peak performance is lower than the ResNet18 architecture. This network also performs better than the random exit test.

In order to better analyse the benefits of the exit policy, the computational cost improvement can be plotted against entropy. We choose entropy as the accuracies cover the same range between the two exit policies. Hence, as entropy is currently the best indicator of classification performance in a tune-able early exit network we compare the cost of classifications at this confidence in figure 4.19. However, when moving between models that operate across different accuracy ranges, more accurate estimates of classification confidence should be estimated. The benefits of the improved early exit policy occur earlier in the MobileNet ar-

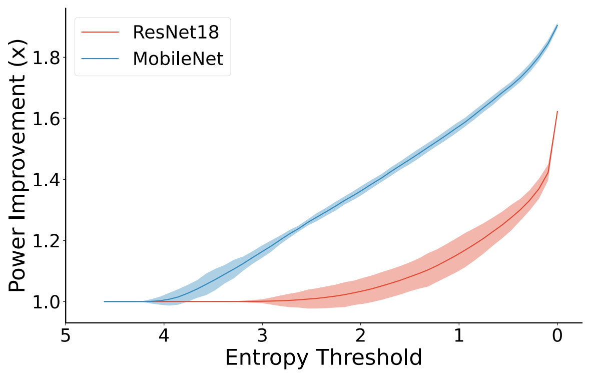


FIGURE 4.19: Power improvement at a given confidence threshold for both the ResNet18 and the MobileNet architecture. Models were trained on the CIFAR100 dataset.

chitecture than they do in the ResNet18 architecture. Furthermore, the performance benefit is greater across all confidence values in the MobileNet architecture. It also emphasises the power improvement that can be obtained using this improved early exit, regardless of architecture, particularly at the peak confidence values.

Following the work focusing on the backbones, the inference process has been investigated in more depth. The early exiting process was formalised, and two new early exiting systems were developed: *mutual agreement* and *entropic mutual agreement*, the latter being an extension of the former. Mutual agreement was a simple policy which performed an early exit when the branch agreed with the one before it. The entropic mutual agreement added an entropic condition to this, which bypassed the mutual agreement condition if the model was confident enough in its prediction. It was found that on the CIFAR100 dataset this policy could outperform the entropic policy, on both the ResNet18 and MobileNetV3 architectures. This easy to implement policy requires no additional training or optimisation, unlike recent policies in the field (Wolczyk et al., 2021).

4.4 Summary

This chapter has explored branched networks as a progressive intelligence system, from their training regimes to their inference methods.

In the first section, we analyse the loss function of a branched network and show how it affects the internal representation. Different branch weight and placement configurations were trialed, and compared using AUC showing that certain weight configurations make improvements of up to ~6.8% over the unbranched models.

We also investigated different training objectives and analysed hierarchically training using class hierarchy data. It was found that models given coarser grained objectives could better distinguish class hierarchy earlier in the inference process, but would perform worse than the unbranched model on the same coarse labels at the final exit. This suggests that models naturally distinguish class hierarchy when learning data, provided there are visual similarities within each coarse class. Furthermore, class weighting is investigated and using confusion matrices it is evident that even with a low branch weighting it is possible to classify specific classes with high accuracy earlier in the network.

Finally CKA analysis was used to investigate the representations throughout the networks. It is evident from these plots that when incorporated into the loss function, branches shift representations that are pertinent in classification earlier in the network to allow early classification. However, these representations are changed to some degree, not matching exactly those from the unbranched networks. This suggests that branches learn an intermediate representation of the data, that allows for classes to be distinguished between one another, whilst also allowing the transformations necessary for final layer performance.

These results suggest that classification branches help models increase generalisability throughout the network and focus this increase earlier in the network rather than the end. It is also clear that only a low weighting is required for these benefits to take hold, a weighting of 20%

in the earlier branch position being sufficient for earlier layer performance. Adding additional branches in the later positions can also improve this performance and enable more progressive generalisation throughout the model. Furthermore, results from figure 4.5 show that the training regime can play an important role in final layer performance and improve upon the early performance of branched networks, with fine-tuning the networks proving to be the most effective method.

There are a number of open questions from this research as well as a number of unanswered questions from the research objectives of the PhD. This work has allowed the inner workings of branched networks to be investigated in more detail, however it was not clear that these performance gains are tangible as currently they were limited to abstract concepts. To realise the benefit of branched neural networks, their resource usage at run-time needed to be considered. Hence, we considered the neural network backbone.

In the second section we vary the width, depth, and number of branches of these systems and find that the peak and minimum operating powers roughly match the theoretical expectations. However it is the effects on the operating curves that is the main interest of this work, and how the architectural properties of the model effect this.

We find the width of these systems has the greatest effect on their operating range, shifting orders of magnitude without significantly decreasing accuracy, we do find the drop of accuracy becomes steeper in these cases though. Varying the depth of the networks acts to extend this range roughly about its original value, but the accuracy is not reduced significantly nor is it increased. The number of branches extends the bottom end of these curves, allowing operation at lower power values.

We then discuss different architectural designs at a more granular level, exploring the MobileNet architecture as a backbone. To do this we use an external codebase, and introduce forward hooks for simulating branched networks. We find the mobileNet operates in a lower power range, but the operating curve is retained, suggesting that the operating ranges seen are architecturally agnostic.

In this section we also examine adaptive widths at run-time, and introduce a proof-of-concept *slimmable* branched network. We find the network can operate across multiple power ranges, at roughly the same accuracy achieved in the previous non-slimmable models. However, reductions in accuracy compromise the benefits of a slimmable approach. The poor accuracy performance is likely due to the increased optimisation cost of additional networks, and more holistic opitmisation approaches should be considered.

Finally, the third section shifts attention to the inference policy which governs the input-wise handling of data and power use. We formalise exit policies and introduce a new parameter-free ensembling policy, *mutual-agreement*. It is easy to implement, comparing consecutive branch outputs to determine an early exit.

We also introduce *entropic mutual agreement*, an adjustable extension which uses the entropy of the output, in conjunction with the ensembling policy, allowing early exiting when the model is confident *or* when it has predicted the same class on consecutive branches. We find this policy improves on conventional early exiting in both the ResNet and MobileNet architectures, peaking at 1.6× and 1.9× respectively.

This chapter has comprehensively explored branched neural networks and their utility in progressive intelligence for classification. We find that inference methods are the most impactful area of improvement, due to the applicability to any progressive intelligence architecture. Hence, in the final chapter we will turn our attention to inference. However, in this domain models are not always given clean inputs with single labels. There is often ambiguity in the label space, noise in the input space, and adversarial attacks in the application space. Therefore, it is important that progressive intelligence systems can operate in these challenging environments. We can consider these sources of additional information and in the next chapter we will consider how they can be used to further improve the utility of progressive intelligence systems.

Chapter 5

Leveraging Additional Information in Progressive Intelligence Systems

The previous chapter discussed how progressive intelligence can be implemented and improved using branched neural networks in the classification domain. In this chapter we consider the idea of incorporating additional information into the system and how this information can be used to increase the progressiveness of the system.

Firstly, we consider epistemic uncertainty, and how it can indicate the sample being *unclassifiable*. This information can be used to help prevent resource wastage on such samples. This would be of use in the drone example given in section 1.2.1.1, if the system moves into a territory in which it is not familar. Whilst it might still be able to identify target data, it will not be able to classify novel inputs. Hence, it would be beneficial to reject these. In this scenario it could also report this to an operator who can take appropriate action.

Secondly, we consider hierarchies in a label structure and how this can be exploited to ignore inputs which are not pertinent at classification time. This may be of use in classification problems where many of the inputs are not of interest. For example, in the social media moderation case study in section 1.2.1.2, many posts on these platforms are not malicious, only a subset require action. Hence, it would be beneficial to reject such inputs early in the classification process to save on computational costs. In this case the fine-grained labels might pertain the reason for reporting a particular message.

As shown in the previous sections, neural networks have been incredibly effective in pushing performance on benchmarks, however the metric of success has always been centered around accuracy. We find that whilst these networks are performant on their tasks they are brittle to data outside that which they are trained on and to data which has been deliberately altered to elicit erroneous outputs in the network. In this section we briefly discuss the background of the ways in which these systems can fail as well as how these shortcomings can be overcome. Then we will look towards progressive intelligence to understand how we can deal with such inputs in these systems.

To understand where machine learning models can go wrong it is beneficial to first understand the uncertainty faced by them. Uncertainty in this context can be categorised into to types: aleatoric and epistemic. Aleatoric uncertainty is the uncertainty arising due to the natural stochasticity in the data, it can be likened to the ability for the model to interpolate effectively. On the other hand, epistemic uncertainty arises due the training data insufficiently representing the data being observed by the model. That is, the model is encountering data it is not familiar with. This might be due to insufficient training data in the area, or it might be due to the data being entirely out-of-distribution, and the model has failed to extrapolate effectively.

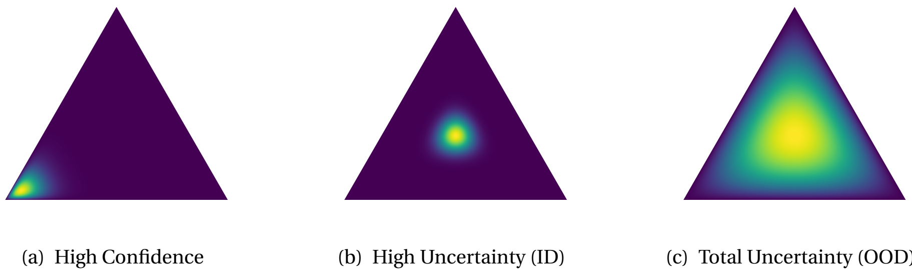


FIGURE 5.1: An exemplar 3 class simplex representing the uncertainty behaviour of a classifier. The lighter colour refers to the probability of a class being placed in the particular area of the space.

A useful graphical tool in understanding uncertainty can be seen in figure 5.1. This figure is known as a simplex and each vertex represents a class in the dataset. This makes a simplex a $K-1$ dimensional object, where K is the number of classes. Here there are 3 classes and a lighter colour in this space equates to a higher probability of placing the sample there. Thus, for a confident prediction a vertex will be highlighted as in figure 5.1 (a), whereas for uncertain predictions figure 5.1 (b) illustrates an aleatoric case, and figure 5.1 (c) an epistemic case. You might equate the area of yellow space in this diagram to the uncertainty of the prediction, but in this case is purely for illustrative purposes.

Figure 5.1 represents the optimal responses of a classifier to certain inputs. In (a) the input is in-distribution (ID) and can easily be attributed to one of the classes. In (b) the input sits perfectly between the three classes at classification time, so the model is confident that it is ID but cannot give a prediction. This can be considered a *known-unknown*. Finally, in (c) the model cannot give a meaningful output with the input, as the data is OOD. Hence, total uncertainty should be returned, this can be considered a *unknown-unknown*.

In reality models will often output incorrect classifications with high confidence as in figure 5.1 (a), when encountering data outside of their trained distribution. Hence, epistemic uncertainty, or lack thereof, can be considered a root cause of the lack of robustness, as the model can become over-confident in its prediction. The area of study concerning the behaviour of a model as it moves out of its trained distribution has been termed *graceful degradation*. One contribution of the PhD is a comprehensive survey of this field (Dymond, 2021). However, much of the work in that survey is tangentially related to progressive intelligence. In this thesis we are mainly concerned with *passive* approaches to graceful degradation, these are techniques which require no additional optimisation once deployed.

We can consider two elements to this problem: first, it is important to return correctly calibrated epistemic uncertainty, and second it is important to understand how we handle this uncertainty. In progressive intelligence, we are concerned with the latter, as at inference time we can only deal with our model outputs. However, it is still useful to understand the vulnerabilities in such models. We can consider two ways in which this uncertainty is generated, one is through data being out-of-distribution which is when the data lies outside of the trained distribution, the other is through malicious means where the sample is generated to produce an erroneous output.

Out of distribution data can be considered a continuous issue, in one sense the training and test distributions are out of distribution from one another, but perhaps less so than a different dataset containing the same classes, which is in turn less OOD than a dataset containing entirely different classes, and so forth. Hence, when considering OOD data it is important to clarify the extent to which the data is OOD, in this work we consider a variety, but we start at data with different classes. This concept is explored in figure 5.2, showing the continuous shift in nearest neighbour distances from uncertainty that is aleatoric in nature to that which is epistemic, we also hint at the possibility of rejecting data belonging to the latter category. Data which shares the same classes is often described as out-of-domain in the field, and represents an entirely different subset of this area.

Adversarial data on the other hand is data which is deliberately designed to cause erroneous predictions. This is a very rich area in the field and there are a variety of methods to generate such samples, a recent comprehensive review has been written by Han et al. (2023). We will briefly discuss some of the relevant methods for this work, namely the area of adversarial *perturbations*, which are small perturbations applied to input images designed to cause erroneous predictions.

Broadly, these techniques can be split into two types: *white-box* attacks, those which have full access to the model, and *black-box* attacks, those which do not. We consider white-box attacks in this work, as they are often the most effective and we wish to understand how we can deal with such issues in the progressive intelligence space.

One efficient method in the field is the *fast-gradient-sign-method*, (FGSM) (Goodfellow et al., 2014). This method applies a perturbation defined using the sign of the gradient, which is multiplied by a small value ϵ and added to the original input to create an adversarial example. This perturbation, η, can be defined as:

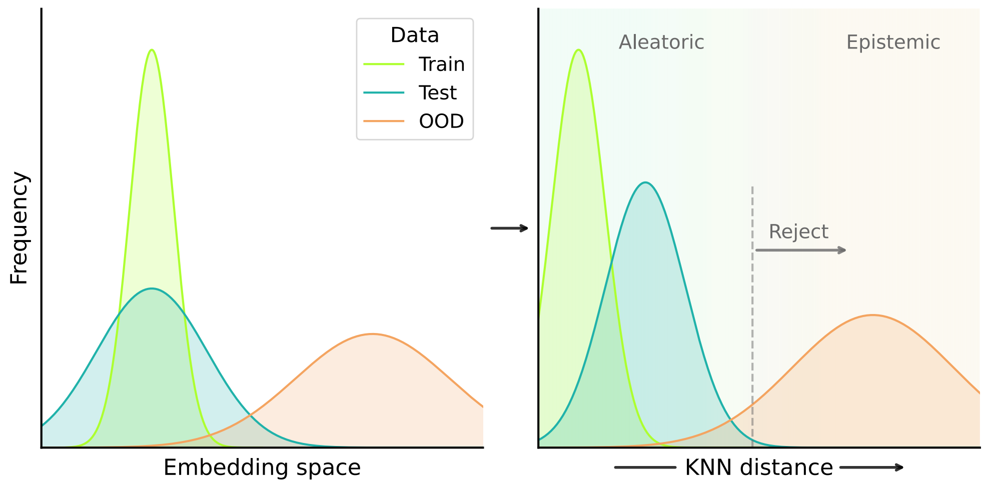


FIGURE 5.2: 1-dimensional projections of distributions in the embedding space of neural networks. The expected embedding distributions are shown in the left figure, and expected KNN distances in the right. The embedding representation of OOD data is expected to be clustered away from the target dataset. We label the uncertainty encountered in the KNN distance regions: aleatoric is that which is close to the the training distribution, epistemic is characterised with larger KNN distances.

$$\eta = \epsilon \operatorname{sign}\left(\nabla_x \mathscr{L}(\theta, x, y)\right), \tag{5.1}$$

where $\mathscr{L}$ refers to the loss function of the model, θ the model parameters, and x and y the input and target, respectively. Using this for an adversarial attack generates our perturbed input $\tilde{x}$ as

$$\tilde{x} = x + \eta.$$

This method takes one gradient step in the direction of the perturbation. Following the ideology of using the gradient, work by Kurakin et al. (2017) proposes an iterative step along this gradient to increase performance, showing that introducing random initialisation to the gradient descent problem enables more effective adversarial samples to be found. Work by Moosavi-Dezfooli et al. (2016) seeks to find the minimum perturbations necessary for achieving erroneous outputs. Finally, Moosavi-Dezfooli et al. (2017) propose a method which produces image-agnostic perturbations, which they call *universal* perturbations and they find by aggregating perturbation vectors across many images. The more recent literature has moved away from developing these attacks and towards black box methods, defending against attacks, and generating attacks on large language models.

Once epistemic uncertainty has been identified it is important to briefly discuss the potential actions a model can take. Perhaps the safest thing to do is to reject the sample, the notion of rejecting classification has been studied in depth previously (Geifman and El-Yaniv, 2019; Hendrickx et al., 2021; Barandas et al., 2022). In this section, we consider this eventuality.

Another option might be that the model can exploit additional information in the label space which can be leveraged, such as hierarchical structure. In the hierarchical classification problem, rather than asking the model to make a prediction in an area in which it is uncertain, the model is instead asked to predict a parent class at a level of hierarchy at which the model is more confident. For example, if the model failed to recognise an animal as a pigeon it might be able to recognise that the animal is a bird and offer a range of classes the bird might fit into. This can be seen as a compromise between the other passive approaches to the graceful degradation problem; the image is not classified due to uncertainties, but more information is offered as opposed to simply rejecting the input. In this chapter we consider label hierarchies from a different perspective, instead allowing adaptability at run-time with target class distributions.

We consider these two approaches to dealing with uncertainty in this chapter. This will provide a progressive intelligence system with additional methods of halting inference, and thus saving power. First we perform a study on rejecting inputs and how epistemic uncertainty is not only indicative of OOD samples, but also adversarial ones. We then look at hierarchically labelled datasets and how they can be handled in the progressive intelligence domain. Finally, we look at the combination of both in the final section.

5.1 Exploiting Epistemic Uncertainty in Branched Networks

Assuming the OOD input cannot be classified and the adversarial data successfully fools the model, we refer to these collectively as *epistemically* uncertain inputs. These are inputs outside of the target distribution of the classifier and thus induce epistemic uncertainty. In safety critical scenarios, it could be dangerous for the classifier to process them, for example, if a self-driving car misclassifies the moon as a traffic light and stops the car suddenly. In such cases, rejecting the classification may be more appropriate.

In these environments, it would be better still if these inputs could be rejected early in the inference process to save power. The idea of robustness has seldom been incorporated into the area of early exiting neural networks, and when it has, the work has focused less on handling epistemic uncertainty, and more on the classification performance on clean data (Hu et al., 2020; Meronen et al., 2023). This overlooks an important factor of resource wastage that OOD and adversarial data introduces to the early exiting classifier.

In conventional models, the best case scenario is the input passes through the model and absolute uncertainty is outputted, e.g. Sensoy et al. (2018), or a reject option is given to the classifier, e.g. Geifman and El-Yaniv (2019). In the early exiting field, there is also an opportunity to reject these inputs early and thus save resource usage. This is the idea we address in this section.

To address this challenge, we use two uncertainty measures. Entropy is used to quantify uncertainty in the *aleatorically* uncertain inputs, and we use a KNN-based classifier to catch epistemically uncertain inputs, allowing for distribution-aware early exiting.

This is the first implementation of OOD adversarial detection in early exit architectures. Furthermore, the method works out-of-the-box, meaning no additional optimisation is required. The following contributions made in this work are listed below and it addresses the following gaps in the field from section 2.4: *Introducing Epistemic Uncertainty to Progressive Intelligence Systems*; *Improving Early Exit Policies*; *Introducing New Early Exiting Mechanisms*:

Exploring the effects of epistemic uncertainty in branched neural networks We present a study of robustness in branched neural networks, addressing OOD robustness as well as adversarial robustness. We find that both adversarial and OOD inputs induce epistemic uncertainty in neural networks, characterised by large KNN distances.

Introducing OOD and adversarial detection to early exiting architectures We introduce an easy to implement KNN-based OOD detection technique which can be applied out-of-the-box in branched neural networks. A novel distribution-aware early exiting system is introduced, which incorporates KNN-based OOD detection alongside conventional early exit classification. Our new early exiting system allows the classifier to save additional power through rejecting inputs it is not able to classify.

Improving inference performance with a new early exiting algorithm A mixed input test set is introduced for distribution-aware early exiting. Using our method, we can save up to 40% power usage or improve accuracy by up to 20%, when compared to a conventional early exiting policy operating under the same constraints.

5.1.1 Out of Distribution and Adversarial Inputs in Branched Neural Networks

We can denote the backbone as a function f and each classification branch as $f_b(x)$, where $b \in \{1, 2, ..., B\}$, and B is the total number of branches. Each branch will give the softmax output vector y_b. We train these networks through the joint optimisation of the branches, with a weighted loss function and a target vector $\hat{y}$ with K target classes. This means our losses are the same as in the previous sections, as seen in equations 3.13 and 3.14.

Using conventional methods, neural networks are often trained to give confident classification outputs in the supervised setting. This presents a challenge when OOD data is encountered. Whilst they might not give results of high confidence, they do not return results which are of low confidence. In branched neural networks, confidence is often quantified for early exiting using entropy Teerapittayanon et al. (2017); Hu et al. (2020). To motivate the introduction of an additional classification method for early exit rejection, we first analyse entropy probability densities when presented with unclassifiable data, shown in figure 5.3.

In accordance with the benchmarks set by Yang et al. (2022), we pass various OOD datasets to our model: Describable Textures Dataset (DTD) (Cimpoi et al., 2014), CIFAR100 (Krizhevsky et al., 2009), Street View House Numbers (SVHN) (Netzer et al., 2011), and Tiny-ImageNet (Le and Yang, 2015). We use a branched ResNet18 (He et al., 2016), which has 3 branches spaced equidistantly, and a final exit which we call the 4th branch. This is trained to convergence on CIFAR10 (Krizhevsky et al., 2009), achieving an accuracy of ~ 95%. We believe our method is not specific to the ResNet18 architecture a number of architectures can be used with little variation in general behaviour, as shown in the previous chapter and other work (Teerapittayanon et al., 2016; Hu et al., 2020; Dymond et al., 2022).

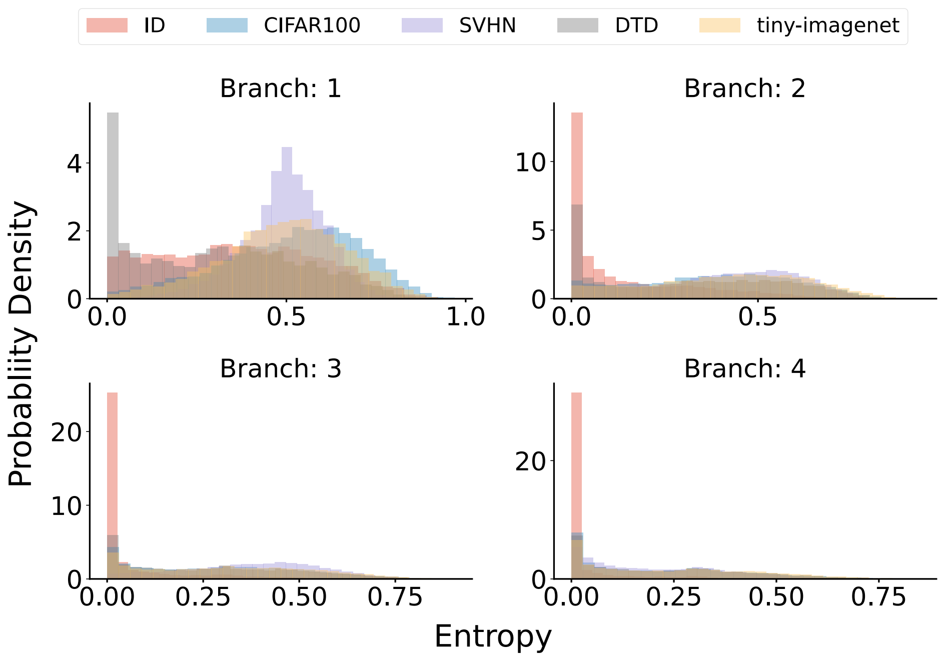


FIGURE 5.3: Histograms denoting the entropy probability densities for each branch on ID data and various OOD datasets.

As indicated by the probability density graph, we find earlier branches generally give outputs of lower confidence, except for DTD, which gives a large quantity of high confidence outputs. As inputs are processed further through the network, ID data is much more likely to prompt low entropy outputs than OOD data. Importantly, however, it is difficult to distinguish these distributions in the higher entropy ranges. Therefore, entropy alone is insufficient for distinguishing ID inputs from OOD inputs. In an early exiting scenario where entropy thresholds are varied at run-time, such networks will be susceptible to confusing ID data with OOD data.

We also examine adversarial inputs, using the fast gradient sign method (FGSM) to generate adversarial inputs, as in equation 5.1.

We vary the perturbation, ϵ, between 0 and 0.3. At 0.3, we find the performance of the model has degraded completely as shown in figure 5.4.

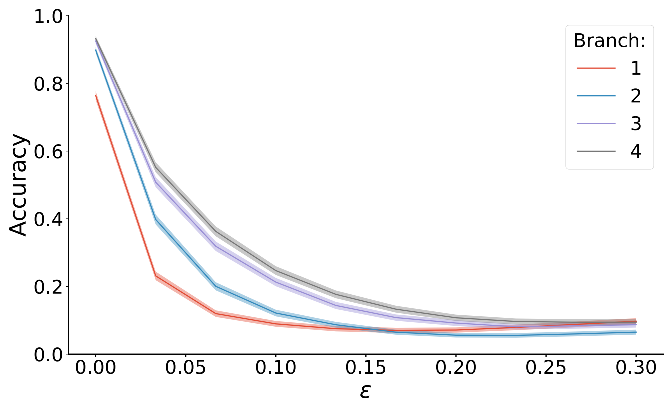


FIGURE 5.4: Accuracy degradation of each branch of the network when presented with adversarial data. We vary ϵ between 0 to 0.3, which is denoted on the x axis. Classification accuracy is shown on the y axis.

Accuracy drops prohibitively when applying the perturbation to the inputs, but the later branches handle this marginally better than the earlier branches. To understand the effect of the attacks on entropy, we analyse the entropy distributions in figure 5.5.

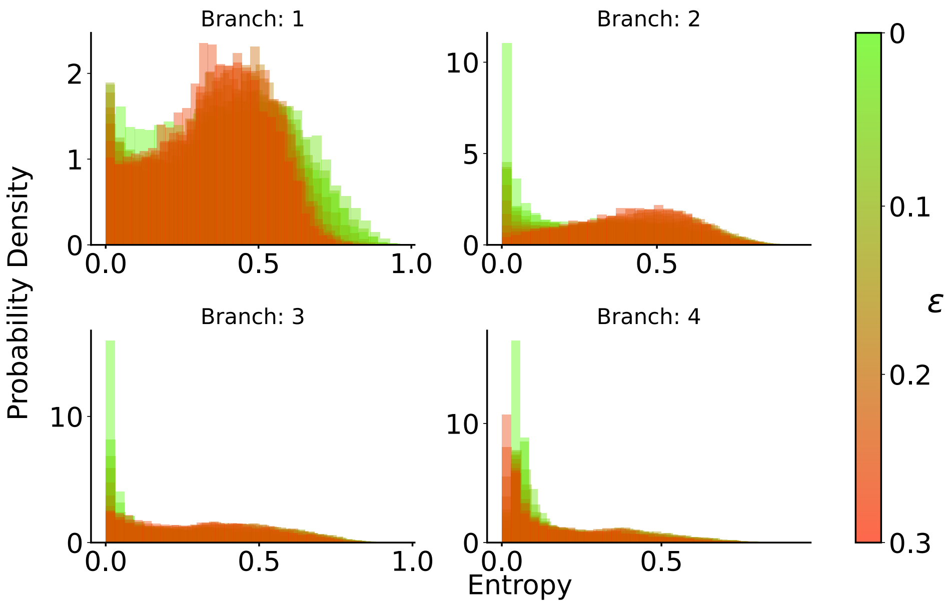


FIGURE 5.5: Histograms denoting the entropy probability densities for each branch on adversarial data.

As with the OOD data, the first branch is less susceptible to giving overconfident outputs, but there is also a lot of overlap with the adversarial inputs. Once again, this effect lessens later in the network.

Hence, we have shown in this section, in conventional early exit policies, entropy alone is insufficient for distinguishing an ID input from an unclassifiable one. We address the challenge of doing so in the next section.

5.1.2 Distinguishing Aleatoric and Epistemic Uncertainty in Branched Neural Networks

As we discussed earlier, epistemic uncertainty arises due to a distributional change in the input data, meaning the input is no longer in the range in which the model was trained. Validation and testing data should fall into this category to some degree. However, in balanced datasets, like the benchmarks seen in the field, this effect is not significant. Hence, for epistemic uncertainty we consider OOD datasets, those which are taken from different sources and have different class labels. We also consider adversarial inputs. As we show in this section, inputs which have been sufficiently perturbed have their distribution shifted in the latent space of the model.

When encountering OOD and adversarial inputs we have shown conventional deep learning models are susceptible to outputting similar predictions to those made on ID data. Hence, we wish to develop a second method which detects these unclassifiable inputs and allows us to classify them individually. First, we focus on OOD inputs.

Out of distribution detection Following recent work in Sun et al. (2022) we employ a KNN classifier in the penultimate layer embedding space. We find this is a particularly appropriate method to use, since it can be applied to any model without the need for additional optimisation. The principle of this is demonstrated in figure 5.2.

Consider the distribution in the embedding space, training inputs will be more tightly clustered than test inputs, but a well optimised model should position these distributions roughly about the same mean. OOD inputs, however, should be in a separate cluster. When translating this to KNN distances, the training inputs will have the lowest distances, followed closely by the test inputs, then the OOD inputs should cluster separately from the target dataset.

To test this principle, we train a baseline four exit branched neural network on a target dataset, recording the k^{th} nearest neighbour distances on the target data test set, and a collection of OOD datasets. Results with the CIFAR10 target dataset are shown in figure 5.6.

We find that there is clear separation between the datasets as the inputs progress through the model. However, in the early branches, this separation is minimal. Furthermore, in each branch there is a large amount of overlap. This separation increases further along the inference process and at the final exit the overlap is minimised. In some datasets, there is sufficient separation to allow for classification in the earlier branches. To examine this further, we analyse the receiver operator characteristic (ROC) of OOD detection on these branches using a KNN classifier. To generate this, we vary the position of the classification boundary between the distributions, defining this boundary according to percentiles of the validation distribution.

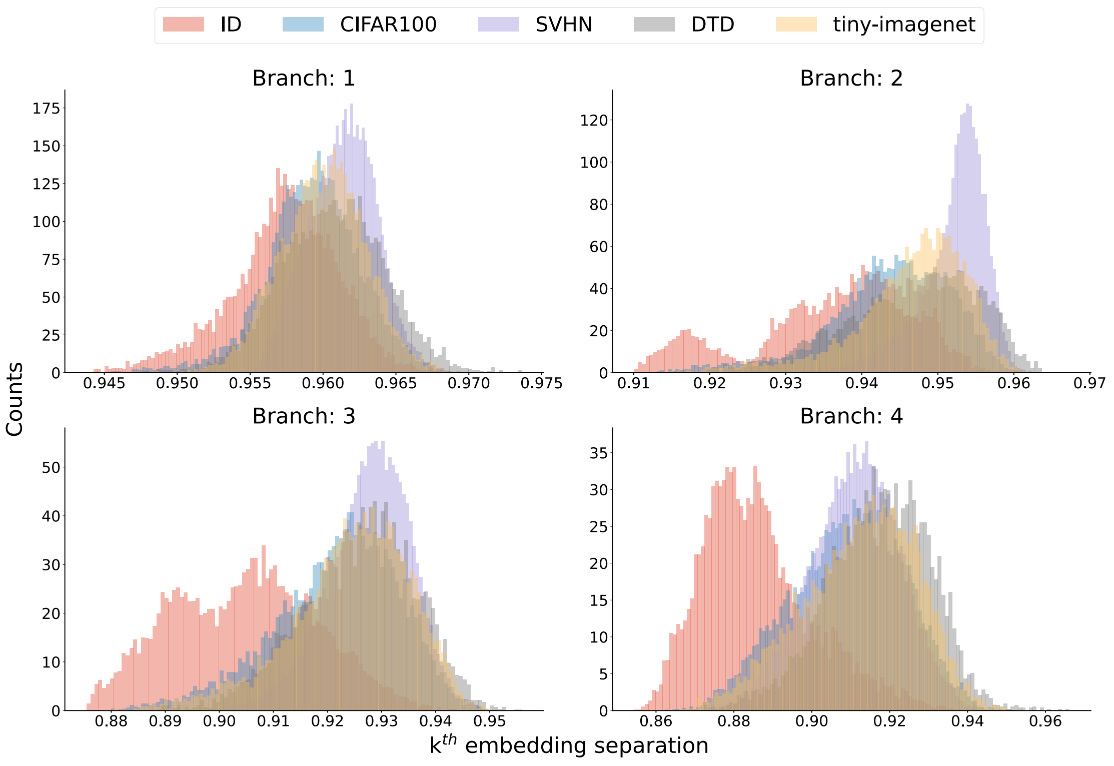


FIGURE 5.6: KNN embedding separation for in distribution (ID) data and various out of distribution datasets.

TABLE 5.1: Table showing the AUROC for each branch given different OOD datasets. Recent benchmarks are shown, where † denotes the benchmark which is implemented out-of-the-box, like ours. The mean and standard deviation (mean ± std) are taken over 5 runs.

	AUROC			
	Dataset			
Branch/Benchmark	CIFAR100	SVHN	DTD	Tiny-Imagenet
1	0.62±0.01	0.65±0.04	0.69±0.04	0.67±0.04
2	0.74±0.01	**0.95±0.01**	0.81±0.05	0.80±0.01
3	0.84±0.01	0.93±0.01	0.88±0.03	0.88±0.03
4	**0.85±0.01**	0.88±0.02	**0.90±0.02**	**0.89±0.01**
UDG Yang et al. (2021a)	0.90	0.93	0.94	0.93
ARPL+CS Chen et al. (2021b); Vojir et al. (2023)	0.89	0.91	0.91	0.89
RegMixup Pinto et al. (2022)	0.90	0.97	-	0.90
GROOD Vojir et al. (2023)	**0.97**	**0.99**	**0.99**	**0.96**
DNN Sun et al. (2022)†	-	0.95	0.95	-
DNN w/ CLSun et al. (2022)	-	**0.99**	**0.99**	-

As shown in figure 5.6, the later branches are more effective in separating the OOD inputs from the target dataset. This is reflected in the area under ROC (AUROC) for each branch, shown in table 5.1. In the final branch, there are competitive results, considering there is no additional optimisation taking place for OOD detection. However, on some datasets, namely SVHN, there are competitive AUROC results on the earlier branches. For most of the datasets tested, we find that identification performance increases further along the network. However, we do find that for the SVHN dataset, performance is maximised in the second branch.

These results suggest that branched neural networks, even when trained using conventional supervised methods, are capable of distinguishing their target dataset from other OOD datasets. However, when models are deployed into real-world situations, it is not only OOD inputs they might encounter. In some scenarios they will encounter adversarial attacks, designed to deliberately confuse the model. We consider this case in the next section.

Detecting adversarial attacks Adversarial perturbations in the inputs present a different challenge, since their effect on the output classification is non-binary. That is, there is a continuous range between the unperturbed inputs and those that are completely perturbed. Hence, some inputs, whilst being perturbed, are still able to be classified.

Therefore, rather than classifying the perturbed from the clean inputs, it is more pertinent to identify which inputs will be classified by the model and which inputs are unclassifiable.

To analyse their distribution in the embedding space, we follow a similar analysis to the previous section. We analyse the distributional shift of the perturbation of the CIFAR10 dataset in figure 5.7.

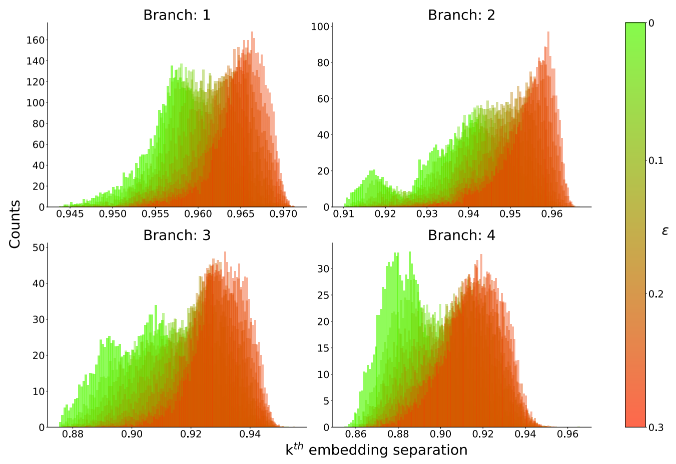


FIGURE 5.7: KNN embedding separation for various perturbation values of the target dataset, ranging from $e = 0 \rightarrow 0.3$. We represent increasing perturbation visually by varying the distribution colours from green to red.

We find that as the data becomes increasingly perturbed, its distribution in the embedding space separates from the clean data. Much like the OOD data in figure 5.6, the distributional shift is most evident in the later layers. However, there is a clear shift as perturbation increases in all branches.

To further examine this, we use the same technique as in the previous section to determine the ROC curve of the model at each branch, for each perturbation amount. We show these results in figure 5.8, and we present AUROC values in table 5.2. We find that the later branches are more successful in identifying the lower perturbation values. However, as the perturbation increases, the AUROC performance increases in the earlier branches. At the greatest value of ϵ, we find that the first branch distinguishes the adversarial samples most successfully.

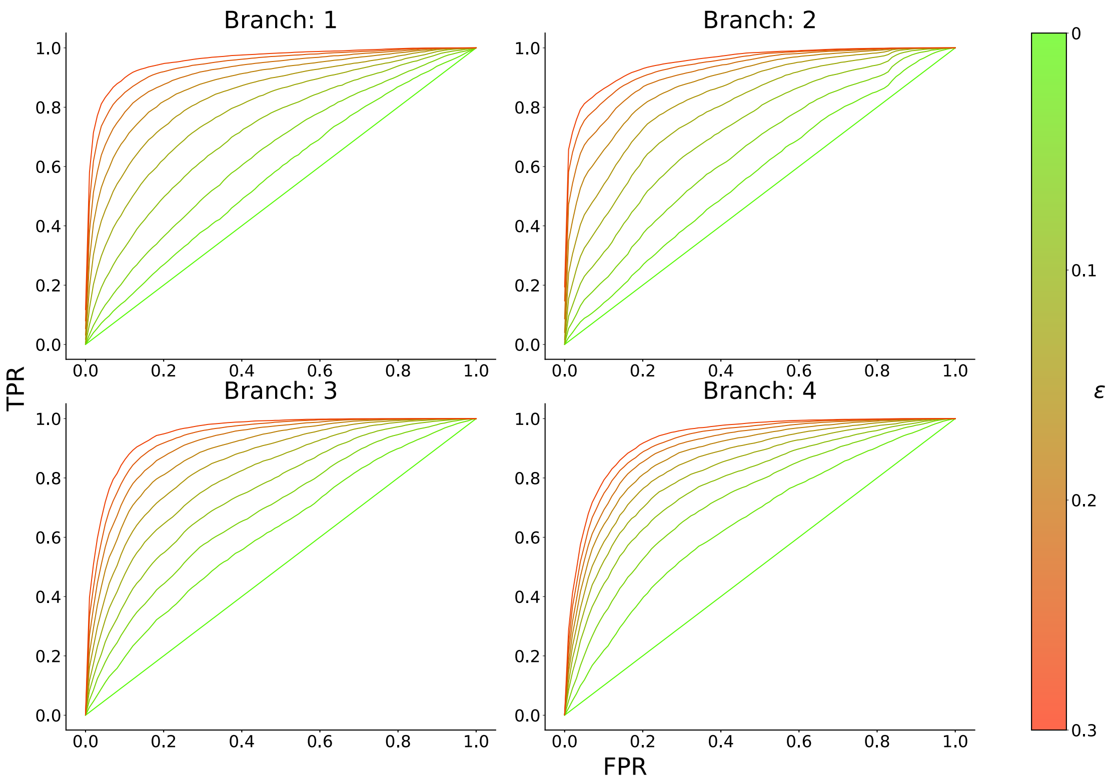


FIGURE 5.8: ROC curves for each branch on adversarial samples for various perturbation values of the target dataset. In all cases, the x and y axes denote the false positive rate (FPR), and true positive rate (TPR), respectively.

As expected, we find that the KNN classifier does not work at low values of ϵ. However, at these values the performance of the model is not prohibitively poor and the performance at greater values of ϵ is competitive. Hence, by treating significantly perturbed values like OOD inputs, there is a potential to detect and reject the classification of such inputs. We consider this scenario and the cost benefits it can achieve in the next section.

TABLE 5.2: Table showing the AUROC for each branch given different perturbation values for adversarial data. The best performing branch for each dataset is highlighted for each value of ϵ. We omit the results for ϵ=0, since the data is unperturbed in this range. We use the benchmark results reported in Zhang et al. (2022).

Branch	AUROC								
	ϵ								
	0.03	0.07	0.1	0.13	0.17	0.2	0.23	0.27	0.3
1	0.57±0.00	0.66±0.01	0.74±0.02	**0.82±0.03**	**0.87±0.03**	**0.90±0.03**	**0.93±0.02**	**0.95±0.02**	**0.96±0.01**
2	0.58±0.00	0.64±0.01	0.71±0.01	0.77±0.02	0.83±0.02	0.87±0.02	0.90±0.02	0.92±0.02	0.93±0.03
3	**0.63±0.01**	0.70±0.01	**0.76±0.01**	0.80±0.01	0.85±0.01	0.88±0.01	0.90±0.02	0.91±0.03	0.91±0.05
4	**0.63±0.01**	**0.71±0.01**	**0.76±0.01**	0.80±0.01	0.83±0.01	0.85±0.01	0.87±0.02	0.88±0.03	0.88±0.04
SPBAS Katzir and Elovici (2019)	-	-	-	-	-	-	-	-	0.79
ML + LOO Yang et al. (2020)	-	-	-	-	-	-	-	-	**1.00**
KD + BU Feinman et al. (2017)	-	-	-	-	-	-	-	-	0.90
LID Ma et al.	-	-	-	-	-	-	-	-	0.95
MAHA Lee et al. (2018)	-	-	-	-	-	-	-	-	**1.00**

5.1.3 Rejecting Classification to Save Power

We have so far established that branched neural networks can detect OOD inputs, as well as adversarial perturbations. We now consider the beneficial situation branched neural networks allow for using epistemic uncertainty to minimise resource usage. To accomplish this, we propose a multi-step modification to the conventional exit policy.

First, the aleatoric uncertainty is handled, ID domain inputs will be processed normally, this will catch any opportunities to perform an early-exit classification. For this, we employ an entropic decision policy on the softmax output of each classification branch. We impose a classification threshold on the entropy of this output, defined as α. If this condition is not met, we move to the epistemic uncertainty quantification phase, designed to classify inputs outside of the training distribution using the KNN distance. If this distance exceeds a certain threshold, δ, then an **early-exit reject** can take place. If neither condition is met, inference moves to the next branch.

Much like work in Sun et al. (2022), we define our KNN threshold using the validation data distribution on the training set and use the 50th nearest neighbour, as this empirically produces the best results. We model it as a Gaussian distribution, and take thresholds on the KNN values based on the percentiles of this distribution. That is, we define a classification boundary that contains δ of the validation distribution. We vary this value between 1.0 and 0.9 at selected intervals.

To test OOD detection we introduce a mixed dataset, D, which contains in equal parts ID inputs and OOD inputs. We produce these for four OOD datasets, Describable Textures Dataset[1] (DTD), CIFAR100, Street View House Numbers (SVHN), and Tiny-ImageNet. We then pass the dataset to our branched neural network. Our *distribution-aware* early exiting process is detailed in algorithm 4.

Here M refers to the collected train embeddings, `get_knn()` a function returning the k nearest neighbours, and `k` the value of k. Following work in Sun et al. (2022), we find k=50 empirically performs well whilst reducing the overhead for KNN reading. We also analyse the sensitivity of classification as k is changed in the KNN algorithm. We change it from 1 to 5, and then to 150 at increments of 5. We find that AUROC does not increase significantly as we do this, we present results in figure 5.9. Hence, we introduce two conditions for an early exit, increasing the opportunity for early exits to take place. To understand the performance of models using such inference methods, we analyse their operating ranges, much like the work in chapter 4.

We vary the entropy threshold for classification, α, from 0 to the maximum value, $\log(C)$. This is to understand the operating points at which the model can work in the accuracy-power space, where accuracy refers to that on the ID data and power the average MAC operations at that particular threshold.

[1] Since DTD only has $\sim$ 5000 input patterns, we use all of the inputs of this dataset for OOD detection.

Algorithm 4 Distibution-aware early exiting algorithm

```
1:  procedure EARLY EXITING INFERENCE WITH REJECT($f, X, \alpha, \delta$, k)
2:      Load model: $f$
3:      Read data, Thresholds, Embeddings, and k: $X, (\alpha, \delta), M$, k

4:      for $x \in X$ do                                   ▷ For all inputs in the test set
5:          for branch $b \in B$ do                        ▷ For all branches in model
6:              $m_b = f_{b-1}(x)$                         ▷ Get embeddings
7:              $\hat{y}_b = \arg\max(f_b(x))$             ▷ Calculate branch prediction
8:              $k_b$ = get_knn($m_b, M$;k)                ▷ Calculate k nearest neighbour distance
9:              $\kappa_b$ = confidence($f_b(x)$)          ▷ Calculate confidence

10:             if $\kappa_b \geq \alpha$ then             ▷ Check classification condition met
11:                 return $\hat{y}_b$                     ▷ Return classification if confident
12:             end if

13:             if $k_b \geq \delta$ then                  ▷ Check rejection condition met
14:                 return $\hat{y}_b = -1$                ▷ Return nil classification
15:             end if

16:         end for                                        ▷ Full inference used without early exit
17:         $\kappa_B$ = confidence($f_B(x)$)              ▷ Calculate final confidence for completeness
18:         $\hat{y}_B = \arg\max(f_B(x))$
19:         return $\hat{y}_B$                             ▷ Return final output
20:     end for                                            ▷ Move to next input

21: end procedure
```

We also scale the KNN distance threshold, α, depending on the entropy threshold. We do this between the KNN distance corresponding to the selected detection percentile and that corresponding to the 100th percentile. This is so that as the classification threshold becomes more stringent and conservative, the KNN rejection threshold does the same. In practice we normalise both α and δ.

The operating range for OOD detection and adversarial detection are shown in figures 5.10 and 5.11 respectively. We show the ID accuracy against the average power usage, for a given entropy exit threshold. A number of minimum values for δ are shown. We show a conventional exit policy in red, which without the early exit reject recourse is forced to process all inputs. Like in the previous section we use equation 4.5 to define our confidence interval.

We find that modest gains can be achieved with OOD CIFAR100 inputs, where the exit policy can improve accuracy across the entire power range of the model, and operate at lower power ranges when the inference policy is at its lowest threshold. The algorithm performs better on the other datasets. There is a clear distinction between each inference policy threshold, across most of the MACs range. We find that the most gains can be achieved on SVHN, with a δ of 0.99, there is no drop in ID accuracy for large power savings.

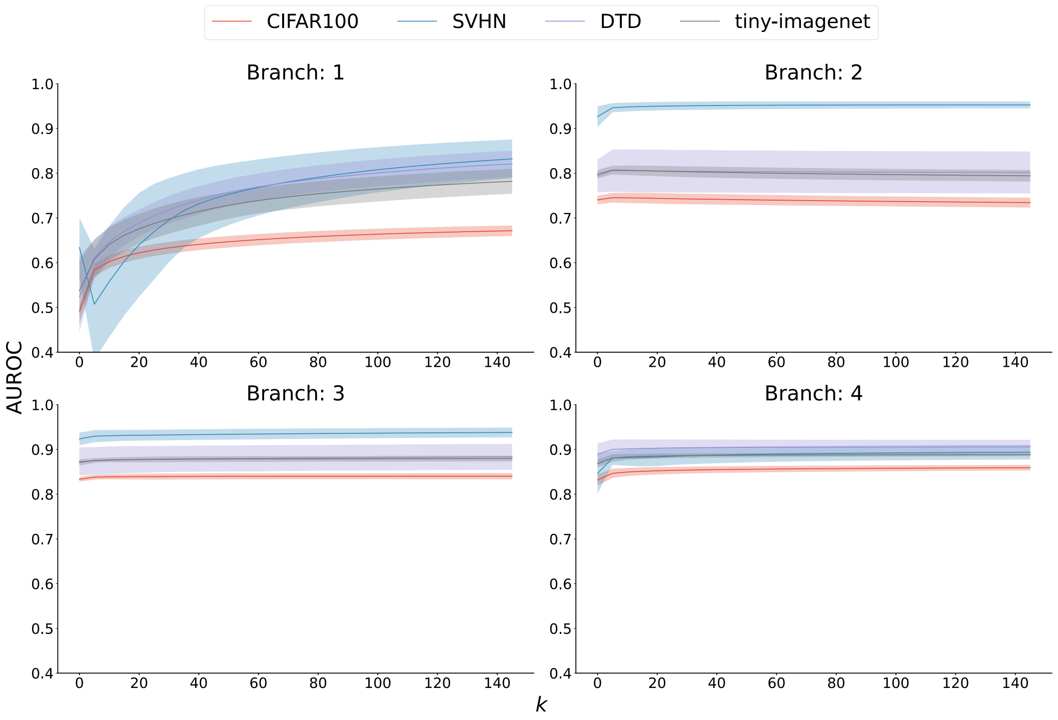


FIGURE 5.9: AUROC results shown on the y axis as k is increased, shown on the x axis. In all plots the shaded area denotes the standard deviation across 3 runs of the experiment.

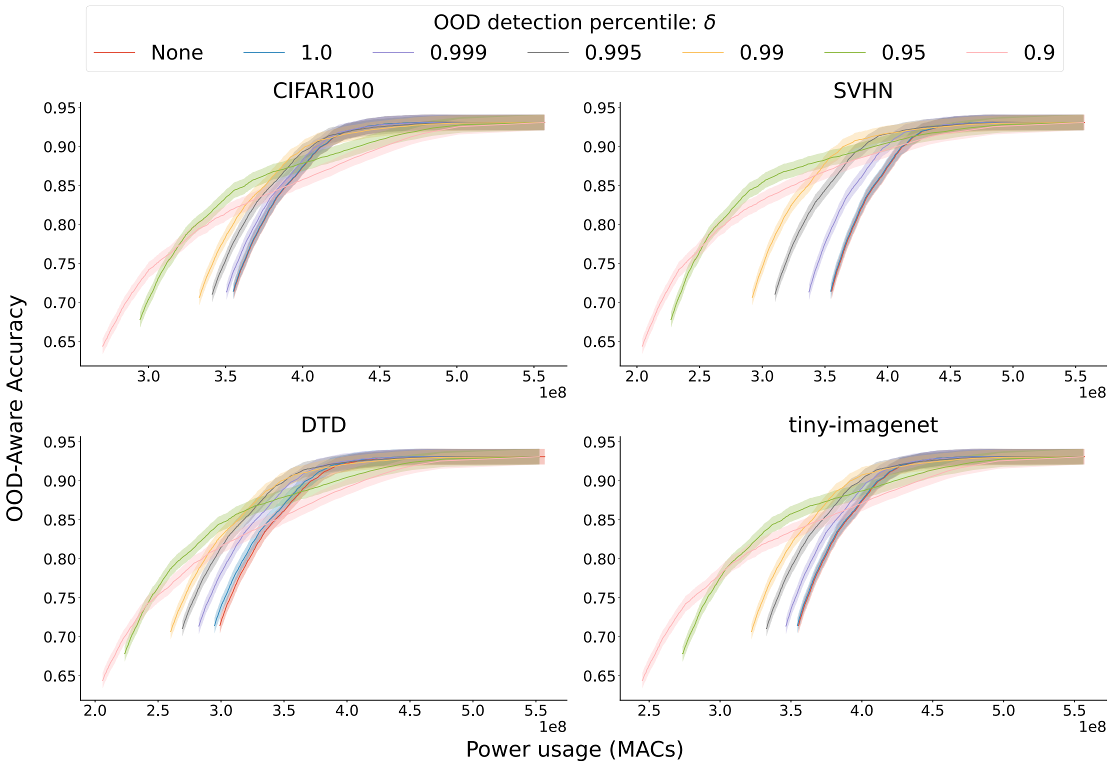


FIGURE 5.10: Operating range of the distribution-aware early exiting algorithm on various OOD datasets. In all cases the x axis denotes ID accuracy, y axis denotes average multiply accumulate operations (MACs).

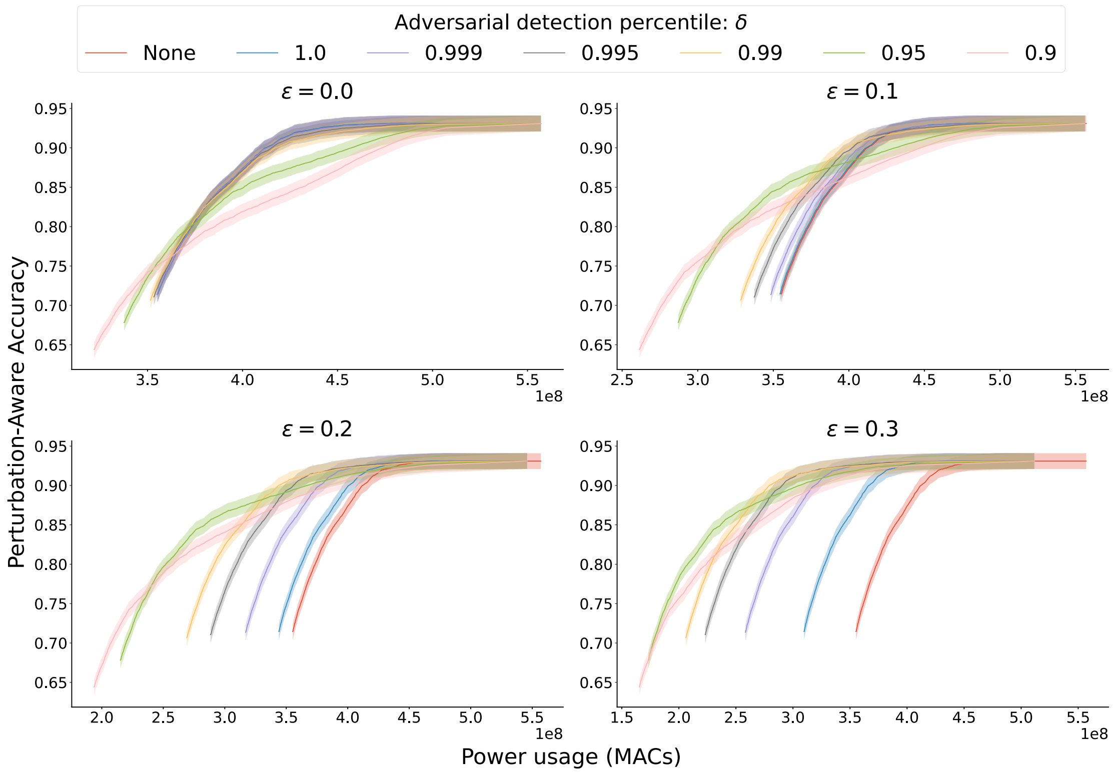


FIGURE 5.11: Operating range of distribution-aware early exiting algorithm on adversarial inputs with a selection of ϵ values.

On adversarial data, we find that the model has difficulty recognising the adversarial inputs, at small values of ϵ, and hence early exiting gains are minimal. However, as ϵ increases, the early exiting gains become more prevalent. At ϵ=0.3 the δ=0.99 exit policy makes significant power reduction gains across the entire accuracy range. In both tests the δ of 0.95 and 0.90 policies have greater power savings, but with significant drops in accuracy.

These findings will allow the model to operate much more stringent ID classification policies, when compared to the conventional early exiting method which wastes resources on the OOD and adversarial data. When the model is rejecting at most 1% of the ID samples ($\delta \geq 0.99$), we do not see this reflected in the ID performance readings. Hence, any power gain is effectively *free* compared to the conventional exit policy.

Since the power savings inherently depend on the ratio of ID inputs to unclassifiable inputs in D, it is more informative to first consider the detection accuracy. This is shown for OOD inputs in table 5.3.

We find, as the figures in section 5.1.3 suggest, that the optimal threshold depends on the allowable ID accuracy drop. However, $\delta = 0.99$ operates well with little drop in performance. We define a *zero loss* row, which denotes either a threshold of $\delta = 1.0$ or 100% relative ID accuracy, detection performance is limited without a small compromise in ID performance.

TABLE 5.3: OOD detection results for various OOD datasets. Relative ID accuracy is shown, and we enter results for a number of different detection percentile thresholds.

		OOD Detection (%)			
Relative ID Performance (%)	Threshold percentile	Dataset			
		CIFAR100	SVHN	DTD	Tiny-Imagenet
Zero Loss	1.0	0.36±0.11	0.44±0.29	2.64±1.24	0.77±0.21
99	0.999	3.53±0.72	6.12±3.12	13.29±3.19	6.59±1.06
	0.995	10.09±0.53	19.55±4.81	27.06±3.19	17.12±0.92
	0.99	**13.38±1.17**	**26.66±5.58**	**32.79±3.19**	**22.05±1.18**
	0.95	12.70±0.62	25.59±5.43	29.99±2.55	20.68±0.71
	0.90	10.53±0.19	21.02±3.92	21.94±0.91	16.05±0.67
95	0.999	4.40±0.77	7.89±3.12	15.59±3.15	8.22±1.14
	0.995	13.80±1.17	27.53±5.99	33.30±3.54	22.65±1.26
	0.99	21.79±0.75	43.97±7.51	45.43±3.45	33.36±1.15
	0.95	**42.34±0.93**	**74.11±5.87**	**66.80±2.70**	**57.21±2.36**
	0.90	40.90±1.22	72.58±5.89	65.49±3.24	55.75±2.49
90	0.999	4.77±0.73	8.64±3.25	16.46±3.12	8.82±1.09
	0.995	15.08±1.12	30.17±6.02	35.20±3.59	24.45±1.37
	0.99	23.61±0.90	47.43±7.92	46.87±3.54	35.70±1.06
	0.95	54.43±0.93	85.37±3.36	76.86±3.18	68.97±1.62
	0.90	**61.95±0.69**	**90.42±2.18**	**82.23±2.53**	**75.62±1.32**

TABLE 5.4: Adversarial detection results for selected thresholds. We show results for a variety of relative ID accuracies.

		Adversarial Detection Performance (%)			
Relative ID Performance (%)	Threshold percentile	ϵ			
		0.0	0.1	0.2	0.3
Zero Loss	1.0	93.26±0.29	30.41±1.90	19.81±3.32	23.40±8.90
99	0.999	93.36±0.30	33.37±2.19	32.57±8.19	43.48±16.57
	0.995	93.59±0.30	38.18±2.23	45.64±8.66	60.63±14.24
	0.99	**93.74±0.29**	**40.54±2.04**	**50.85±8.66**	**66.79±12.96**
	0.95	93.72±0.29	39.77±1.96	47.51±7.65	61.52±10.93
	0.90	93.68±0.29	37.69±1.59	38.34±4.97	47.47±7.83
95	0.999	93.39±0.30	34.14±2.39	35.06±8.88	46.68±16.78
	0.995	93.75±0.29	40.79±2.20	51.37±8.67	67.23±13.16
	0.99	94.12±0.29	45.83±1.88	60.47±7.38	76.79±9.34
	0.95	**95.30±0.27**	**58.47±2.24**	**76.93±5.32**	**90.71±3.39**
	0.90	95.21±0.25	57.70±2.10	75.96±5.52	90.11±3.72
90	0.999	93.40±0.29	34.36±2.38	35.68±8.95	47.34±16.79
	0.995	93.81±0.29	41.52±2.21	52.75±8.69	68.79±12.71
	0.99	94.22±0.27	46.97±1.97	62.22±7.09	78.51±8.53
	0.95	96.20±0.17	66.26±1.81	84.37±3.66	94.72±2.18
	0.90	**96.88±0.23**	**71.15±1.93**	**87.96±2.88**	**96.41±1.51**

We find the most liberal exiting policies allow for up to 90% of the OOD samples to be detected. Whilst a relative performance drop of 1% allows for 10-30% of the OOD inputs to be detected, depending on the dataset. Table 5.4 shows similar results for adversarial data.

Since adversarial perturbation is not guaranteed to result in an incorrect classification, like OOD data, rather than denoting the raw detection accuracy, we detail the inference accuracy. That is, we allow the KNN classifier to predict the correctness of the output, as opposed to the origin of the input.

We again find a δ=0.99 is the most effective. There is very little drop in relative performance at an ϵ of 0.0, which is likely due to the base classifier achieving ~95% accuracy. However,

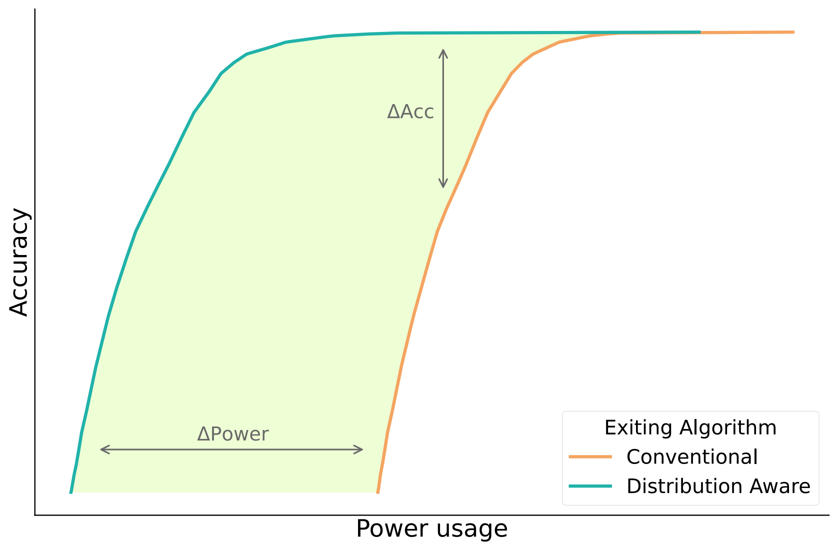


FIGURE 5.12: Accuracy change is denoted by the distance between the curves on the y axis, and power improvement by the difference on the x axis. The shaded area represents the improvement made over the conventional algorithm.

this performance drops as ϵ increases, before rising again. At higher perturbation levels, we find over 30% of these inputs can be classified, with no performance drop in the classifier. Allowing a 1% drop increases this detection accuracy to ~80%. Detection accuracy peaks at ~86% when relative performance is at 90%.

To better understand the performance gains this gives rise to, we analyse the power savings and accuracy benefits from the operating range curves shown in figures 5.10 and 5.11. We can record accuracy increases, and power increases, by taking the difference in these values from the figures. This is demonstrated in figure 5.12.

We record the maximum increases in each of these metrics for our exit policies. These values are shown for OOD data and adversarial data in tables 5.5 and 5.6, respectively.

For OOD data we find the highest peak accuracy gains are made in $\delta = 0.99$ and $\delta = 0.95$ thresholds, depending on the dataset. It should be noted, however, that from figure 5.10 it is evident the $\delta = 0.99$ policy can make similar accuracy gains without moving far from the maximum accuracy of the base model. Peak performance gains are achieved by the least stringent detection threshold of $\delta = 0.90$ but accuracy values show this is at the expense of decreased ID performance.

In the adversarial test, we find that the top performing detection threshold is dependent in some part on the test data. In this experiment, the $\delta = 0.995$ threshold is marginally better than the $\delta = 0.99$ threshold. Once again, in figure 5.11, it is evident the 0.99 policy makes similar accuracy gains without compromising maximum ID accuracy. Peak gains are again achieved by the least stringent detection threshold: $\delta = 0.90$, at the expense of decreased performance.

TABLE 5.5: Peak improvements made by the early exiting algorithm on $\mathcal{D}$, for a number of different OOD datasets, when compared to conventional early exiting algorithms. We compare a number of different detection thresholds, peak accuracy and power improvements are shown. We highlight the best performing values for each dataset.

		Peak Improvement (%)			
Detection Threshold	Improvement	Dataset			
		CIFAR100	SVHN	DTD	Tiny-Imagenet
1.0	Acc	0.18±0.01	0.23±0.19	1.00±0.52	0.45±0.18
	Power	0.08±0.03	0.11±0.07	0.80±0.39	0.20±0.06
0.999	Acc	1.97±0.51	4.03±2.50	4.83±1.38	3.92±0.66
	Power	0.75±0.12	1.62±1.04	2.41±0.57	1.49±0.25
0.995	Acc	5.69±0.55	10.96±3.04	8.97±0.76	9.35±0.96
	Power	2.79±0.25	6.13±2.05	5.78±0.57	4.76±0.57
0.99	Acc	8.56±0.60	**13.91±2.27**	**10.74±0.50**	**11.87±0.94**
	Power	4.84±0.19	10.24±2.31	8.44±0.37	7.64±0.67
0.95	Acc	**9.04±1.60**	7.20±1.68	9.06±1.59	8.24±2.04
	Power	14.77±0.40	24.43±1.47	18.52±0.59	19.62±1.20
0.90	Acc	-0.75±2.24	-4.80±2.36	0.89±2.03	-1.81±1.66
	Power	**22.67±0.66**	**31.44±1.91**	**25.30±0.63**	**27.72±1.26**

TABLE 5.6: Peak improvements made by the early exiting algorithm on $\mathcal{D}$, for a number of different ϵ values, when compared to conventional early exiting algorithms. We compare a number of different detection thresholds, peak accuracy and power improvements are shown.

		Peak Improvement (%)			
Detection Threshold	Improvement	ϵ			
		0.0	0.1	0.2	0.3
1.0	Acc	0.00±0.00	0.66±0.34	5.31±3.04	9.24±5.39
	Power	0.00±0.00	0.31±0.15	3.07±1.85	6.99±4.48
0.999	Acc	0.16±0.01	4.14±1.77	12.64±3.45	15.44±4.10
	Power	0.14±0.00	1.61±0.75	7.63±3.73	13.06±5.82
0.995	Acc	0.62±0.08	8.67±1.28	**15.90±1.83**	**16.52±1.95**
	Power	0.70±0.01	4.44±1.10	14.23±4.05	21.78±5.13
0.99	Acc	1.52±0.14	**10.88±1.12**	15.46±1.83	15.11±2.01
	Power	1.37±0.01	6.87±1.18	18.22±3.68	26.11±3.95
0.95	Acc	**3.34±0.36**	8.61±1.98	5.90±2.64	4.16±1.74
	Power	6.42±0.04	17.51±1.35	30.01±2.58	35.74±1.69
0.90	Acc	1.05±1.00	-1.51±2.23	-6.75±1.49	-6.75±1.49
	Power	**12.20±0.11**	**25.46±1.51**	**36.39±1.75**	**40.33±0.84**

All recorded results were averaged over 5 runs of the experiment, each trained from random initialisations. Results using CIFAR100 as the ID dataset are also collected, instead treating CIFAR10 as OOD. We show AUROC results for the branched network on adversarial data, comparing against various benchmarks. This is shown in table 5.2. We find our method is most competitive at higher perturbation values.

AUROC readings are shown for OOD data below in table 5.7, compared with the benchmarks we could find in the literature. We again find that we get competitive results when this dataset is used as ID.

We take the same readings for the adversarial data in table 5.8. We find our algorithm gets competitive results at higher perturbation values.

TABLE 5.7: Table showing the AUROC for each branch given different OOD datasets. The best performing branch for each dataset is highlighted. Recent benchmarks are shown. The mean and standard deviation are taken over 5 runs, and CIFAR100 was used as the ID dataset.

Branch/Benchmark	AUROC			
	Dataset			
	CIFAR10	SVHN	DTD	Tiny-Imagenet
1	0.59±0.00	0.74±0.05	0.48±0.02	0.55±0.02
2	0.59±0.01	0.81±0.03	0.80±0.03	0.76±0.03
3	0.62±0.02	**0.91±0.02**	**0.90±0.01**	0.81±0.01
4	**0.75±0.01**	0.81±0.02	0.85±0.01	**0.83±0.01**
UDG Yang et al. (2021a)	0.76	0.88	0.80	0.77
RegMixup Pinto et al. (2022)	**0.81**	**0.89**	-	**0.83**

TABLE 5.8: Table showing the AUROC for each branch given different perturbation values for adversarial data, with a model trained on CIFAR100. The best performing branch for each dataset is highlighted for each value of ϵ. We omit the results for ϵ=0, since the data is unperturbed in this range. We use the benchmark results reported in Zhang et al. (2022).

Branch	AUROC								
	ϵ								
	0.03	0.07	0.1	0.13	0.17	0.2	0.23	0.27	0.3
1	0.54±0.01	0.57±0.01	0.58±0.01	0.56±0.03	0.54±0.01	0.53±0.02	0.53±0.02	0.52±0.03	0.53±0.04
2	0.58±0.00	**0.69**±0.01	**0.77±0.02**	**0.84±0.03**	**0.88±0.03**	0.91±0.04	0.93±0.04	0.94±0.04	0.95±0.04
3	0.58±0.01	0.67±0.01	0.75±0.01	0.82±0.01	**0.88±0.01**	**0.92±0.01**	**0.95±0.00**	**0.96±0.00**	**0.97±0.00**
4	**0.59±0.01**	0.68±0.01	0.75±0.01	0.81±0.01	0.85±0.01	0.88±0.01	0.91±0.02	0.92±0.02	0.93±0.02
SPBAS Katzir and Elovici (2019)	-	-	-	-	-	-	-	-	0.51
ML + LOO Yang et al. (2020)	-	-	-	-	-	-	-	-	**1.00**
KD + BU Feinman et al. (2017)	-	-	-	-	-	-	-	-	0.98
LID Ma et al.	-	-	-	-	-	-	-	-	1.00
MAHA Lee et al. (2018)	-	-	-	-	-	-	-	-	**1.00**

We also present the OOD detection accuracy in table 5.9. We find more modest detection accuracies with this ID dataset, however, there can still be high detection accuracy with a small drop in relative ID accuracy.

We repeat the experiment for adversarial data, instead allowing the algorithm to classify the correctness of the classification. These results are shown in table 5.10. We find the base model could have been better optimised on the target dataset, however we find similar findings to that when training on CIFAR10. Detection performance is maximised when the adversarial perturbation is maximised.

We record additional peak improvements, with CIFAR100 as the ID dataset, we show this in table 5.11. The improvements are lower when CIFAR100 is the ID dataset, however, we still find ∼20% power can be saved, or ∼15% accuracy improvements can be made. We generally found the more liberal exit policies performed slightly better in this experiment.

We take the same readings for the adversarial tests using the CIFAR100 dataset, these are shown in table 5.12. We again find that the more liberal exit policies perform better on this dataset. We find that a ∼24% accuracy improvement can be made over conventional exiting algorithms, or a ∼29% power improvement.

TABLE 5.9: OOD detection results for various OOD datasets, with CIFAR100 as ID. Relative ID accuracy is shown, and we enter results for a number of different detection percentile thresholds. We take the mean and standard deviation over 5 runs. That is, we record the results from 5 models trained from different random initialisations.

		OOD Detection (%)			
Relative ID Performance (%)	Threshold percentile	Dataset CIFAR10	SVHN	DTD	Tiny-Imagenet
ZeroLoss	1.0	0.09±0.08	0.08±0.04	3.19±1.60	0.39±0.29
99	0.999	0.39±0.12	0.97±0.46	9.95±1.58	1.75±0.40
	0.995	1.53±0.12	5.43±2.90	20.36±1.81	5.24±0.67
	0.99	2.19±0.31	**7.35±3.43**	**24.14±2.37**	**7.09±1.11**
	0.95	2.41±0.30	8.30±3.86	21.55±1.46	7.31±0.94
	0.90	**2.42±0.21**	7.47±2.70	16.46±1.35	6.59±0.63
95	0.999	0.58±0.15	1.91±1.18	12.53±1.69	2.44±0.56
	0.995	2.59±0.24	9.09±4.25	26.66±1.12	8.26±0.71
	0.99	4.63±0.41	15.54±6.61	34.68±1.11	13.37±1.10
	0.95	**11.87±0.74**	**34.38±7.97**	51.79±1.72	**27.54±1.54**
	0.90	11.67±0.66	34.06±8.39	**55.39±1.13**	27.28±1.54
90	0.999	0.65±0.14	2.43±1.59	13.60±1.56	2.76±0.56
	0.995	3.06±0.33	10.63±4.84	28.59±1.39	9.51±0.94
	0.99	5.50±0.50	18.16±7.30	37.47±1.19	15.43±1.30
	0.95	19.13±1.01	49.24±8.68	62.67±1.41	39.30±1.68
	0.90	**22.90±1.50**	**55.76±8.28**	**66.93±1.48**	**44.47±2.29**

TABLE 5.10: Adversarial detection results for selected thresholds. We show results for a variety of relative ID accuracies. We take the mean and standard deviation over 3 runs, the ID data the model is trained on is CIFAR100.

		Adversarial Detection Performance (%)			
Relative ID Performance (%)	Threshold percentile	ϵ 0.0	0.1	0.2	0.3
Zero Loss	1.0	72.87±0.24	14.19±0.62	6.63±0.60	7.15±2.06
99	0.999	72.98±0.23	14.87±0.60	12.42±2.43	20.66±6.96
	0.995	73.26±0.22	17.18±0.71	26.17±6.13	43.53±9.88
	0.99	73.45±0.23	18.43±0.96	**31.35±7.32**	**51.04±11.35**
	0.95	**73.49±0.27**	**18.77±0.76**	29.59±4.96	44.34±7.79
	0.90	**73.49±0.25**	18.54±0.72	23.56±3.14	31.65±4.63
95	0.999	73.03±0.24	15.27±0.67	15.38±2.89	26.69±8.09
	0.995	73.56±0.25	19.25±0.90	34.59±6.41	55.52±10.26
	0.99	74.08±0.26	23.06±1.12	46.66±7.15	68.79±9.62
	0.95	**75.90±0.40**	**34.21±1.94**	**66.83±7.91**	**85.62±6.96**
	0.90	75.85±0.41	34.02±1.86	66.57±7.60	85.36±6.71
90	0.999	73.05±0.23	15.45±0.73	16.67±2.89	28.61±8.22
	0.995	73.67±0.27	20.18±1.06	37.70±6.76	59.35±10.52
	0.99	74.34±0.23	24.70±1.27	50.44±7.52	72.58±9.07
	0.95	77.58±0.39	43.63±2.34	77.03±6.65	91.72±4.52
	0.90	**78.52±0.41**	**47.71±2.72**	**80.25±6.40**	**93.30±3.96**

TABLE 5.11: Peak improvements made by the early exiting algorithm on $\mathscr{D}$, for a number of different OOD datasets, when compared to conventional early exiting algorithm. We compare a number of different detection thresholds, peak accuracy and power improvements are shown. We highlight the best performing values for each dataset. We take the mean and standard deviation over 5 runs of the experiment, CIFAR100 is used as the ID dataset.

		Peak Improvement (%)			
Detection Threshold	Improvement	Dataset			
		CIFAR10	SVHN	DTD	Tiny-Imagenet
1.0	Acc	0.01±0.01	0.00±0.00	0.31±0.15	0.02±0.03
	Power	0.02±0.01	0.01±0.01	0.40±0.18	0.02±0.02
0.999	Acc	0.23±0.08	0.57±0.53	2.51±0.22	0.77±0.22
	Power	0.17±0.01	0.41±0.26	1.66±0.17	0.41±0.05
0.995	Acc	0.97±0.03	3.49±1.60	6.26±0.22	3.35±0.70
	Power	0.91±0.05	1.84±0.71	4.10±0.37	1.89±0.29
0.99	Acc	1.99±0.32	5.34±2.45	8.31±0.55	5.92±0.99
	Power	1.74±0.06	3.31±1.01	5.99±0.32	3.42±0.37
0.95	Acc	**14.74±1.94**	9.06±1.59	**12.54±0.88**	13.84±0.72
	Power	7.86±0.23	12.63±1.23	14.14±0.63	12.04±0.79
0.90	Acc	9.92±0.33	**16.76±2.16**	11.81±0.46	**14.81±0.49**
	Power	**22.55±1.88**	**25.30±0.63**	**20.64±0.69**	**19.73±0.91**

TABLE 5.12: Peak improvements made by the early exiting algorithm on $\mathscr{D}$, for a number of different ϵ values, when compared to conventional early exiting algorithms. We compare a number of different detection thresholds, peak accuracy and power improvements are shown. We take the mean and standard deviation over 5 runs, trained on CIFAR100 dataset.

		Peak Improvement (%)			
Detection Threshold	Improvement	ϵ			
		0.0	0.1	0.2	0.3
1.0	Acc	0.00±0.00	0.02±0.03	0.17±0.21	0.61±0.43
	Power	0.00±0.00	0.02±0.02	0.14±0.12	0.48±0.36
0.999	Acc	0.08±0.02	0.46±0.16	4.29±1.22	8.35±2.43
	Power	0.14±0.00	0.32±0.07	1.97±0.48	3.98±1.09
0.995	Acc	0.47±0.03	2.66±0.60	12.73±3.62	19.02±3.70
	Power	0.68±0.01	1.60±0.28	6.81±1.76	10.86±2.40
0.99	Acc	1.14±0.29	4.89±1.22	17.49±3.08	22.84±3.01
	Power	1.33±0.01	3.04±0.37	10.14±2.04	14.66±2.62
0.95	Acc	4.66±0.62	13.00±1.22	**21.73±1.73**	**23.67±0.81**
	Power	6.23±0.05	11.57±1.06	20.21±2.71	23.40±2.87
0.90	Acc	**6.47±0.04**	**14.47±0.49**	17.47±0.88	17.53±1.52
	Power	**11.88±0.09**	**19.31±1.18**	**25.91±2.31**	**27.95±2.29**

It is worth noting that these results, and those that follow in section 5.3, assume a classification cost which is negligible when compared to that of the neural network backbone. The cost of the KNN classifier scales proportionally to the dimensionality of the input, the number of samples in the training set, and in the simple case, the number of neighbours, k. The most prohibitive of these in our case being the dimensionality (between 64 and 512 depending on the branch) and the number of samples (5000 for CIFAR datasets). Hence in this prototypical setting, using a KNN classifier may incur a performance impact which is not reflected in these results. However, compressing the embedding space and selecting representative samples in the validation set for use in the KNN algorithm, will reduce these costs. Furthermore, we encourage future work to introduce improved OOD detection mechanisms to this setting.

We now shift our attention towards forms of early exit rejection which can be used in the absence of unclassifiable data. That is, a form of rejection which can be applied within the class distribution of the training data.

5.2 Using Hierarchies in the Label Structure

In the previous section information regarding the embedding space was considered in the inference process. Here, we consider how additional information in the labels can be incorporated in the inference process. For example, two animals are likely to be closer to one another in the embedding space and therefore confused for one another than an animal and a vehicle. Hence, initial predictions can utilise the intra-label relationships to better understand the likelihood of the final label being of a given category (e.g. animals or vehicles).

This section once again addresses the literature gap of *Exploring Progressive Intelligence Systems, Introducing New Early Exiting Mechanisms.* In doing so, we make the following contributions:

Branched Neural Networks Intrinsically Learn Hierarchies We revisit hierarchically labelled datasets, now focusing on the tangible benefits of them. That is, we look at the accuracy change when training with the label hierarchy vs when just using the fine-grained labels.

Introduce Class-Aware Early Exiting We introduce a new early exit policy which operates in a test environment where some classes within the trained distribution take precedence over others. This is adaptable at inference time and we report the performance gains that can be achieved when performing class-aware early exiting.

Introduce a New Hierarchically Labelled Dataset We introduce a new variation of the tiered-Imagenet dataset, collecting all of the inputs into a single dataset. Furthermore, we define new *coarser* grained labels, creating 3 classes: Animal, Object, Other.

First we consider hierarchical datasets within the field as possible starting points, then an inference process is shown which uses them at inference time.

5.2.1 Hierarchical Datasets

There are a number of datasets in the field, which have an inherent hierarchy. For example, the commonly used benchmark in the field CIFAR10/100 has an inherent label hierarchy.

Though the dataset does not explicitly have a hierarchical label structure, one is implied and can be developed using the ‘fine’ label names: [airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]. The inferred three-tier (coarse, middle, fine) CIFAR10 hierarchy is described in figure 5.13.

Since the dataset is balanced at the fine level with 6000 images for each class, the only imbalance is introduced at the coarse level of hierarchy, due to the animal class having six of the ten fine-grained classes.

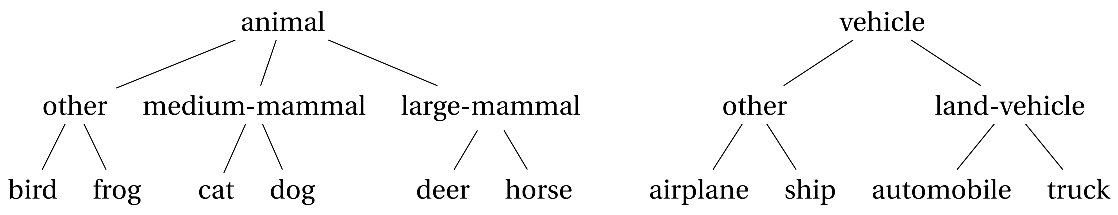


FIGURE 5.13: Three-tier hierarchy of CIFAR10. On the upper row we report the two coarse classes, while at the middle and last row we have the middle and fine levels of granularity, respectively.

There is also a hierarchy in the CIFAR100 dataset, at the coarse level this is defined by the dataset, with 100 fine-level classes, and 20 coarse-level classes. In a similar fashion as above, we can introduce a further coarse class collecting the dataset into coarser classes. Due to the small size of the images and the low sample count, initial experiments can be developed using these datasets, before moving onto the larger and more representative datasets.

There are a multitude of other datasets in the field which have a well defined, and more importantly, explicitly labelled hierarchy, we discuss a number of them below.

***tiered*ImageNet** This is a subset of the ImageNet dataset, which groups classes into broad categories corresponding to higher-level nodes in the ImageNet hierarchy. It is generated using the ImageNet dataset, where inputs are sampled from certain categories and down-sampled, much like *mini*ImageNet. There are utilities available on Github which can help generate the dataset. All inputs are of equal size, and there is no class imbalance at the fine label level. This means the dataset will be quicker to train on, and that class imbalanced can be manufactured in a more controlled fashion. (Ren et al., 2018)

iNaturalist This is a species classification dataset, which uses the natural hierarchy of animal/plant species to arrange its classes. It can be downloaded freely from the its website and has bounding boxes as well as classification labels. There is a large amount of class imbalance in this dataset, but it is a rich dataset with lots of classes. The imbalance is present both at the fine-grained level, and the coarser grained level. This will make it a suitable dataset to test the impact of limited amounts of data on certain classes. For out-of-distribution samples, there is ample opportunity to remove some of the classes in a given super-class, or remove a superclass entirely, in order to test different levels of robustness to out-of-distribution data.

Furthermore, the underrepresented classes can be used to organically analyse the performance a model can achieve on classes with limited examples. (Van Horn et al., 2018)

Caltech256 This is an old object detection dataset, which has an inherent class hierarchy. It has more levels of hierarchy than other datasets, but fewer instances in total. (Griffin et al., 2007)

CompCars A car classification dataset which splits its inputs into manufacturers, models, car types, and also year. Presenting a versatile and complex classification task. The images are both stock images, and CCTV captured images of the vehicles. There is class imbalance present at all levels of hierarchy (Buzzelli and Segantin, 2021). This dataset was also revisited in a recent paper by Buzzelli and Segantin (2021), where they adapted the train-test split in the data so it was more representative of real-world data.

RPC Retail Product Classification dataset is a hierarchical dataset containing retail products for automatic classification when going through the checkout. There are bounding box scenes as well as single object images. (Wei et al., 2019)

FGVC-Aircraft Similar to the CompCars dataset, this is an aircraft dataset which splits its inputs into manufacturers, families, and variants. (Maji et al., 2013)

VegFru A fruit and vegetable classification dataset containing fruit and vegetables of a variety of different types. It can be split further into fru92 and veg200 containing 92 and 200 classes respectively. (Hou et al., 2017)

Below is a table breaking down each dataset, giving the number of classes, levels of hierarchy, number of instances, class imbalance, whether or not is has bounding boxes, and the input resolution if known.

Dataset	No. Classes	Levels	N	Class Imbalance	Bounding Boxes	Resolution
CIFAR10	10	1(3)	50,000	In coarse labels		32x32
CIFAR100	100	2(3)	60,000	In coarse labels		32x32
*tiered*ImageNet	608	2	779,165	In coarse labels		84x84
iNaturalist	5089	2	579,184	In all labels	Yes	mixed
Caltech256	256	3-6	30,607	In all labels		mixed
CompCars Dataset	4455	3	136,7275	In all labels	Yes	mixed
RPC	200	2	53,739	In all labels	Yes	mixed
FGVC-Aircraft	100	3	10,000	In coarse labels	Yes	not reported
Veg200	200	2	91,117	In all labels		not reported
Fru92	92	2	69,614	In all labels		not reported

TABLE 5.13: Comparison list of datasets and their various properties. In brackets we denote inferred values when adding levels of hierarchy.

Upon reviewing these datasets, the two most appropriate datasets to intitially test with are *tiered*ImageNet, and iNaturalist, as both of these datasets have a large number of classes, a large number of instances, and have a rich hierarchy.

Due to *tiered*ImageNet's consistent and relatively small resolution, this would be a good dataset to perform experiments with, that will produce more representative results. However, to obtain an understanding of how long each model will take to train and produce a proof of-concept, it may be more suitable to perform initial experiments on CIFAR datasets, namely CIFAR100 due to its slightly richer hierarchy.

5.2.2 Using Hierarchies at Inference Time

The additional information provided in hierarchical datasets inform the user and the system about how classes are organised. This can be used to help indicate what the likely final classification is going to be. For example, if at the first exit the model predicts animal classes such as 'dog', 'horse', 'deer', whilst the classification itself might not be confident, it can be inferred with high confidence that it is going to be an animal, and with reasonable confidence that it will be a large mammal. Conversely, this also means the system can rule out other groups of classes, such as vehicles and buildings. Without this information embedded into the inference method, it becomes more difficult for the machine learning system to rule out such classes. Or rather, it becomes easier to group noteworthy classes at run-time.

We produce a proof of concept inference system which performs early exiting when the probability of a target class is not high enough, we call this inference *class-rejection* inference. For now we can simply take the sum over the target class probabilities in the output distribution,

$$r_b = \sum_c^C p_c \, \phi_c \tag{5.2}$$

where ϕ refers to a one-hot encoded vector denoting the target classes, that is a vector is of length C where 1 denotes an important class and 0 otherwise. The probability of class c is denoted by p_c. We can also express this as an inner product,

$$r_b = \mathbf{P}^\intercal \Phi,$$

where the capitalised symbols refer to their lower-case equivalents in equation 5.2. Hence, $\mathbf{P} = \hat{y}_b$, the model output at branch b. Φ is the same one-hot encoded vector.

To incorporate this concept into an early exiting system we combine the class-based early exit rejection with the confidence-based early exit classification. This inference algorithm is described in algorithm 5.

Here `target_prob` refers to the target class probability which is used to determine how likely the final classification is to be among the target classes.

Algorithm 5 Class-reject early exiting algorithm

```
1: procedure EARLY EXITING INFERENCE WITH CLASS-REJECT($f, X, \alpha, \gamma$)
2:     Load model:   $f$
3:     Read data and Thresholds:   $X, (\alpha, \gamma)$

4:     for $x \in X$ do                                          ▷ For all inputs in the test set
5:         for branch $b \in B$ do                               ▷ For all branches in model
6:             $\hat{y}_b = \arg\max(f_b(x))$                    ▷ Calculate branch prediction
7:             $\kappa_b = \text{confidence}(f_b(x))$            ▷ Calculate classification confidence
8:             $r_b = \texttt{target\_prob}(f_b(x))$             ▷ Calculate target class probability

9:             if $\kappa_b \geq \alpha$ then                    ▷ Check classification condition met
10:                return $\hat{y}_b$                            ▷ Return classification if confident
11:            end if

12:            if $r_b \leq \gamma$ then                         ▷ Check rejection condition met
13:                return $\hat{y}_b = -1$                       ▷ Return nil classification
14:            end if

15:        end for                                               ▷ Full inference used without early exit
16:        $\kappa_b = \text{confidence}(f_B(x))$                ▷ Calculate final confidence for completeness
17:        $\hat{y}_B = \arg\max(f_B(x))$
18:        return $\hat{y}_B$                                    ▷ Return final output
19:    end for                                                   ▷ Move to next input

20: end procedure
```

This algorithm is trialed on the CIFAR100 dataset, we initially use 3 coarse classes: Animal, Object, and Other. We vary (normalised) entropy and target class probabilty thresholds between 0 and 1 and record the power usage of the model. Much like in the previous section where we only record accuracy on the in distribution data, here we record accuracy on *target classes*. The results are shown in figure 5.14. We find that the models accuracy is quite low in the early branches for the animal classes, however in this subset there are more classes. In the 'other' subset the accuracy is much higher but it contains a smaller distribution of the classes. We find in all subsets that when the target classes fall into a hierarchical category the model can achieve greater accuracy earlier in the power-range. This improvement is gained through early exit rejection.

Since these distributions overlap completely in their operating range, we also analyse the performance of the algorithm using the AUC metric. For each target class group we normalise the accuracy range they operate in between 0 and 1, we then do the same for the power range, giving us a *normalised* AUC measurement. The results of this are shown in table 5.14. We find, as figure 5.14 suggests, that the greatest increase in performance is attained on the *other* class. However, significant performance improvement is also seen on the object class. In no case does the use of class-based rejection lower accuracy, or indeed the AUC measurement of the operating curve.

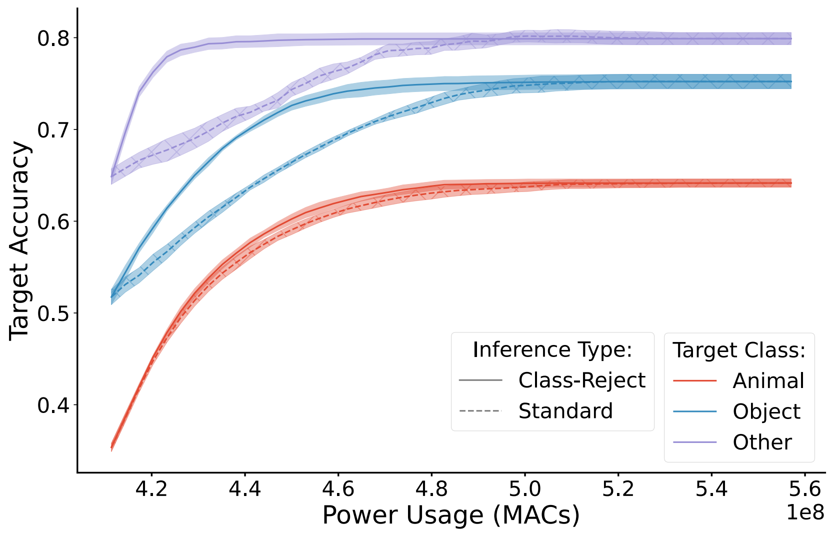


FIGURE 5.14: Operating ranges for class-based rejection in 3 branch ResNet. The model is trained on CIFAR100 data with 3 levels of hierarchy. Data is collected on the validation set over 5 runs, the shaded area denotes the standard deviation across these, and the data itself the mean.

TABLE 5.14: Normalised AUC measurements comparing class-reject inference to standard inference methods, when looking at target-class accuracy on the CIFAR100 dataset with hierarchically defined classes. Percentage increase is shown in the right-hand column, the class-reject inference performs best on the *other* class hierarchy relative to standard inference.

Target Class	Normalised AUC		↑ (%)
	Standard	Class-Reject	
Animal	0.8501±0.0006	0.8675±0.0001	2.04±0.06
Object	0.7698±0.0014	0.8688±0.0012	12.86±0.06
Other	0.7658±0.0003	0.953±0.0002	**24.43±0.08**

To further understand the generalisation of this technique we analyse the performance on the tieredImagenet dataset. These results are shown in figure 5.15 and AUC results are given in table 5.15. We once again find that the model performs worst on the animal classes in the initial branch and best in the other classes. However, the final performances on the classes are much closer to one another. Once again, there is significant improvement in target accuracy for a given power usage when class-based early exit rejection is enabled. The AUC results are broken down in table 5.15. The *other* class distribution is most responsive to the

TABLE 5.15: Normalised AUC measurements as in table 5.14, this time with the model trained on the tiered-ImageNet dataset.

Target Class	Normalised AUC		↑ (%)
	Standard	Class-Reject	
Animal	0.5731±0.0003	0.7962±0.0003	38.94±0.12
Object	0.6353±0.0011	0.7258±0.0007	14.25±0.07
Other	0.5738±0.0012	0.8845±0.0004	**54.16±0.39**

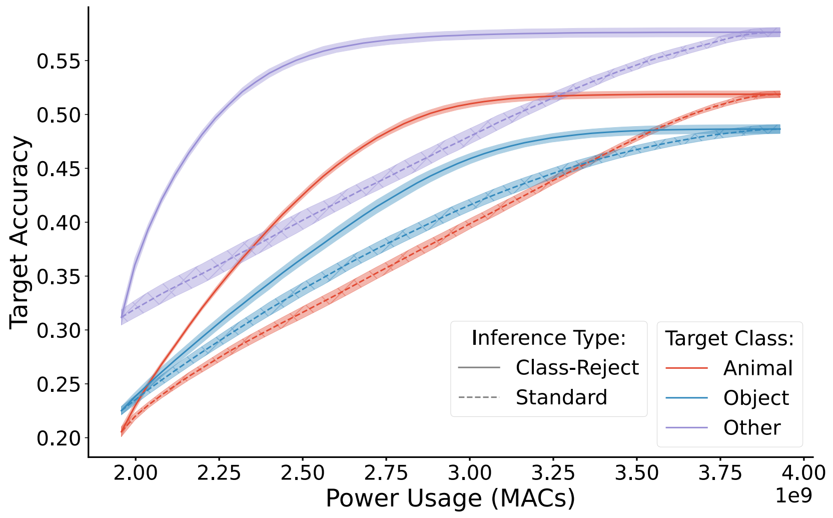


FIGURE 5.15: Operating ranges for class-based rejection in 3 branch ResNet. The model is trained on the tieredImagenet data with 3 levels of hierarchy. Data is collected on the validation set over 5 runs, the shaded area denotes the standard deviation across these, and the data itself the mean.

new inference technique, improving on AUC measurements by over 50%. In this dataset the animal class also performs well attaining ~40% improvement.

Hence, class-based rejection is an effective method of improving performance on a target class at inference time, given a specific inference budget. To understand the impact of the class hierarchy, we analyse random distributions of target classes on each dataset. First we use CIFAR100, these results are shown in table 5.16. We find that the utility of class-based

TABLE 5.16: Normalised AUC measurements comparing class-reject inference to standard inference methods on the CIFAR100 dataset with randomly assigned target classes of differing percentages of the class distribution. Percentage increase is shown in the right-hand column, class reject improvements are greater when there are fewer classes in the target-class distribution.

N Classes (%)	Normalised AUC		↑ (%)
	Standard	Class-Reject	
1.0	0.8129±0.0001	**0.9732±0.0**	**19.72±0.02**
5.0	0.8165±0.0011	0.924±0.0005	13.17±0.21
10.0	0.8142±0.0004	0.8878±0.0004	9.04±0.11
25.0	0.8113±0.001	0.8435±0.0003	3.97±0.09
50.0	0.8114±0.0007	0.8187±0.0005	0.89±0.02
75.0	0.8128±0.0007	0.813±0.0007	0.04±0.01
90.0	0.8114±0.0003	0.8114±0.0003	0.0±0.0
95.0	0.8116±0.0003	0.8116±0.0003	0.0±0.0
99.0	0.8118±0.0004	0.8118±0.0004	0.0±0.0

rejection quickly decreases at a target class percentage of 50% the improvements approach zero. The gains seen in the previous experiments are only seen at very low percentages, when a handful of classes are prioritised over the others in the 100 class dataset.

This suggests that not only is the class-based reject best used on hierarchically labelled datasets, but also that the models typically have an intrinsic understanding of the hierarchy. That is, when the model is predicting a specific class, the other classes predicted by the model are from within the same super-class, lending itself to class-based early exiting.

However, it is worth highlighting, even with random target class distributions, the early exiting algorithm never lowers the target class accuracy at a given power value. To further understand the impact of the hierarchy, we compare percentages similar to that of each target class. That is, for target class distributions in the hierarchy we sample the same percentage of random classes to compare performance increase when using hierarchically defined target classes. The results of this are shown in table 5.17. We find there is significant improvement

TABLE 5.17: Class-based reject AUC improvements comparing hierarchically defined classes against randomly assigned classes of the same class-counts. Difference denotes the absolute change in improvement, model is trained on CIFAR100.

Target Class (%)	Class-Based Improvement (%)		↑ Difference
	Random	Hierarchical	
Animal (50.0)	0.83±0.02	2.04±0.06	1.21±0.04
Object (40.0)	1.76±0.03	12.86±0.06	11.1±0.03
Other (10.0)	9.25±0.06	**24.43±0.08**	**15.18±0.02**

between the random distributions, this time measuring the absolute increase between percentages. We find that whilst the increase is minimal when comparing to the animal distribution, there is still an increase in accuracy across the operating range. The greatest increase is seen in the lowest percentage, which should be expected since for a balanced dataset there are larger proportion of inputs for which early exiting can take place.

We again repeat the random target experiment for the new tieredImagenet dataset. We find similar results, with all random sample distributions shown in table 5.18 and direct comparisons with hierarchical classes in table 5.19. Once again, the most performant early exiting

TABLE 5.18: Normalised AUC measurements comparing class-reject inference to standard inference methods with randomly assigned target classes as in table 5.16 this time on tiered-ImageNet.

N Classes (%)	Normalised AUC		↑ (%)
	Standard	Class-Reject	
1.0	0.5955±0.002	**0.9116±0.001**	**53.08±0.7**
5.0	0.5996±0.0005	0.7833±0.0005	30.63±0.19
10.0	0.6004±0.0	0.7205±0.0009	20.01±0.16
25.0	0.5986±0.0003	0.6507±0.0005	8.72±0.03
50.0	0.5994±0.0011	0.6016±0.0012	0.37±0.0
75.0	0.5984±0.0005	0.5984±0.0005	0.0±0.0
90.0	0.5987±0.0007	0.5987±0.0007	0.0±0.0
95.0	0.5988±0.0006	0.5988±0.0006	0.0±0.0
99.0	0.5989±0.0006	0.5989±0.0006	0.0±0.0

takes place on the lowest class percentage, and at ~50% the gains from class-based rejection are negligible. We do find that on the tieredImagenet dataset, the improvements are greater than those which can be achieved on the CIFAR100 dataset. When making a direct comparison to the specific classes, we find that the hierarchical classes allow for a great improvement over the random class distributions, in the animal and other distributions, the improvements are over 30%. In all experiments, we find the *other* subclass allows for the most successful

TABLE 5.19: Randomly assigned target classes vs hierarhically defined target classes, as in table 5.17, this time using the tiered-ImageNet dataset.

Target Class (%)	Class-Based Improvement (%)		↑ Difference
	Random	Hierarchical	
Animal (40.8)	5.0±0.01	38.94±0.12	33.94±0.11
Object (45.1)	1.97±0.05	14.25±0.07	12.28±0.02
Other (14.1)	14.9±0.04	**54.16±0.39**	**39.26±0.35**

class-based early exit rejection. This is likely due to this class being a lower percentage, this is supported by the random distribution tests; they show that whilst the improvements are minimised in the random case, the lower percentage of the classes the target distribution represents, the greater the performance benefits.

This section has defined class-based rejection, an inference policy which uses hierarchical label information to infer the probability of an input being in the target class distribution. We evaluate this approach, and find that even in the event that these target classes are random, class-based early exiting can provide inference cost savings. When the target classes are defined using the hierarchy, this performance benefit increases dramatically in most cases. To confirm this is not down to class counts, we also analyse random distributions of the same size as the hierarchically defined class distributions and confirm hierarchically defined class distributions perform significantly better than those of the same percentage defined randomly.

We use normalised AUC measurements to evaluate the performance of the new inference technique when compared to conventional early exiting methods. We find that in all cases class-based techniques outperform standard inference techniques. Leading to increased progressiveness, that is, improved accuracy given the same power usage or decreased power usage for a given accuracy requirement.

Now finally, there may exist a use case whereby the target inputs vary at run-time, and there is also exposure to OOD inputs. We describe this use use case as *nested-set* inference. In this scenario it would be beneficial to include both the OOD detection *and* the class-based rejection.

5.3 Nested-Set Inference

To complete our exploration of additional inference mechanisms in branched networks, we consider *nested-set* inference. In this paradigm we consider the scenario in which the model has a target class distribution within its trained classes which can vary at run-time, but also one where the model is exposed to OOD inputs, which should be rejected. This type of scenario would exist where class-based rejection algorithms are being deployed in unknown or foreign areas, where the model is likely to encounter unseen types of inputs, if the model is multi-use, and these are likely to vary at run-time.

This section once again addresses the literature gap of *Introducing New Early Exiting Mechanisms*. In doing so, we make the following contribution:

Introduce *Nested-Set* Early Exiting We introduce a new early exit policy which operates in a mixed-input test environment where some classes within the trained distribution take precedence over others. This is adaptable at inference time, and we report the performance gains that can be achieved when performing nested-set early exiting when compared to conventional methods, and when using only one of OOD-Aware early exiting, and class-based early exiting.

5.3.1 Defining Nested-set Early Exiting

The principle of the inference problem is shown in figure 5.16, there are a number of sets within the input space of the model, each will be nested within one another in the embedding space. In this scenario, the model should be rejecting everything outside of the red circle in all cases, and classifying everything inside of the green circle according to the particular use case. The efficacy of the model in separating the class distributions can be likened to the class-based rejection in the previous section, whereas the overlap of ID and OOD boundaries will be defined by the OOD-Detection performance from section 5.1. That is, our inference policy will require two components used in conjunction with one another, one to separate the ID samples from each other to identify target classes, and another method which identifies the OOD samples. Both of these systems have been discussed in this chapter, in sections 5.2 and 5.1 respectively.

In order to prepare a model for such scenarios, we can combine OOD detection in section 5.1 and class-based rejection in section 5.2. This will give the inference algorithm denoted in algorithm 6. This is simply a combination of algorithms 4 and 5, allowing additional means of early exiting, this will lead to increased accuracy or decreased power usage at given operating points.

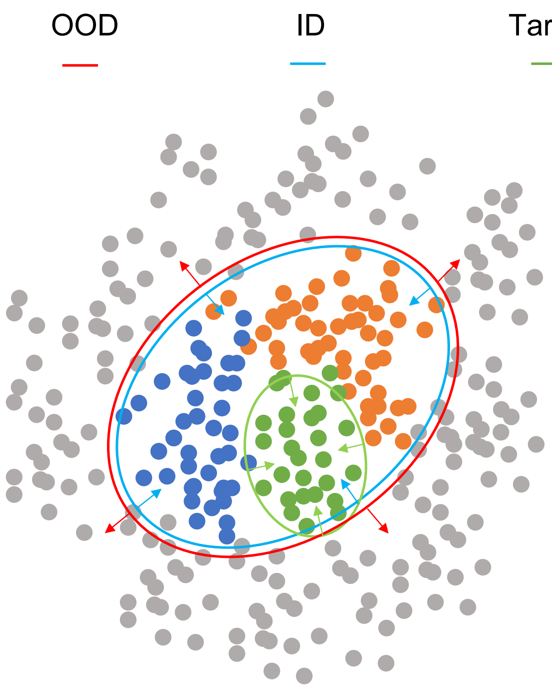


FIGURE 5.16: Nested set inference scenario, where the three splits of input are OOD inputs shown in grey, ID inputs shown in blue green and orange, then the target classes are shown in green. The three areas in the embedding space are contained by the red, blue, and green circles respectively.

Hence, we have a nested-set inference algorithm which functionally is very similar to the other two inference methods proposed in this chapter, and the way in which they are evaluated will be quite similar. We will discuss this in the next section.

5.3.2 Performance Benefits of Nested-Set Inference

Given its similarity to the previous two inference methods, being a combination of them both, it follows the best evaluation method is a combination of the previous ones. That is, to examine our inference methods we once again introduce a mixed input test set as in section 5.1, whilst also concentrating on the target class accuracy, as in the previous section.

For example, below we compare all inference policies on a mixed input test set of the CIFAR100 test set, and OOD data taken from the CIFAR10 test set, we then target the Object class from the ID data. We allow all 4 combinations of inference. The inference operating curves are shown in figure 5.17. We find that the resultant inference mode which is able to operate on and reject OOD inputs whilst also performing class-based early exit rejection, saves significant amounts of power. This allows for an even greater power saving in the mixed-input test set. In this scenario the nested set inference reaches peak accuracy at roughly the same power usage of the least conservative conventional early exiting policy. This is due to the power being saved rejecting OOD inputs, combined with the early exiting from class-based

Algorithm 6 Nested-set early exiting algorithm

```
1: procedure EARLY EXITING INFERENCE WITH OOD DETECTION AND CLASS-BASED REJECTION($f, X, \alpha, \delta, \gamma, \mathtt{k}$)
2:     Load model:  $f$
3:     Read data, Thresholds, Embeddings, and k:   $X, (\alpha, \delta, \gamma), M, \mathtt{k}$

4:     for $x \in X$ do                                    ▷ For all inputs in the test set
5:         for branch $b \in B$ do                         ▷ For all branches in model
6:             $m_b = f_{b-1}(x)$                          ▷ Get embeddings
7:             $\hat{y}_b = \arg\max(f_b(x))$              ▷ Calculate branch prediction
8:             $k_b = \mathtt{get\_knn}(m_b, M; \mathtt{k})$   ▷ Calculate k nearest neighbour distance
9:             $r_b = \mathtt{target\_prob}(f_b(x))$       ▷ Calculate target class probability
10:            $kappa_b = \text{confidence}(f_b(x))$       ▷ Calculate confidence

11:            if $\kappa_b \geq \alpha$ then              ▷ Check classification condition met
12:                return $\hat{y}_b$                      ▷ Return classification if confident
13:            end if

14:            if $r_b \leq \gamma$ then                   ▷ Check class-based rejection condition met
15:                return $\hat{y}_b = -1$                 ▷ Return nil classification
16:            end if

17:            if $k_b \geq \delta$ then                   ▷ Check OOD rejection condition met
18:                return $\hat{y}_b = -1$                 ▷ Return nil classification
19:            end if

20:        end for                                         ▷ Full inference used without early exit
21:        $\kappa_b = \text{confidence}(f_B(x))$          ▷ Calculate final confidence for completeness
22:        $\hat{y}_B = \arg\max(f_B(x))$
23:        return $\hat{y}_B$                              ▷ Return final output
24:    end for                                             ▷ Move to next input

25: end procedure
```

rejection, allowing for a more conservative classification policy in the target class distribution.

Since these distributions do not overlap when analysing them this way, we can instead revisit the quantification technique from section 5.1, and record the peak improvements in both the power and accuracy axes. Since there are many combinations of the, target class, OOD data, and rejection thresholds, we will fix the rejection thresholds at the most successful for each ID dataset, and reduce the number of OOD datasets to those which are most operationally similar. Specifically we fix the rejection thresholds at 0.95 and 0.99 for CIFAR100 and tiered-Imagenet respectively, and drop the SVHN and DTD datasets. Rather than using mini-Imagenet we use tiered-Imagenet in this instance.

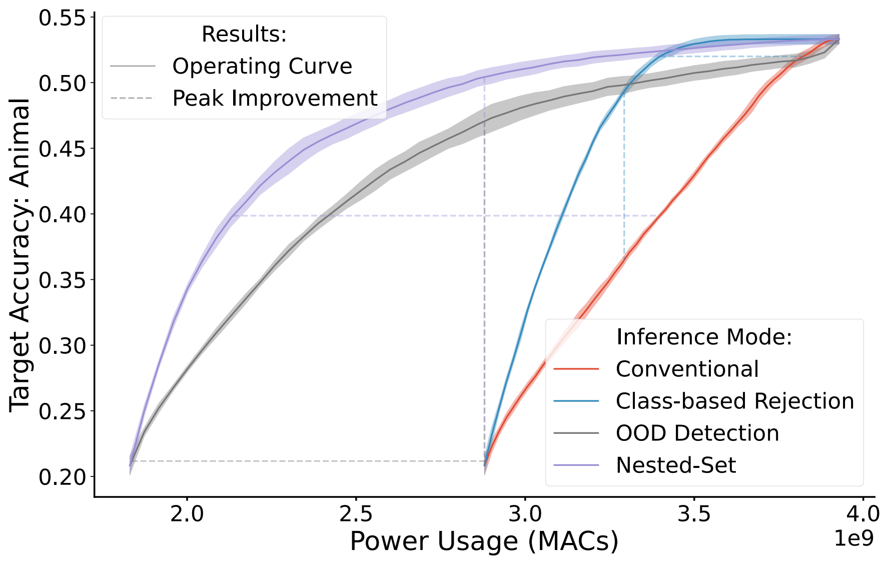


FIGURE 5.17: Comparing the combination of inference methods, example inference comparison using CIFAR100 test set for ID data, and CIFAR10 for OOD data, and the target class is Object. Lines and shaded regions denote the mean and standard devisation respectively over 5 runs. Nested set inference, the combination of OOD detection and class-based rejection, is denoted by the purple line. Peak increases along each dimension are annoted with a dashed line of the same colour while the solid lines denote the operating curves/ranges.

Tables 5.20 and 5.21 give results with CIFAR100 as the ID dataset, using CIFAR10 and tiered-Imagenet as the OOD datasets respectively. Then tables 5.22 and 5.23 give results with tiered-Imagenet as the ID dataset, using CIFAR10 and CIFAR100 as the OOD datasets respectively.

TABLE 5.20: Peak AUC increases for the new early exit reject policies, using CIFAR100 as the ID data and CIFAR10 as in the OOD data.

		Peak Increase (%)		
		Inference Type		
Target Class	Improvement	Class-Based	OOD-Detection	Nested-Set
Animal	Accuracy	1.82±0.56	25.22±0.58	**26.65±0.44**
	Power	4.18±1.52	35.25±0.31	**35.27±0.28**
Object	Accuracy	6.50±0.24	18.13±0.28	**22.45±0.75**
	Power	5.88±0.38	35.14±1.24	**35.32±0.56**
Other	Accuracy	10.28±0.63	12.04±0.5	**14.93±0.72**
	Power	10.78±1.1	35.18±1.57	**36.79±0.22**

For the models using CIFAR100 as the ID dataset, we find that the class-based exiting policy performs poorly for the animal target class. For the other target classes, there are improvements of over 5% in the object class, and over 10% in the 'other' class distribution. The OOD detection policy performs much better, being permitted to classify the OOD data, with power savings of over 30% in all target classes. Accuracy improvements are more varied, greatest on

TABLE 5.21: Peak AUC increases for the new early exit reject policies, using CIFAR100 as the ID data and tiered-ImageNet as in the OOD data.

		Peak Increase (%)		
Target Class	Improvement	Inference Type		
		Class-Based	OOD-Detection	Nested-Set
Animal	Accuracy	1.82±0.56	26.38±0.5	**27.08±0.55**
	Power	4.18±1.52	35.20±0.44	**35.23±0.39**
Object	Accuracy	6.50±0.24	20.17±0.52	**22.9±0.79**
	Power	5.88±0.38	35.06±1.46	**35.34±0.51**
Other	Accuracy	10.28±0.63	12.98±0.65	**14.65±0.91**
	Power	10.78±1.1	35.09±1.95	**35.94±0.53**

the Animal target class, and lowest on the 'other' class. In all experiments we find the Nested-Set inference policy performs the best. These findings are largely repeated when using CIFAR10 as the OOD dataset.

TABLE 5.22: Peak AUC increases for the new early exit reject policies, using tiered-ImageNet as the ID data and CIFAR10 as in the OOD data.

		Peak Increase (%)		
Target Class	Improvement	Inference Type		
		Class-Based	OOD-Detection	Nested-Set
Animal	Accuracy	12.85±0.3	26.22±1.01	**29.58±0.61**
	Power	11.07±0.44	36.34±0.35	**37.15±0.58**
Object	Accuracy	4.01±0.79	21.8±0.36	**23.17±0.52**
	Power	6.39±1.31	**36.35±0.38**	33.66±0.4
Other	Accuracy	16.08±1.32	21.9±1.04	**26.12±1.33**
	Power	15.58±0.75	36.31±1.31	**41.03±0.89**

TABLE 5.23: Peak AUC increases for the new early exit reject policies, using tiered-ImageNet as the ID data and CIFAR100 as in the OOD data.

		Peak Increase (%)		
Target Class	Improvement	Inference Type		
		Class-Based	OOD-Detection	Nested-Set
Animal	Accuracy	12.85±0.3	26.39±0.8	**30.34±0.48**
	Power	11.07±0.44	35.87±0.23	**37.2±0.52**
Object	Accuracy	4.01±0.79	22.23±0.38	**23.61±0.54**
	Power	6.39±1.31	**36.35±0.38**	33.72±0.48
Other	Accuracy	16.08±1.32	22.22±1.3	**26.64±1.26**
	Power	15.58±0.75	35.86±1.13	**40.59±1.1**

For the tiered-Imagenet model we find that all policies generally perform better, or match CIFAR100 based models within their error bounds, with the exception of the class-based policy on the object classes. However, we do find the class-based policy performs better on the animal classes. Once again the nested-set inference policy predominantly records the greatest performance increases. In this set of models we do find that the OOD-Detection policy performs slightly better on the object set of classes, this is due to a sharper rise in the early stage of the OOD-detection's operating range on that particular task, indicating the class-based rejection does not perform well in this small range.

In all cases we see identical performance between OOD datasets in the class-based reject inference policy, this is because the inference policy acts only on ID data, therefore the OOD data makes no difference. Allowing exits on OOD data only on OOD-aware policies reflect the power benefit they can obtain, whilst presevering the pareto-optimal relationship of each when analysing target class accuracy against power usage, as in figure 5.17.

This section has defined nested-set inference, which through combining hierarchical label information and OOD-detection techniques allows for non-target and OOD samples to be rejected early in the inference process. Furthermore, we have shown that significant amounts of power can be saved at inference time when using nested-set inference, when compared to standard early exit classification policies, and indeed inference policies that use only one early-exit reject method.

5.4 Summary

In this chapter we have explored a new paradigm of inference in branched networks, one based on the rejection of inputs. Work before it focussed only on classification, we consider the scenarios in which the inputs being presented are not of use by the system and we open up the opportunity to reject them. In the first section we consider OOD inputs, the second target and non-target class distributions defined by class hierarchy, and in the final section we combine both of these.

The first section considered a key idea in the robustness space for neural networks, distinguishing uncertainty of epistemic origin from that of aleatoric origin. We introduce KNN classifiers to branched neural networks in order to detect OOD and adversarial inputs. That is, inputs they are not equipped to classify. We present extensive experimentation on pre-trained branched neural networks and motivate the detection of such samples. Our proposed approach functions on a number of benchmark OOD tests and on FGSM adversarial attacks. AUROC results are competitive, given the out-of-the-box nature of our method.

A novel early exiting algorithm is detailed, which allows for early exit rejects. To show the entire operating range of the model, we vary the classification and rejection thresholds in unison. This allows us to understand the performance gains our exiting algorithm can make over conventional methods, recording the peak gains.

We show up to ~90% of OOD data can be detected and rejected using our methods. When compared to conventional early exiting methods under the same resource constraints, we show this can lead to a ~15% accuracy improvement, or a ~30% power improvement. We also find that substantial performance gains can be achieved through detecting adversarial inputs. We instead consider the final model accuracy as our detection target and find we can detect up to ~95% of these inputs. Comparing to a conventional exiting method under the same constraints, we find our exiting algorithm records accuracy improvements of ~20% or power saving of ~40%, depending on the strength of the adversarial attack. We find our algorithm is better at detecting the stronger adversarial attacks.

In the second section, we consider class-based early exit rejection. This form of early exit reject considers the scenario in which there are dynamically defined, *target* classes, which are pertinent at classification time, and the others are deemed non-target. To understand this scenario we define these target classes according to class-space hierarchies.

At inference time, we take the summed probability of these classes and if it is below a certain value, the inference can be halted as it is not likely to be a target class. We find that this in almost all cases increases the accuracy of the classifier on the target classes, across the power range the model operates at. This is because the power savings being made on non-target inputs allow for more stringent classification on the target classes.

We study the impact of the hierarchy on this effect by analysing random distributions of the same size, finding that the hierarchically defined target classes lead to much higher improvements than randomly defined distributions. This suggests that the models naturally learn the hierarchies, and intrinsically put similar classes at higher probabilities when classifying them. That is for example, if the model is predicting a dog, it will also put *horse* and *cat* at higher probabilities than *car* or *boat*.

Finally, we look at combining both new inference techniques in the final section, where we describe *nested-set* inference. This is an inference scenario where class-based rejection can be incorporated, but the model is also likely to encounter OOD samples.

We find that this method of inference largely produces the highest increases in peak accuracy and power improvements on a mixed input test set. When compared to OOD detection and class-based rejection alone.

Chapter 6

Conclusions and Future Work

This work has considered Progressive Intelligence as a form of resource aware machine learning. The key distinction between progressive intelligence and other approaches in the field is that it approaches this challenge incrementally, prioritising early gains in performance for low-power costs and improving upon these incrementally. In chapter 3 we consider the dichotomy of data-space and model-space machine learning systems, and how these translate to progressive intelligence systems. In chapter 4 we focus on branched neural networks as an effective model-based approach, exploring them more deeply. Finally, in chapter 5 we explore how additional information can be utilised in branched neural network inference modes, and how this can be used to enable new forms of early exiting.

In this chapter we will first, in section 6.1, go through each of the research chapters and summarise their findings and contributions. Then in chapter 6.2 we will consider the future work for progressive intelligence, both from the perspective of continuing the direction of this work, and from a more general perspective considering new directions.

6.1 Summary

This work has consisted of three main components, chapter 3 which explores progressive intelligence from the perspective of data-space and model-space machine learning systems, chapter 4, which focuses down on branched neural networks, then chapter 5 which focuses down further on new modes of inference in branched neural networks. In this section we will revisit and summarise the work of each chapter, how it fits into the goals of the work, and the contributions made. First briefly mentioning a review in graceful degradation review undertaken at the beginning of the PhD, then we go through chapters 3 to 5 in order of writing.

6.1.1 Graceful Degradation Review

At the beginning of the PhD a review on graceful degradation was written for the Defence Science and Technology Laboratory (Dstl) and the Alan Turing Institute (ATI). Graceful degradation refers to the degradation in performance as a machine learning model encounters data increasingly outside of the distribution on which it was trained. This work investigates ways in which uncertainty can be quantified, methods in maintaining performance on information outside of the distribution, and methods models can employ to actively update themselves when leaving the distribution. Furthermore, the study opened further research opportunities during an industrial placement, where collaborative work was undertaken between Dstl and the ATI investigating hierarchical classification. This work helped inform the work of the final research chapter, chapter 5.

6.1.1.1 Paper and Source Code

This paper is publicly available on arXiv under the title: *Graceful Degradation and Related Fields*, (Dymond, 2021).

6.1.2 Data-Space vs Model Space

In chapter 3 we explored progressive intelligence and how it fits into the dichotomy of data-based systems and model-based systems in the literature. Firstly however, it was necessary to explore a way of quantifying progressiveness, defining a generalisable measure of performance in progressive intelligence systems.

We find that the area under a performance–cost pareto curve is shown to be equivalent to weighting early gains in performance and decreasing this linearly as we move to higher costs. This meets two main criteria, we desire monotonicity in progressive intelligence systems, and we want to prioritise fast gains in performance. The AUC meets both of these, as it encourages high gradient values in the pareto curve at low powers, but also monotonicity at higher powers.

We then shift our attention to data-space progressive intelligence systems, we use 1-dimensional Gaussian distributions to create a simple classification problem. Using these along with the analytical solution, we can record our uncertainty with the numerically determined threshold as we increase the number of samples, N. We find that in all cases the error in our estimation of the threshold, t, should decrease with $N^{-\frac{1}{2}}$. To verify this we implement a prototype data-space progressive intelligence using a KNN classifier. We find on an artificial dataset, that the $\Delta t \propto N^{-\frac{1}{2}}$ line can be considered an upper limit on the error.

Finally in this section, we present a prototype data-based progressive intelligence. Using the mean distance as a measure of confidence we present an early exiting algorithm for use in a progressive intelligence KNN model. We trial this on real-world datasets and find that the distance threshold does not need to be large to achieve good performance in this setting, peak performance is reached at a normalised distance threshold of ~0.8. We also assess the number of samples against KNN performance, finding that mean performance very quickly peaks in some datasets, and the standard deviation decreases as more samples are used, however this finding is dependent on the dataset.

This chapter is concluded with a study of model-based progressive intelligence systems. We present two systems, a prototype ensemble based system, and branched neural network. Trained on CIFAR10, we trial these with a progressive inference method which incrementally increases the power usage of the model. In the ensemble case we gradually add models to the output, in the branched case we work through the model between each intermediate classifier. When using the AUC as measure of performance, we find the branched neural network outperforms the progressive ensemble, as such we choose to take this approach forwards for work in the subsequent chapters.

6.1.2.1 Contributions

Quantifying *Progressiveness* We present a measure of progressiveness in progressive intelligence systems as the AUC of performance–cost distributions. We show that this is equivalent to prioritising early rises in performance for early increases in power, whilst also valuing monotonicity later in the curve.

Upper Bound in Error for Data-Based Progressive Intelligence We show analytically that the error in a data-based classifier should decrease with $N^{-\frac{1}{2}}$, where N represents the number of training samples. We then show this can be considered an upper bound using a KNN classifier on an artificial dataset.

Prototype Data-Based and Model-Based Systems Finally we present prototype progressive intelligence systems, both in the data-based and model-based paradigms. For a data-based progressive intelligence we introduce a progressive KNN algorithm, which incrementally adds inputs until the mean distance is below a certain threshold. We test these on real world tabular datasets. For model-based systems we use CIFAR10 to test our approaches, we introduce a progressive ensemble architecture with a novel early exiting algorithm for such a system. We compare it against a branched neural network and find the branched neural network performs better on the task, using AUC as a measure of performance.

6.1.3 Exploring Branched Networks for Progressive Intelligence

In chapter 4 branched neural networks are explored as a progressive intelligence system, after they are identified as a suitable starting place in chapter 3. We study these systems from three aspects, their training, their architectural design, and their inference policy.

When studying the training of branched neural networks we predominantly use three methods of understanding the effect, class-separation, linear probe accuracy, and centered kernel alignment. Class-separation shows the euclidean separation of classes, whereas the linear probe accuracy shows the *separability* of the classes, centered kernel alignment allows us to compare the similarity of representations between two networks directly.

We find that branch positioning and weighting in branched neural networks has a significant effect on the linear probe accuracy possible on the backbone. The linear probe accuracy improves largely monotonically throughout the backbone, earlier branches increase this more than later branches. Accordingly, stronger weightings give rise to a larger increase in linear probe accuracy. Branch combinations increase this quantity the most, giving rise to the greatest AUC measurements. Class separation is largely low until the later layers of the network.

When analysing the CKA scores we find that the branches shift representations backwards in the network, meaning they act to compress the representational changes the model performs on the data in a forward pass. This has the effect of shifting the transformations important for classification to earlier positions in the network.

Training regimes are also examined, finding that fine-tuning the backbone increases the performance of the network whilst preserving the representational benefits seen from class-separation measurements. We also explore hierarchical losses, finding that certain classes can be weighted earlier in the branches, increasing their relative accuracy earlier in the network, when compared to conventionally trained models.

Following the study of branched network optimisation, we explore the effect of architectural changes in branched networks, now shifting focus to the more pertinent *operating range* of the system. The operating range is generated by changing the confidence requirement of early exiting, simulating inference with this confidence threshold, and collecting power usage and accuracy measurements for each.

We find that decreasing the width has a substantial effect on operating range, shifting the range over which the model operates in. Increasing the width however, has diminishing returns in this setting. The depth of the network extends this operating range at the higher power range, but does little to improve network accuracy. Finally, we examine the number of branches, finding that an increased number of branches increases the range the model can operate at in the lower power ranges, by giving the model earlier exit opportunities at inference time.

We introduce a new inference policy in the final section, mutual agreement. This is an easy to implement, parameter-free, addition to an entropic policy which uses information from previous branches. When two branches agree with one-another, an exit is triggered. We find this can save power in all areas of the operating range, saving ∼ 38% and ∼47% on CIFAR100 inference in Branched ResNet18 and Branched MobileNet architectures respectively. This led us to focus in on branched neural network inference when bringing new ideas to progressive intelligence.

6.1.3.1 Contributions

Branches Increase Class-Separability We find that branches attached to the backbone induce increased class separability, measured using linear probe accuracy, after training. We find this is also heavily influenced by branch positioning and weighting, where combinations of branches, with a majority weighting on the final exit will maximise the AUC of a linear probe accuracy with network depth plot.

Branches Compress Representations CKA analysis reveals that when training a branched model representations that are positioned later in an unbranched model are pushed earlier in the network. This has the effect of compressing the representations that the model forms, we find this effect is dependent on branch positioning.

Fine-tuning Branched Networks increases AUC further Different optimisation strategies are trialed, showing that performance is maximised when the branches are initially trained together, and then the final exit is fine-tuned. This allows the representational benefit of the branches to persist whilst maximising the final exit performance.

Width, Depth, and Branches Change Operating Range We find that the width shifts the operating range of the branched network, the depth extends the upper end of this range, and the number of branches the lower end. Decreasing the width dramatically decreases the power usage of the model, whilst maintaining reasonable levels of accuracy.

Mutual Agreement Exit Policy Mutual Agreement is introduced as a more energy efficient early exiting policy in branched neural networks. It compares the outputs of consecutive exits and performs an exit if the branches agree with one another. Combining this principle with an entropic policy results in a saving of up to ∼47% on the CIFAR100 dataset, depending on the branched neural network backbone.

6.1.3.2 Paper and Source Code

We published the majority of this work at the 33rd British Machine Vision Conference, under the paper title: *Adapting Branched Neural Networks to Achieve Progressive Intelligence*. (Dymond et al., 2022).

The source code for all experiments provides the tools to experiment with the inference processes and scale architectures of branched neural networks. This can be found on GitHub at: `github.com/J-Dymond/adapting-branched-networks`.

6.1.4 Leveraging Additional Information in Progressive Intelligence Systems

Chapter 5 extends the improvements made using mutual agreement and introduces the novel idea of an early exit rejection. The early exit reject requires additional information about the input and the task, namely we use distributional information in the embedding space to detect unclassifiable samples, and label hierarchy information to detect non-target samples. With these new sources of information we introduce three new early exiting policies, distribution-aware early exiting, class-based early exit rejection, and nested-set inference.

Distribution-aware early exiting uses a KNN classifier in the penultimate layer embedding space to identify OOD samples, this is the first time OOD detection has been performed in branched neural networks and it functions out-of-the-box on pre-trained architectures. It can be used to identify OOD samples and adversarial samples, and performs reasonably well attaining an AUROC score of ~0.9 in most cases. When implemented on a mixed-input test set we find that using this inference policy can save up to ~40% of the power usage, or increase accuracy by ~20%, when compared to a conventional early exiting policy which cannot classify the unclassifiable samples.

Class-based early exit rejection introduces the idea of target and non-target classes in a dataset. This is the scenario where at any given time the operator of the progressive intelligence system is only interested in a subset of the trained classes, in our implementation this can vary at run-time. We find, using normalised AUC measurements of the operating curve, this can improve without loss on the conventional early exiting policies, when recording target-class accuracy.

Finally, nested-set inference is the combination of both early exit policies, for the scenario in which there are changing classification priorities in an environment where there is likely to be exposure to OOD samples. Using peak improvement measurements, we find nested-set inference predominantly records the highest improvements on most target-class–OOD combinations.

6.1.4.1 Contributions

Exploring Epistemic Uncertainty in Branched Neural Networks We perform a study of the effect of epistemic uncertainty in branched neural networks, we perform this by introducing OOD and adversarial inputs to the models. We find that entropy distributions are indistinguishable from ID/clean samples, but that the k-neighbour distances in the penultimate layer embedding space produce separable distributions as the input is processed by deeper layers. We collect these two types under the term unclassifiable.

OOD and Adversarial Detection in Branched Neural Networks We introduce a KNN neighbour classifier to the penultimate layer embedding space detect OOD and adversarial samples. This is the first implementation of OOD detection and Adversarial detection in branched networks.

Distribution-Aware Early Exiting Using the KNN classifier we detect unclassifiable inputs at inference time, saving power by rejecting these samples, which in the best case would be processed by the entire model without an early-exit reject system. This can also allow for a more stringent classification policy on in distribution samples, allowing for greater accuracy at a given power budget.

Class-Based Rejection We introduce a novel class-based rejection policy, which is designed to operate in a dynamic environment where target-classes will vary at operating time. This exit policy always improves or matches conventional early exiting, when analysing power usage against the target-class accuracy. We find this policy works much better when using hierarchically defined target classes, however random target classes also give some improvement.

Tiered-ImageNet We introduce a new benchmark for the above classification task using the tieredImageNet dataset.

Nested-Set Inference Finally, a third inference strategy is proposed and implemented in which the model rejects OOD samples, and non-target samples. We refer to this as Nested-Set inference and find that on a mixed input test set it outperforms using only one of a class-based or an inference-based rejection policy, as well as conventional early exiting, when analysing peak improvements of target-class accuracy against power usage distributions.

6.1.4.2 Paper and Source Code

We published the first chapter of this work at the 26th European Conference on Artificial Intelligence, under the paper title: *Exploiting Epistemic Uncertainty at Inference Time for Early-Exit Power Saving*. (Dymond et al., 2023).

We plan to write up the final chapter as a journal paper presenting the implementation and utility of early exit reject mechanisms. This will be structured very similar to the chapter culminating with nested-set inference. The target will be an IEEE venue such as Transactions in Pattern Analysis and Machine Intelligence.

The source code for all experiments provides the tools to experiment with the inference processes and scale architectures of branched neural networks. This can be found on GitHub at: `github.com/J-Dymond/distribution-aware-exiting`.

6.2 Progressive Intelligence Going Forwards

This thesis has largely focused on image classification for its experimentation, specifically on branched neural networks. However, as discussed in chapter 2 there are a plethora of topics and fields that can be applied to progressive intelligence. Hence, in this section we will discuss some ways in which the work of this thesis could be extended, but also tangential directions which have yet to be investigated.

6.2.1 Transformers for Progressive Intelligence

There is only so much that can be achieved with image classification, many of the most recent developments in deep learning are integrating object detection and semantic segmentation into their research. These tasks pose an issue to early exiting CNNs, as their structure impose a rigid inductive bias on the problem. CNNs operate by sequentially downsampling an image through convolutions, whilst this has been shown to be very effective in classification tasks, it also prioritises local information early in the inference process. This is not a problem for conventional neural networks, as the global information is processed later in the network as the receptive field increases. However, for early exiting neural networks this means that the early exit branches are not able to effectively process global information. For image classification this is overcome using pooling layers which aggregate all of the information obtained by the network so far, but for more granular tasks such as object detection and semantic segmentation, this is not an option.

Hence, a new architectural methodology should be adopted to enable early exiting techniques to be applied to these more complex vision problems. Recently a paper by Raghu et al. (2021) has shown that vision transformers, due to their design, incorporate global information much sooner in the inference process. Furthermore, they show that the representations formed are much more similar throughout the model, suggesting that early exiting will be more accurate and more confident when using vision transformer based backbones. They do show that local information remains an important part of the inference process and present training methodologies that encourage the adoption of local representations. Indeed, there have been recent papers which introduce early exiting to transformer architectures, but they focus their development on image classification, missing the opportunity to extend early exiting to more granular tasks (Xu et al., 2023; Bakhtiarnia et al., 2021). So whilst they were not appropriate for the initial investigations in this thesis, they would be an appropriate next step when moving to more complex problems in the vision space.

6.2.2 Optimising Slimmable-Branched Neural Networks

This work investigated *slimmable* neural networks and presented a prototype in section 4.2.3. However, we found that the implementation in this work is difficult to optimise. It is evident from the operating curves, that there is potential to significantly increase the range of a progressive intelligence system using a branched network which is slimmable at run-time.

Existing literature shows that these systems can be optimised without branches to operate at many operating modes. This suggests an issue with our implementation, rather than that of the optimisation problem. Hence, moving away from the sequential training approach may be beneficial for optimising such systems. Yu et al. (2018) use a switchable batch normalisation in the training of slimmable neural networks, this was something lacking in the implementation of this work. However, we also get significantly better results than their *naive* training, where they report 0.1% test accuracy. Yu and Huang (2019b) introduce improvements to the training process, and Li et al. (2021) show that these networks can operate dynamically, albeit without early exiting. Hence, there are definitely opportunities in this area to implement dynamic width and dynamic depth in a progressive intelligence neural network.

6.2.3 Progressive Intelligence for Video Streams

As mentioned in section 6.2.1 the work in the thesis has pertained single data points, specifically the classification of images. However, in the real world this is not necessarily representative of the problems we wish to use machine learning for. For example, object detection (Zou et al., 2023) and image segmentation (Thisanke et al., 2023) are very active areas of the vision space and will often be used in video streams as opposed to individual images.

Progressive intelligence could be incorporated into these problems at a granular, frame-by-frame level, meaning power is not wasted on irrelevant scenes in a video. Branched convolutional neural networks would not be appropriate for such applications due to the locally focused representations early in the network. However, vision transformers might open the door to potential use cases.

Furthermore, work such as that by Figurnov et al. (2017) incorporates *spatially adaptive computation* within an image frame. This could be extended in the progressive intelligence paradigm. This might be through adapting the aggressiveness of the resource saving depending on the context of each frame.

6.2.4 Moving Away From Vision

This work has focused on vision, as vision problems are an accessible test-bed for the ideas of progressive intelligence, and it can be argued that results in vision problems can be translated to other problem domains, such as language problems.

Transformers are a suitable architecture for early exit architectures, and these are the dominant technology in the field of natural language processing, furthermore they have been demonstrated to be amenable to a branched architecture (Xin et al., 2020). Applying the early exiting principles in chapter 5 could improve the benefits of early exiting in this domain.

6.2.5 Progressive Intelligence for Training

Whilst the main focus of the thesis has been on inference cost savings, there is no reason progressive intelligence cannot be implemented at training time also. Training is the most significant cost of such systems, not least of which in vision transformers as in earlier subsection. Hence, there would be significant potential benefits in minimising these costs.

Early stopping was first proposed by Morgan and Bourlard (1989) and is a common method used in neural network training used to reduce resource usage, progressive intelligence could take a more aggressive approach to early stopping. This might be when validation accuracy requirements are met, regardless of what the peak operating performance might be.

Furthermore, active learning is an open area in the field where models are updated in an online manner, but particularly in scenarios where the cost of updating the model is high, making sample selection very important (Cohn et al., 1996; Tharwat and Schenck, 2023). Progressive intelligence could be implemented in such systems where samples are used to update the model in an intelligent manner, but only when absolutely necessary in order to meet the operating requirements of the application. There could also be varying levels of selection costs in such a scenario, however this would be closer to inference-focused progressive intelligence.

6.2.6 Ubiquity in Progressive Intelligence

This work has had a heavy focus on computational resource usage, namely power usage, where we have focused on MACs in an effort to generalise our findings as much as possible. However, the costs of using a machine learning system can be much more general. For example, within the computational resource usage there is latency, memory usage, and disk usage to name a few. Beyond raw computational resource usage, environmental impact can be considered, the economic cost, or even the extent to which privacy is breached when processing sensitive information.

Whilst these might ultimately stem back to power usage, doing as such might obscure components of the system contributing to these costs, and fundamentally a user is more interested in the real-world costs of using a system, than the MAC operations it is using. Hence, approaching progressive intelligence from these cost-points would give rise to a more ubiquitous view of the problem, better integrated with our society and its needs.

For example to combine some of the above ideas, future work could investigate a progressive intelligence form of conversational language models which perform tasks progressively with respect to privacy. This might take the form of asking for personal information only as and when it is needed, otherwise performing tasks to the best of its ability without the information.

6.3 Final Remarks

This work has considered Progressive Intelligence. This is a form of resource-aware machine learning which approaches inference incrementally, making low-cost–low-confidence predictions first, then incrementally improving upon this until confidence requirements are met. In this work we have defined progressive intelligence, and what it means for a model to be progressive with the AUC measurement. We have also discussed how it can fit into the dichotomy of model-based and data-based machine learning systems in the field, producing prototype versions of each. We then shift our focus to branched neural networks as the most complete and example of a progressive intelligence benchmark system. Branched neural networks are explored from each place in their development pipeline, starting at their training finishing with their inference policies. Finally, we look at the inference policies of branched neural networks, and introduce the concept of the early-exit reject, which allows a novel avenue of power savings in such systems.

Progressive intelligence is a rich and interesting area of research, which has the potential to improve the way we use machine learning. The concept of using only what you need is an efficient, judicious, and sustainable way of approaching problems in general, and when applied to machine learning it can significantly decrease the lack of accessibility seen at the state-of-the-art. Hence, by introducing progressive intelligence ideas to modern day machine learning systems we will make them more efficient, cheaper, and ultimately, more accessible for the everyday user.

Appendix A

Mathematical Background

Perceptron The perceptron can be defined as the weighted sum of the input x of dimension, D, and the weightings w of the same dimensionality. The bias term, b, is added which is a vector of length D. It can also be expressed in vector notation as shown on the right hand side in the equation below which corresponds to the linear classification boundary.

$$\text{Perceptron:} \qquad \hat{y} = f\Big(\sum_{i}^{D} w_i x_i + b_i\Big) = \mathbf{w}^{\intercal}\mathbf{x} + \mathbf{b} \tag{A.1}$$

At time-step t, the perceptron algorithm is updated according to the learning rule,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta(\hat{\mathbf{y}} - \mathbf{y})^{\intercal}\mathbf{x}$$

where η is the learning rate and the bias term has been absorbed into the weight vector as w_0. This is using a vectorised form of the target and prediction vectors $\mathbf{y} and \hat{\mathbf{y}}$ respectively, making the right hand term the error rate. Since $\hat{\mathbf{y}}$ takes the values 1 and -1, an update is only made where samples are misclassified. This update is guaranteed to converge provided the data is linearly separable. Furthermore, when the data is completely separable the loss goes to zero, meaning the maximally separating boundary is not guaranteed to be found.

K-Nearest Neighbours The KNN algorithm uses the distance between a test sample x and the training samples x' going back to scalar notation. That is, $\text{dist}(x, x')$, where the distance function is most often the Euclidean distance. The smallest k of these can be interpreted as a subset of the dataset, $S_x \subseteq D$. The class label is then assigned according to the modal value of the subset:

$$\text{k-nearest neighbours:} \qquad \hat{y} = \text{mode}\big(y : (x', y) \in S_x\big) \quad | \quad \text{dist}(x, x') \tag{A.2}$$

Support Vector Machines The complexity is defined by the number of support vectors and the kernel, hence also scale linearly with complexity. Similarly to the case of the perceptron, the decision boundary or hyperplane can be defined as $\mathbf{w}^{\intercal}\mathbf{x} + b = 0$. Away from the decision boundary the sign of this equation will denote the class assignment. The distances, d, from this hyperplane are given by $d_i = |\mathbf{w}^\intercal \mathbf{x}_i + b|$. We wish to find the minimum distances:

$$D = \min_{i \in D} |\mathbf{w}^\intercal \mathbf{x}_i + b|,$$

and then maximise these distances by moving the hyperplane, where i now refers to an input in the dataset. The issue with this constraint is that it doesn't use the label information and hence the hyperplane may not separate the data. To overcome this we can define the *functional margin*:

$$F = \min_{i \in D} y_i |\mathbf{w}^\intercal \mathbf{x}_i + b|$$

Where now, assuming $y \in \{-1, 1\}$, multiplying the distance by the target will give the correct minima. To ensure scale invariance we can normalise w and b using the vector norm, defining the *geometric margin*:

$$M = \min_{i \in D} y_i \left| \frac{\mathbf{w}}{||\mathbf{w}||}^\intercal \mathbf{x}_i + \frac{b}{||\mathbf{w}||} \right|$$

This produces a dual problem which is solved using the lagrangian method, however it is outside of the scope of the thesis but can be found in the book by Vapnik (1999).

Multi-Layer Perceptron In the MLP, the perceptrons are stacked with the signal passing through each sequentially, where the input of one perceptron is the output of the one before it. Hence, we can consider a two-layer MLP with layer widths $D^{(1)}$ and $D^{(2)}$ as

$$\text{MLP (two layer):} \qquad \hat{y}_k = \sigma\Big(\sum_j^{D^{(2)}} w_{k,j}^{(2)} h\Big(\sum_i^{D^{(1)}} w_{j,i}^{(1)} x_i + w_{j,0}^{(1)} \Big) + w_{k,0}^{(2)} \Big), \tag{A.3}$$

where the output vector has k components, corresponding to k classes, h refers to the activation function, and σ the softmax function which acts to normalise the model output and is defined by

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}. \tag{A.4}$$

The output of the function produces an vector of equal length to its input. By absorbing the bias term into the weights vector we can simplify the notation to $\hat{y}_k = \sigma\Big(\sum_j^{D^{(2)}} w_{k,j}^{(2)} h\Big(\sum_i^{D^{(1)}} w_{j,i}^{(1)} x_i\Big)\Big)$. Hence, we can now generalise this to N layers

$$\hat{y}_k = \sigma\Big(\sum_j^{D^{(N)}} w_{k,j}^{(N)} h\Big(\dots \Big(\sum_f^{D^{(n)}} w_{g,f}^{(n)} h\Big(\dots \Big(\sum_b^{D^{(2)}} w_{c,b}^{(2)} h\Big(\sum_a^{D^{(1)}} w_{b,a}^{(1)} x_a \Big)\Big) \dots \Big)\Big) \dots \Big)\Big), \tag{A.5}$$

where the activation function is constant between all N layers, and the model outputs k outputs, which in a classification scenario will correspond to k classes. It can be optimised using a variety of loss functions, the most common are the mean square error (MSE) and cross entropy (CE) for regression and classification tasks respectively. For a given input vector $\mathbf{x}$, output vector $\hat{\mathbf{y}}$, and target vector $\mathbf{y}$, these losses are:

$$\text{MSE:} \quad E = \left|\left|\hat{\mathbf{y}} - \mathbf{y}\right|\right|^2 \tag{A.6}$$

$$\text{CE:} \quad E = -\sum_{c}^{K} y_c \log(\hat{y}_c) \tag{A.7}$$

Where the cross entropy is calculated for a single element in the output vector and will be summed across all outputs. These errors are used to update the model weights $\mathbf{w}$ by gradient descent through backpropagation. Developed by Rumelhart et al. (1985), backpropagation incorporates non-linearities known as an *activation functions*, on each neuron of the network, these produce *activations* which allow meaningful gradients of each neuron to be calculated.

Backpropagation collects these gradients and uses them to determine how each parameter should be changed. To do this the gradients must be found: $\Delta\mathbf{w} = -\eta\frac{\partial E}{\partial \mathbf{w}}$. Since the internal *hidden* layers have no explicit gradient signal with respect to the target, we can use the chain rule with respect to the intermediate output, a

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}, \qquad \text{and we can define:} \quad \frac{\partial E}{\partial a_j} \equiv \delta_j.$$

Where i and j refer to a given layer and the layer succeeding it respectively. The partial derivative of the intermediate output of layer with its weights is simply the activation which we can define as z, meaning we have

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i.$$

For the output layer, k, we can directly use the output and target vectors to compute the local gradient of that layer:

$$\delta_k \equiv \frac{\partial E}{\partial a_k} = h'(a_k)\frac{\delta E}{\delta \hat{y}_k},$$

and the gradient of the hidden layers is given by

$$\delta_j \equiv \frac{\partial E}{\partial a_j} = \sum_k \frac{\delta E}{\delta a_k}\frac{\delta a_k}{\delta a_j} = h'(a_j)\sum_k (w_{k,j})\delta_k.$$

Since the variations in a_j only effect the overall error through the variables in the layer after it a_k. Once all gradients are known we can use this formula to update the parameters of the MLP, more commonly known as a neural network, using gradient descent. The full derivation for backpropagation can be found in chapter 5 of the book by Bishop and Nasrabadi (2006), the other techniques discussed in this section are also explained in greater detail.

Bibliography

Mohammad Abdallah, Alaa Hammad, and Daniel Staegemann. A data collection quality model for big data systems. In *2023 International Conference on Information Technology (ICIT)*, pages 168–172, 2023.

Stefan Aeberhard and M. Forina. Wine. UCI Machine Learning Repository, 1991. DOI: https://doi.org/10.24432/C5PC7J.

Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

Christoph Aoun, Naseem Daher, and Elie Shammas. An energy optimal path-planning scheme for quadcopters in forests. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 8323–8328. IEEE, 2019.

Arian Bakhtiarnia, Qi Zhang, and Alexandros Iosifidis. Multi-exit vision transformer for dynamic inference. *arXiv preprint arXiv:2106.15183*, 2021.

Marília Barandas, Duarte Folgado, Ricardo Santos, Raquel Simão, and Hugo Gamboa. Uncertainty-based rejection in machine learning: Implications for model development and interpretability. *Electronics*, 11(3):396, 2022.

Kevin Baum, Joanna Bryson, Frank Dignum, Virginia Dignum, Marko Grobelnik, Holger Hoos, Morten Irgens, Paul Lukowicz, Catelijne Muller, Francesca Rossi, et al. From fear to action: Ai governance and opportunities for all. *Frontiers in Computer Science*, 5:1210421, 2023.

Dominic Belcher, Adam Prugel-Bennett, and Srinandan Dasmahapatra. Generalisation and the geometry of class separability. In *NeurIPS 2020 Workshop: Deep Learning through Information Geometry*, 2020.

Irwan Bello, William Fedus, Xianzhi Du, Ekin Dogus Cubuk, Aravind Srinivas, Tsung-Yi Lin, Jonathon Shlens, and Barret Zoph. Revisiting resnets: Improved training and scaling strategies. *Advances in Neural Information Processing Systems*, 34, 2021.

Eduard Bertran and Alex Sànchez-Cerdà. On the tradeoff between electrical power consumption and flight performance in fixed-wing uav autopilots. *IEEE Transactions on Vehicular Technology*, 65(11):8832–8840, 2016.

Thomas Bird, Friso Kingma, and David Barber. Reducing the computational cost of deep generative models with binary neural networks. In *International Conference on Learning Representations*, 2021.

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference, 2017.

Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyama, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. High-capacity expert binary networks. In *International Conference on Learning Representations*, 2021.

Marco Buzzelli and Luca Segantin. Revisiting the compcars dataset for hierarchical car classification: New annotations, experiments, and results. *Sensors*, 21(2):596, 2021.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2019.

Shaofeng Cai, Yao Shu, Wei Wang, and Beng Chin Ooi. Dynamic routing networks, 2020.

Shih-Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. Directional pruning of deep neural networks. *Advances in Neural Information Processing Systems*, 33:13986–13998, 2020.

Brian Chen, Andrew Rouditchenko, Kevin Duarte, Hilde Kuehne, Samuel Thomas, Angie Boggust, Rameswar Panda, Brian Kingsbury, Rogerio Feris, David Harwath, James Glass, Michael Picheny, and Shih-Fu Chang. Multimodal clustering networks for self-supervised learning from unlabeled videos. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 8012–8021, October 2021a.

Guangyao Chen, Peixi Peng, Xiangqian Wang, and Yonghong Tian. Adversarial reciprocal points learning for open set recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):8065–8081, 2021b.

Xi Chen, Xiao Wang, Soravit Changpinyo, AJ Piergiovanni, Piotr Padlewski, Daniel Salz, Sebastian Goodman, Adam Grycner, Basil Mustafa, Lucas Beyer, Alexander Kolesnikov, Joan Puigcerver, Nan Ding, Keran Rong, Hassan Akbari, Gaurav Mishra, Linting Xue, Ashish Thapliyal, James Bradbury, Weicheng Kuo, Mojtaba Seyedhosseini, Chao Jia, Burcu Karagol Ayan, Carlos Riquelme, Andreas Steiner, Anelia Angelova, Xiaohua Zhai, Neil Houlsby, and Radu Soricut. Pali: A jointly-scaled multilingual language-image model, 2023.

Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11030–11039, 2020.

M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145, 1996.

Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. Algorithms for learning kernels based on centered alignment. *The Journal of Machine Learning Research*, 13(1):795–828, 2012.

Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

Nello Cristianini, John Shawe-Taylor, André Elisseeff, and Jaz Kandola. On kernel-target alignment. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Xin Dai, Xiangnan Kong, and Tian Guo. Epnet: Learning to exit with flexible multi-branch network. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 235–244, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368599.

Jeffrey Dean. 1.1 the deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 8–14. IEEE, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Benoit Dherin, Michael Munn, Mihaela Rosca, and David Barrett. Why neural networks find simple solutions: The many regularizers of geometric complexity. *Advances in Neural Information Processing Systems*, 35:2333–2349, 2022.

James Diffenderfer, Brian R Bartoldson, Shreya Chaganti, Jize Zhang, and Bhavya Kailkhura. A winning hand: Compressing deep networks can improve out-of-distribution robustness. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

Benoit Donnet, Bruno Baynat, and Timur Friedman. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT conference*, pages 1–12, 2006.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

Fabio Duarte. Amount of data created daily. `https://explodingtopics.com/blog/data-generated-per-day`, 2023. Accessed: December 15, 2023.

Jack Dymond. Graceful degradation and related fields. *arXiv preprint arXiv:2106.11119*, 2021.

Jack Dymond, Sebastian Stein, and Stephen Gunn. Exploiting epistemic uncertainty at inference time for early-exit power saving. In *ECAI 2023: Proceedings of the 26th European Conference on Artificial Intelligence*. IOS Press, August 2023.

Jack Dymond, Sebastian Stein, and Steve R Gunn. Adapting branched networks to realise progressive intelligence. In *33rd British Machine Vision Conference 2022, BMVC 2022, London, UK, November 21-24, 2022*. BMVA Press, 2022.

Stéphane d'Ascoli, Hugo Touvron, Matthew L Leavitt, Ari S Morcos, Giulio Biroli, and Levent Sagun. Convit: Improving vision transformers with soft convolutional inductive biases. In *International Conference on Machine Learning*, pages 2286–2296. PMLR, 2021.

Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1039–1048, 2017.

R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2018.

Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

Yonatan Geifman and Ran El-Yaniv. Selectivenet: A deep neural network with an integrated reject option. In *International conference on machine learning*, pages 2151–2159. PMLR, 2019.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. 2007.

Cong Guo, Yuxian Qiu, Jingwen Leng, Xiaotian Gao, Chen Zhang, Yunxin Liu, Fan Yang, Yuhao Zhu, and Minyi Guo. SQuant: On-the-fly data-free quantization via diagonal hessian approximation. In *International Conference on Learning Representations*, 2022.

Aminu Hamza, Ameer Mohammed, and Abubakar Isah. Towards solar-powered unmanned aerial vehicles for improved flight performance. In *2019 2nd International Conference of the IEEE Nigeria Computer Chapter (NigeriaComputConf)*, pages 1–5, 2019.

Sicong Han, Chenhao Lin, Chao Shen, Qian Wang, and Xiaohong Guan. Interpreting adversarial examples in deep learning: A review. *ACM Comput. Surv.*, 55(14s), jul 2023. ISSN 0360-0300.

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

Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015b.

Soufiane Hayou, Jean-Francois Ton, Arnaud Doucet, and Yee Whye Teh. Robust pruning at initialization. In *International Conference on Learning Representations*, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.

Kilian Hendrickx, Lorenzo Perini, Dries Van der Plas, Wannes Meert, and Jesse Davis. Machine learning with a reject option: A survey. *arXiv preprint arXiv:2107.11277*, 2021.

Byeongho Heo, Jeesoo Kim, Sangdoo Yun, Hyojin Park, Nojun Kwak, and Jin Young Choi. A comprehensive overhaul of feature distillation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Saihui Hou, Yushan Feng, and Zilei Wang. Vegfru: A domain-specific dataset for fine-grained visual categorization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 541–549, 2017.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

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

Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3812–3821, 2019.

Ting-Kuei Hu, Tianlong Chen, Haotao Wang, and Zhangyang Wang. Triple wins: Boosting accuracy, robustness and efficiency together by enabling input-adaptive inference. *arXiv preprint arXiv:2002.10025*, 2020.

Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q Weinberger. Multi-scale dense networks for resource efficient image classification. *arXiv preprint arXiv:1703.09844*, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

Fatih Ilhan, Ling Liu, Ka-Ho Chow, Wenqi Wei, Yanzhao Wu, Myungjin Lee, Ramana Kompella, Hugo Latapie, and Gaowen Liu. Eenet: Learning to early exit for adaptive inference, 2023.

Joy Andrew Immanuel Damanik, Imanuel Maurice Dermawan Sitanggang, Fajar Sam Hutabarat, Guntur Petrus Boy Knight, and Albert Sagala. Quadcopter unmanned aerial vehicle (uav) design for search and rescue (sar). In *2022 IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM)*, pages 01–06, 2022.

Md Saroar Jahan and Mourad Oussalah. A systematic review of hate speech automatic detection using natural language processing. *Neurocomputing*, page 126232, 2023.

Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc Le, Yun-Hsuan Sung, Zhen Li, and Tom Duerig. Scaling up visual and vision-language representation learning with noisy text supervision. In *International Conference on Machine Learning*, pages 4904–4916. PMLR, 2021.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

Ziv Katzir and Yuval Elovici. Detecting adversarial perturbations through spatial behavior in activation spaces. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9, 2019.

Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*, pages 3301–3310. PMLR, 2019.

Patrick Kidger and Terry Lyons. Universal approximation with deep narrow networks. In *Conference on learning theory*, pages 2306–2327. PMLR, 2020.

Hyeji Kim, Muhammad Umar Karim Khan, and Chong-Min Kyung. Efficient neural network compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12569–12577, 2019.

Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

Ron Kohavi and David H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In *ICML*, pages 275–283, 1996.

Simon Kornblith, Honglak Lee, Ting Chen, and Mohammad Norouzi. What's in a loss function for image classification? *CoRR*, abs/2010.16402, 2020.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International Conference on Machine Learning*, pages 3519–3529. PMLR, 2019.

Alex Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world, 2017.

Salla-Maaria Laaksonen, Jesse Haapoja, Teemu Kinnunen, Matti Nelimarkka, and Reeta Pöyhtäri. The datafication of hate: Expectations and challenges in automated hate speech monitoring. *Frontiers in big Data*, 3:3, 2020.

Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*, pages 1–15, 2020.

Ya Le and Xuan Yang. Tiny imagenet visual recognition challenge. *CS 231N*, 7(7):3, 2015.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Jaeho Lee, Sejun Park, Sangwoo Mo, Sungsoo Ahn, and Jinwoo Shin. Layer-adaptive sparsity for the magnitude-based pruning. In *International Conference on Learning Representations*, 2021.

Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. *Advances in neural information processing systems*, 31, 2018.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

Cyrus Levinthal. How to fold graciously. *Mossbauer spectroscopy in biological systems*, 67: 22–24, 1969.

Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. Dynamic slimmable network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8607–8617, June 2021.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets, 2016.

Zhenyu Liao, Romain Couillet, and Michael W. Mahoney. Sparse quantized spectral clustering. In *International Conference on Learning Representations*, 2021.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search, 2018.

Xingchao Liu, Mao Ye, Dengyong Zhou, and Qiang Liu. Post-training quantization with multiple points: Mixed precision without mixed precision. *arXiv preprint arXiv:2002.09049*, 2020.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021a.

Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021b.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11976–11986, June 2022.

Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. In *International Conference on Learning Representations*.

Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013.

Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

Fanxu Meng, Hao Cheng, Ke Li, Huixiang Luo, Xiaowei Guo, Guangming Lu, and Xing Sun. Pruning filter in filter. *Advances in Neural Information Processing Systems*, 33:17629–17640, 2020a.

Xiangming Meng, Roman Bachmann, and Mohammad Emtiyaz Khan. Training binary neural networks using the bayesian learning rule. In *International conference on machine learning*, pages 6852–6861. PMLR, 2020b.

Lassi Meronen, Martin Trapp, Andrea Pilzer, Le Yang, and Arno Solin. Fixing overconfidence in dynamic neural networks. *arXiv preprint arXiv:2302.06359*, 2023.

Marvin Minsky and Seymour Papert. Perceptrons. 1969.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1765–1773, 2017.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

Nelson Morgan and Hervé Bourlard. Generalization and parameter estimation in feedforward nets: Some experiments. *Advances in neural information processing systems*, 2, 1989.

Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan-Kaufmann, 1989.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, , and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *VLDB DISPA Workshop 2020*, 2020.

Mariusz Naumowicz, Paweł Pietrzak, Szymon Szczęsny, and Damian Huderek. Cmos perceptron for vesicle fusion classification. *Electronics*, 11(6), 2022. ISSN 2079-9292.

Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. A modern take on the bias-variance tradeoff in neural networks, 2019.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

Thao Nguyen, Maithra Raghu, and Simon Kornblith. Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth. In *International Conference on Learning Representations*, 2021.

OpenAI. Gpt-4 technical report, 2023.

Sejun Park, Jaeho Lee, Sangwoo Mo, and Jinwoo Shin. Lookahead: A far-sighted alternative of magnitude-based pruning. In *International Conference on Learning Representations*, 2020.

Wonpyo Park, Dongju Kim, Yan Lu, and Minsu Cho. Relational knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Florent Perronnin, Jorge Sánchez, and Thomas Mensink. Improving the fisher kernel for large-scale image classification. In *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV 11*, pages 143–156. Springer, 2010.

Hieu Pham, Zihang Dai, Qizhe Xie, and Quoc V Le. Meta pseudo labels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11557–11568, 2021.

Francesco Pinto, Harry Yang, Ser-Nam Lim, Philip Torr, and Puneet K. Dokania. Using mixup as a regularizer can surprisingly improve accuracy & out-of-distribution robustness. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

Haotong Qin, Zhongang Cai, Mingyuan Zhang, Yifu Ding, Haiyu Zhao, Shuai Yi, Xianglong Liu, and Hao Su. Bipointnet: Binary neural network for point clouds. In *International Conference on Learning Representations*, 2021a.

Haotong Qin, Yifu Ding, Mingyuan Zhang, YAN Qinghua, Aishan Liu, Qingqing Dang, Ziwei Liu, and Xianglong Liu. Bibert: Accurate fully binarized bert. In *International Conference on Learning Representations*, 2021b.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6076–6085. Curran Associates, Inc., 2017.

Maithra Raghu, Thomas Unterthiner, Simon Kornblith, Chiyuan Zhang, and Alexey Dosovitskiy. Do vision transformers see like convolutional neural networks? *Advances in Neural Information Processing Systems*, 34, 2021.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8821–8831. PMLR, 18–24 Jul 2021.

Brandon Reagan, Udit Gupta, Bob Adolf, Michael Mitzenmacher, Alexander Rush, Gu-Yeon Wei, and David Brooks. Weightless: Lossy weight encoding for deep neural network compression. In *International Conference on Machine Learning*, pages 4324–4333. PMLR, 2018.

Mengye Ren, Eleni Triantafillou, Sachin Ravi, Jake Snell, Kevin Swersky, Joshua B Tenenbaum, Hugo Larochelle, and Richard S Zemel. Meta-learning for semi-supervised few-shot classification. *arXiv preprint arXiv:1803.00676*, 2018.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Amir Rosenfeld and John K. Tsotsos. Intriguing properties of randomly weighted networks: Generalizing while learning next to nothing. In *2019 16th Conference on Computer and Robot Vision (CRV)*, pages 9–16, 2019.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Tilman Räuker, Anson Ho, Stephen Casper, and Dylan Hadfield-Menell. Toward transparent ai: A survey on interpreting the inner structures of deep neural networks. In *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 464–483, 2023.

Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning, 2020.

Achim Schilling, Claus Metzner, Jonas Rietsch, Richard Gerum, Holger Schulze, and Patrick Krauss. How deep is deep enough?–quantifying class separability in the hidden layers of deep neural networks. *arXiv preprint arXiv:1811.01753*, 2018.

Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

Murat Sensoy, Lance Kaplan, and Melih Kandemir. Evidential deep learning to quantify classification uncertainty. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3179–3189. Curran Associates, Inc., 2018.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Bernard W Silverman and M Christopher Jones. E. fix and jl hodges (1951): An important contribution to nonparametric discriminant analysis and density estimation: Commentary on fix and hodges (1951). *International Statistical Review/Revue Internationale de Statistique*, pages 233–238, 1989.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Le Song, Alex Smola, Arthur Gretton, Justin Bedo, and Karsten Borgwardt. Feature selection via dependence maximization. *Journal of Machine Learning Research*, 13(5), 2012.

J Springenberg, Alexey Dosovitskiy, Thomas Brox, and M Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (workshop track)*, 2015.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

Samuel Stanton, Pavel Izmailov, Polina Kirichenko, Alexander A Alemi, and Andrew G Wilson. Does knowledge distillation really work? In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 6906–6919. Curran Associates, Inc., 2021.

Tianxiang Sun, Yunhua Zhou, Xiangyang Liu, Xinyu Zhang, Hao Jiang, Zhao Cao, Xuanjing Huang, and Xipeng Qiu. Early exiting with ensemble internal classifiers. *arXiv preprint arXiv:2105.13792*, 2021.

Yiyou Sun, Yifei Ming, Xiaojin Zhu, and Yixuan Li. Out-of-distribution detection with deep nearest neighbors. In *International Conference on Machine Learning*, pages 20827–20840. PMLR, 2022.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv e-prints*, page arXiv:1512.00567, December 2015.

Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*, pages 10096–10106. PMLR, 2021.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019.

Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020.

Shengkun Tang, Yaqing Wang, Zhenglun Kong, Tianchi Zhang, Yao Li, Caiwen Ding, Yanzhi Wang, Yi Liang, and Dongkuan Xu. You need multiple exiting: Dynamic early exiting for accelerating unified vision language model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10781–10791, 2023.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks, 2017.

Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

Alaa Tharwat and Wolfram Schenck. A survey on active learning: State-of-the-art, practical challenges and research directions. *Mathematics*, 11(4):820, 2023.

Hans Thisanke, Chamli Deshan, Kavindu Chamith, Sachith Seneviratne, Rajith Vidanaarachchi, and Damayanthi Herath. Semantic segmentation using vision transformers: A survey. *Engineering Applications of Artificial Intelligence*, 126:106669, 2023.

Yonglong Tian, Dilip Krishnan, and Phillip Isola. Contrastive representation distillation. *arXiv preprint arXiv:1910.10699*, 2019.

Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

Grant Van Horn, Oisin Mac Aodha, Yang Song, Yin Cui, Chen Sun, Alex Shepard, Hartwig Adam, Pietro Perona, and Serge Belongie. The inaturalist species classification and detection dataset. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8769–8778, 2018.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 1999.

Vladimir Naumovich Vapnik. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780, 1963.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. Scalable-effort classifiers for energy-efficient machine learning. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335201.

Tomas Vojir, Jan Sochman, Rahaf Aljundi, and Jiri Matas. Calibrated out-of-distribution detection with a generic representation, 2023.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

Peng Wang, Shijie Wang, Junyang Lin, Shuai Bai, Xiaohuan Zhou, Jingren Zhou, Xinggang Wang, and Chang Zhou. One-peace: Exploring one general representation model toward unlimited modalities, 2023.

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks, 2017.

Y. Wang, J. Shen, T. K. Hu, P. Xu, T. Nguyen, R. Baraniuk, Z. Wang, and Y. Lin. Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):623–633, 2020.

Yulin Wang, Rui Huang, Shiji Song, Zeyi Huang, and Gao Huang. Not all images are worth 16x16 words: Dynamic transformers for efficient image recognition. *Advances in Neural Information Processing Systems*, 34, 2021.

Yutong Wang and Clayton Scott. VC dimension of partially quantized neural networks in the overparametrized regime. In *International Conference on Learning Representations*, 2022.

Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016.

Xiu-Shen Wei, Quan Cui, Lei Yang, Peng Wang, and Lingqiao Liu. Rpc: A large-scale retail product checkout dataset. *arXiv preprint arXiv:1901.07249*, 2019.

Xiuying Wei, Ruihao Gong, Yuhang Li, Xianglong Liu, and Fengwei Yu. QDrop: Randomly dropping quantization for extremely low-bit post-training quantization. In *International Conference on Learning Representations*, 2022.

Mangasarian Olvi Street Nick Wolberg, William and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.

Maciej Wolczyk, Bartosz Wójcik, Klaudia Bałazy, Igor T. Podolak, Jacek Tabor, Marek Śmieja, and Tomasz Trzcinski. Zero time waste: Recycling predictions in early exit neural networks. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V. Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference, 2020.

Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

Guanyu Xu, Jiawei Hao, Li Shen, Han Hu, Yong Luo, Hui Lin, and Jialie Shen. Lgvit: Dynamic early exiting for accelerating vision transformer. In *Proceedings of the 31st ACM International Conference on Multimedia*, pages 9103–9114, 2023.

Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. Recu: Reviving the dead weights in binary neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5198–5208, 2021.

Kohei Yamamoto. Learnable companding quantization for accurate low-bit neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5029–5038, 2021.

Jingkang Yang, Haoqi Wang, Litong Feng, Xiaopeng Yan, Huabin Zheng, Wayne Zhang, and Ziwei Liu. Semantically coherent out-of-distribution detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8301–8309, 2021a.

Jingkang Yang, Pengyun Wang, Dejian Zou, Zitang Zhou, Kunyuan Ding, WENXUAN PENG, Haoqi Wang, Guangyao Chen, Bo Li, Yiyou Sun, et al. Openood: Benchmarking generalized out-of-distribution detection. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael Jordan. Ml-loo: Detecting adversarial examples with feature attribution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6639–6647, 2020.

Taojiannan Yang, Sijie Zhu, Matias Mendieta, Pu Wang, Ravikumar Balakrishnan, Minwoo Lee, Tao Han, Mubarak Shah, and Chen Chen. Mutualnet: Adaptive convnet via mutual learning from different model configurations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021b.

Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. In *International Conference on Machine Learning*, pages 10820–10830. PMLR, 2020a.

Xucheng Ye, Pengcheng Dai, Junyu Luo, Xin Guo, Yingjie Qi, Jianlei Yang, and Yiran Chen. Accelerating cnn training by pruning activation gradients. In *European Conference on Computer Vision*, pages 322–338. Springer, 2020b.

Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1803–1811, 2019a.

Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019b.

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks, 2018.

Shigeng Zhang, Shuxin Chen, Xuan Liu, Chengyao Hua, Weiping Wang, Kai Chen, Jian Zhang, and Jianxin Wang. Detecting adversarial samples for deep learning models: A comparative study. *IEEE Transactions on Network Science and Engineering*, 9(1):231–244, 2022.

Shunshi Zhang and Bradly C. Stadie. One-shot pruning of recurrent neural networks by jacobian spectrum evaluation. In *International Conference on Learning Representations*, 2020.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

Zhuofan Zong, Guanglu Song, and Yu Liu. Detrs with collaborative hybrid assignments training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 6748–6758, October 2023.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016.

Zhengxia Zou, Keyan Chen, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *Proceedings of the IEEE*, 111(3):257–276, 2023.