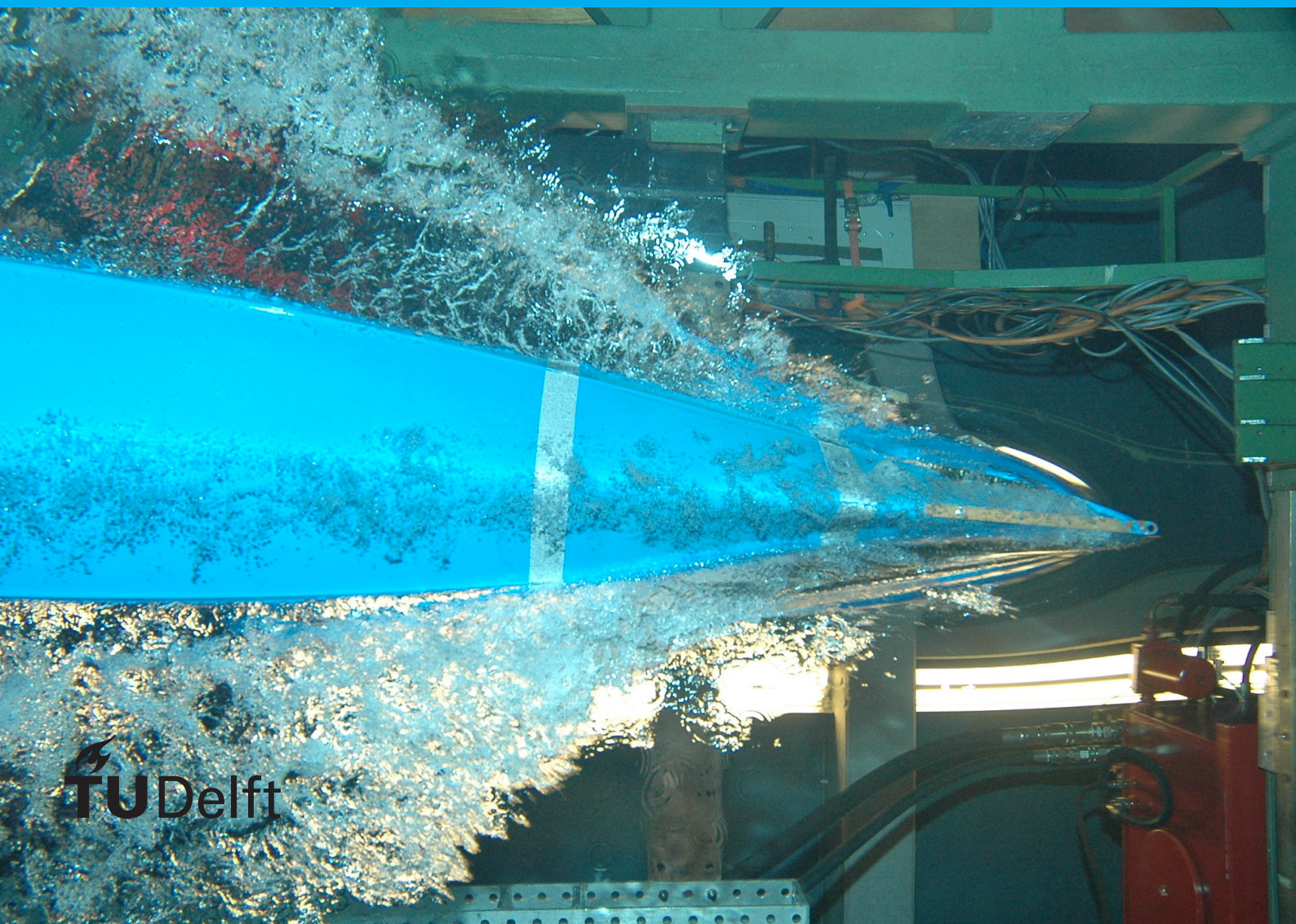
Explaining overthinking in Multi-Scale Dense networks

Why more computation does not always lead to better results

Damian Voorhout



Explaining overthinking in Multi-Scale Dense networks

Why more computation does not always lead to better results

by

Damian Voorhout

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday February 16, 2022 at 13:00 PM.

Student number:	4388550	
Project duration:	November 26, 2020 – February 16, 2022	
Thesis committee:	Prof. dr. J. van Gemert,	Delft University of Technology, supervisor
	PhD candidate X. Liu,	Delft University of Technology, daily supervisor
	Prof. dr. P. Bosman,	Delft University of Technology, committee member

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

Originally, the goal of this thesis was to improve upon the efficiency of inference on temporal data. Inference on image data tends to get slow, inference on video data is slower still. We started working with Multi-Scale Dense networks early on to manifest some type of speedup on video inference and it was during that time we encountered the counter-intuitive phenomenon now known as overthinking. I must have done the experiments at least 5 times before I became convinced that my results were correct. "How can a network get worse results when spending more computation". It made little sense at first. However, once convinced that it was not just some consequence of my sloppiness, I actually became rather invested in it and we changed course, trying to uncover the machinations underpinning this strange occurrence.

The thesis ended up taking many twists and turns, some good, some less so. Nevertheless, I would like to thank Jan for encouraging any and all scientific inquiries and tangents that we ended up going on. Embodying a true researcher, Jan always motivates you to just try it; in the worst case you'll encounter some other line of reasoning to pursue. I would also like to thank Xin, my daily supervisor, at times the voice of reason when my ideas were unreasonable. Her expertise did not end up aligning with what we ended up researching, I'm therefore all the more grateful she helped me whenever I asked her to. Finally, I would like to thank Peter, for taking time out of his schedule to be a part of the Thesis Committee.

Damian Voorhout
Rotterdam, February 2022

Abstract

Traditional convolutional neural networks exhibit an inherent limitation, they can not adapt their computation to the input while some inputs require less computation to arrive at an accurate prediction than others. Early-exiting setups exploit this fact by only spending as much computation as is necessary and subsequently exiting the sample early. In an end-to-end trained convolutional neural network with multiple classifiers, one might expect deeper classifiers to perform better in every circumstance than shallow classifiers; deeper layers make use of the computation done by earlier layers after all. However, this is not always the case and more computation can lead to worse results. This phenomenon, which has been dubbed overthinking, has been documented in several traditional convolutional neural networks with intermediate classifiers. It has been conjectured that it happens due to later classifiers making use of more complex feature which benefit from a larger receptive field. These later classifiers then claim to discern said features in regions of the image which do not contain them, effectively making the classifiers misclassify images that can be classified correctly by shallow classifiers. However, we have observed overthinking in Multi-Scale Dense networks, an end-to-end hand-tuned network optimized for early-exiting for which the given argument in relation to the receptive field does not hold due to its unique architecture. For this reason, in this thesis we attempt to explain overthinking in Multi-Scale Dense networks. We show that in general there seems to be no connection between what a classifier in a Multi-Scale Dense network learns and the data itself. This in turn suggests that overthinking does not take place due to specialization of the classifiers. Instead, we offer up an alternative theory for overthinking in the form of stochasticity inherent to the training process.

List of Figures

List of Tables

Contents

1

Introduction

Deep neural networks have started to play an important role in several areas including natural language processing [55, 10, 5] and computer vision [17, 28, 19, 52]. While mathematical constructs that underpin deep learning have been known for decades, recent advancements in computing power have served as a catalyst for the growth in this field of computer science. As computing power increases, more and more deep and powerful models emerge including VGG [50], AlexNet [28], ResNet [17], DenseNet [19] and GoogleNet [52]. These models serve as the de facto benchmark for future deep learning innovation. Performing inference on some of the deepest models can take up to 10 milliseconds for ResNet-152 [4]. If we consider a setting where a large collection of samples need to be classified as is the case for many indexing websites such as Google and Yahoo, saving even one-thousandth of a second per classified sample can reduce the total computation time by almost 3 hours in the case of 10 million images. Note that Google has likely indexed well over a dozen trillion images at this point if we consider the individual frames uploaded to YouTube. Lowering the computational cost of inference translates directly into a reduction in power consumption. A sizable body of work has already been dedicated to the optimization of deep neural networks, ranging from knowledge distillation [40, 7] and parameter reduction [62, 37, 38] to weight quantization [58, 13]. These type of methods are often model-agnostic and can even improve performance. However, whether or not these methods are implemented, traditional deep neural networks still have static computational graphs and fixed parameters. As a consequence, a static neural network will always spend the same amount of computation if the input dimensions remain fixed. If the input sample is hard to classify, we are perfectly happy to apply the full power of the network to maximize the likelihood of it returning an accurate prediction. If on the other hand the input is easy, throwing the full weight of a static deep neural network at the problem would be overkill. Spending unnecessary computation increases both the network's inference time and power consumption. Ideally, we would have the network only spend as much computation as it needs to. This is the essence of early-exiting; the input is processed iteratively until either the computational budget has been spent or the network has become confident enough in its prediction. Not only do these type of dynamic networks enjoy an overall reduction of computation, they are also compatible with existing efficiency optimization techniques and more widely applicable in real world settings due to their flexibility.

There are two settings where dynamic neural networks have a distinct advantage over their static counterparts: During anytime prediction where a network is asked to output a result at a moments notice, and budgeted batch classification where a group of samples need to be processed within a singular fixed computational budget. In the latter case, a dynamic network will be able to maximize its performance by spending more of the budget on harder samples. Search engines indexing images is an example of a budgeted batch classification scenario. For a real world example of anytime prediction, think of self driving cars [3] where road signs need to be classified to potentially alter the vehicle's driving behaviour. Road signs can appear around corners, the car can be driving at various speeds and there might be stochasticity in the speed with which signs are recognized in the first place. All these variables lead to variability in time constraints imposed on the classification network of the vehicle. Static networks will not be able to adapt to the changing circumstances, whereas dynamic networks can.

Early-exiting can be implemented in several ways, where arguably the most straightforward way is

by using a cascade of networks. In this method, several pre-trained models are figuratively stacked on top of each other and each consecutive model is larger and more powerful than the previous one. If the output from the first, most efficient model is deemed sufficient, the prediction is treated as the final output of the network, otherwise the model next in line is invoked until the final model is reached. While the simplicity of a cascade setup is certainly attractive, it suffers from computational redundancy as each model has to start inference from scratch. For this reason, more nuanced early-exiting approaches have emerged recently in the form of hand-tuned end-to-end networks optimized for early-exiting. These type of networks contain multiple classifiers and each classifier makes use of computation that has already been performed by the classifier that preceded it. One of the most innovative and high performing architectures in this paradigm is the Multi-Scale Dense Network (MSDNet) [20].

Classifiers in an MSDNet share the majority of their parameters as each consecutive classifier makes use of computation performed by previous layers. This is part of the reason why end-to-end trained dynamic networks are preferred over model cascades. One might reasonably expect then that each consecutive classifier performs better than the one before it and on average over a given test set this is almost always the case; the average performance of individual classifiers in an MSDNet increases monotonically. However, this is not necessarily the case for individual samples. It can happen that a classifier later in the network misclassifies a sample, while an earlier classifier already managed to come to the correct conclusion; more computation can lead to worse results. We refer to this phenomenon as 'overthinking'. In general, the set of samples a classifier at depth $i + k$ gets correct is not a superset of the set of samples that classifier i gets correct even though the average performance of classifier $i + k$ surpasses that of classifier i. This suggests a certain level of independence in each classifier despite the overwhelming level of built-in dependencies inherent in MSDNet's design philosophy. Early-exiting enables an MSDNet to exploit this independence to an extent as the network will end up selecting the most appropriate classifier for each input based on the confidence a classifier has in its prediction. A higher confidence is correlated with higher accuracy and samples thus tend to exit at classifiers that are likely to classify the sample correctly. As a consequence, early-exiting directly combats overthinking. It is then maybe no surprise that the collective performance of all classifiers in an MSDNet surpasses that of any individual classifier in the network. Understanding overthinking is closely tied to understanding why some classifiers correctly classify particular samples and why other classifiers can not. This knowledge can subsequently be used to improve upon existing early-exiting strategies, improving adaptive inference performance and reducing computation of dynamic neural networks.

Any code that was used in relation to Multi-Scale Dense networks in this thesis is often based on work done by the original authors. Their code can be found on their Github page[1].

1.1. Research objectives

In this thesis we explore the concept of overthinking in the context of Multi-Scale Dense networks. The goal is to gain insight in the phenomenon itself to potentially improve on early-exiting in dynamic neural networks. To this end, we derive several research questions to structure our research around.

1. Can early-exiting in MSDNet be learned directly by means of a policy network?
2. Do classifiers in MSDNet end up specializing in specific subsets of the original dataset?
3. Are there any image statistics that can be used to discriminate between the subsets that classifiers in an MSDNet end up learning?
4. What causes overthinking in MSDNets?

1.2. Contributions

The main contribution of this thesis is showing the potential absence of a connection between the input data and what a classifier in an MSDNet ends up learning. This suggest that classifiers do not end up specializing in specific subsets of the original datasets. It allows us to suggest a different potential cause, namely that overthinking in MSDNets is caused by stochasticity inherent to its training process. To guide the reader to this conclusion throughout this thesis we provide them with background information and auxiliary findings. The literary contributions are as follows:

[1] https://github.com/kalviny/MSDNet-PyTorch

1. We provide the reader with a rough overview of the workings of deep neural networks, specifically in the context of visual data.
2. We provide the reader with an overview of how adaptive inference can be performed in convolutional neural networks. We go particularly in depth into the workings of early-exiting.

The main experimental findings can be summarized as follows:

1. We show that policy networks can be used to learn early-exit strategies in MSDNets if there is a substantial presence of correlation between the data and what classifiers end up learning.
2. We show that in general, policy networks are unable to detect a correlation between the data and what classifiers in an MSDNet end up learning.

Finally, there are auxiliary findings that are worth mentioning:

1. We show how overthinking can be reduced based on work from the original authors of the Multi-Scale Dense network [34].
2. We show how the frequency domain is not a reliable indicator of sample difficulty and discrete cosine transform analysis does not suffice as a discriminatory feature to separate the learned subsets of classifiers in MSDNets.

1.3. Outline

We start the thesis off with a chapter explaining the basics of deep learning in chapter 2. The second halve dives into convolutional neural networks which specialize in visual data. At the end we stress the importance of the concept of the receptive field, which plays a fundamental role in this thesis. Next, chapter 3 goes into the fundamentals of adaptive inference, the resulting computational settings and early-exiting. Here, we also cover the workings of MSDNets in detail. The chapter ends with an explanation of overthinking and a way to reduce it. Chapter 4 covers the main findings, in it we explore the idea of classifiers in MSDNets specializing as a reason for overthinking. We describe the experimental setup and show that the results suggest a lack of specialization. We conclude the chapter with an alternative explanation of what might be causing overthinking in MSDNets. Finally, chapter 5 covers concluding remarks, limitations of the experiments and its results and possible future work.

2

Deep learning

Traditional programs are able to solve a great deal of complicated tasks and problems. Programmers specify exactly what the program should do using both high and- low level specifications. This works well for problems that have solutions that can either be expressed in rigorous mathematics or are clearly definable by humans, such as figuring out what the closest bakery is to your location, or creating a traffic light controller that minimizes the average wait time of motorists. Given a list of bakeries and their distance to the relevant location we can easily determine what a solution should look like. Showing that a traffic light controller is optimal is more involved, but we can all agree on whether an arbitrary traffic light controller is valid or not given the appropriate constraints. The same can not be said for other problems however. Take for example the classification of images where we want to know what object or organism is depicted in the image. This is a trivial task for humans and it is certainly easier than creating an optimal traffic light controller, we simply use our intuition and experience. Yet we have a hard time coming up with a robust set of rules that govern our intuition: we do not know exactly how we recognize a dog in an image, let alone be capable of putting it into a concrete set of instructions that a computer could interpret. This is why machine learning has become exceedingly popular over the last few decades, to deal with problems that have solutions that are hard to define. Instead of creating detailed instructions for the computer to follow such as in traditional programs, machine learning operates on the principle of trial and error. To go back to the problem of recognizing a dog, by feeding the machine learning network examples of dogs the network will over time learn to recognize dogs, without the need for any external specifications of what a dog looks like. Doing it this way of not giving the network any information on dogs or what it should focus on is called deep learning, a form of machine learning which is not dissimilar to the way we learn ourselves.

In this chapter we cover the basics of deep learning to give context to chapters to come. We do so by first introducing the architecture behind basic neural networks and subsequently going over the mathematics that underpin it. We follow up with how networks are penalized and trained with chapters on loss functions and the backpropagation algorithm, respectively. Finally, the chapter concludes with an explanation of how deep neural networks function in the context of images. We stress the importance of the convolutional layer and the significance of the concept of receptive fields.

2.1. Single-layer perceptron

The most basic version of a neural network is a single-layer perceptron and can be used to approximate a set of functions f^*, which is fairly limited as we will see. The network consists of an input layer $\mathbf{x} = \{x_1, x_2, \dots, x_m\}$ and an output layer $\hat{\mathbf{y}} = \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n\}$. In the case of a single-layer perceptron with a single output, as seen in Figure 2.1, the output is the result of a weighted combination of the inputs plus an optional bias factor. The activation function then decides whether the neuron $\hat{y}$ gets activated, or "fires". The activation function in this example is the step-function that outputs 0 if the weighted combination is less than 0 and 1 otherwise. The binary output represents whether the network considers the input $\mathbf{x}$ to belong to a certain class or not.

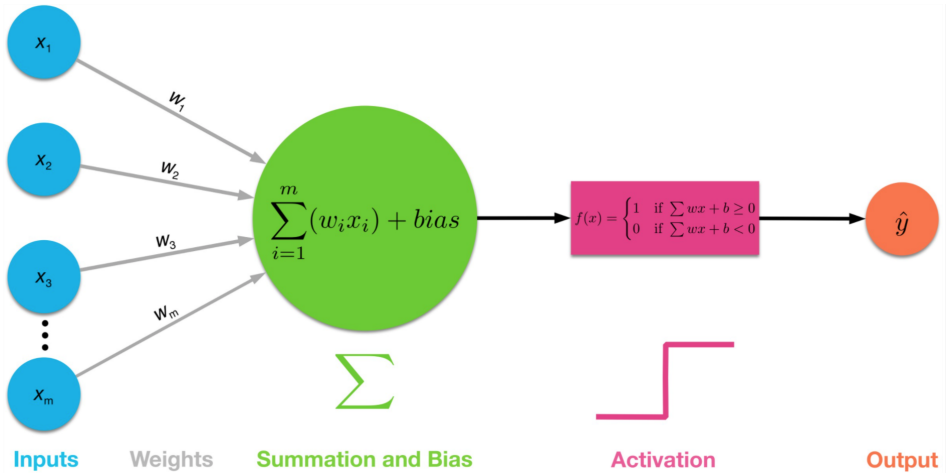


Figure 2.1: Single-layer, single class perceptron [24].

Training a single-layer perceptron is done by iteratively altering the weights based on different training inputs:

1. Initialize the weights in the weight vector $\mathbf{w} = \{w_1, w_2, \dots, w_m\}$ to 0.
2. For every sample j in the training set D, where each sample consists of an input and desired output, or target $(\mathbf{x}_j, \mathbf{y}_j)$:
   - Calculate the output of the network:
   $$\hat{y}_j = g(\sum_{i=1}^{m}(w_i x_{j,i})) \tag{2.1}$$
   where $g(x)$ is the activation function.
   - Update each weight $\mathbf{w}_i$:
   $$\mathbf{w}_{i\_new} = \mathbf{w}_{i\_old} + lr(y_j - \hat{y}_j)x_{j,i} \tag{2.2}$$
   where lr is the learning rate.

Note that if the network correctly predicts the sample's label, the weights are not updated. Step 2 can be repeated an arbitrary number of times or until a performance criteria is met. Each of these repetitions is called an epoch.

As was said, the perceptron shown in this example will only be able to distinguish between 2 classes as its output is binary. To increase the number of classes we can add neurons to the output layer from $\hat{\mathbf{y}} = \{\hat{y}_1\}$ to $\hat{\mathbf{y}} = \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n\}$, representing n possible classes. Every input will then be connected to each output neuron just as when there was only a single output neuron. Each of these connections will also have its own respective weights. Training works the same way as in the single-class case. Each output neuron is mapped to one of the classes in the dataset that it is supposed to learn and a neuron is supposed to only output 1 if the input corresponds to the class the neuron represents. If so, its weights are not updated, otherwise the weights are updated in the same way as we saw before. The downside of using the step function in this case would be that multiple neurons are able to output 1 at the same time, which is not a problem during training, but leads to ambiguity when the network is deployed. For that reason it might be useful to use a different activation function that is continuous in nature. The result of the network would then be the maximum output of all output neurons: $\hat{y} = argmax(\hat{\mathbf{y}})$. This strategy is also referred to as One-vs.-all as each neuron outputs its projected probability of the input belonging to its corresponding class versus the input belonging to any of the other classes.

Single-layer perceptrons are only able to perfectly distinguish classes that are linearly separable, meaning the classes in the dataset are separable by a hyperplane from the other classes. This limits the number of functions f^* they can approximate. To introduce non-linearity in a perceptron we have to increase the number of layers, leading to multi-layer perceptrons.

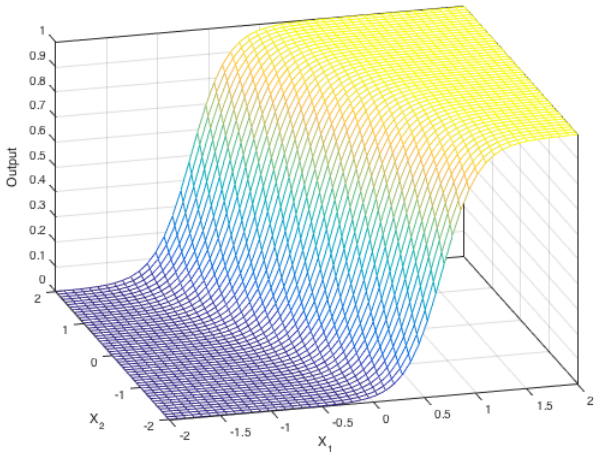


Figure 2.2: Result of applying the sigmoid function to a weighted combination of input parameters $\mathbf{x} = \{x_1, x_2\}$ [1].

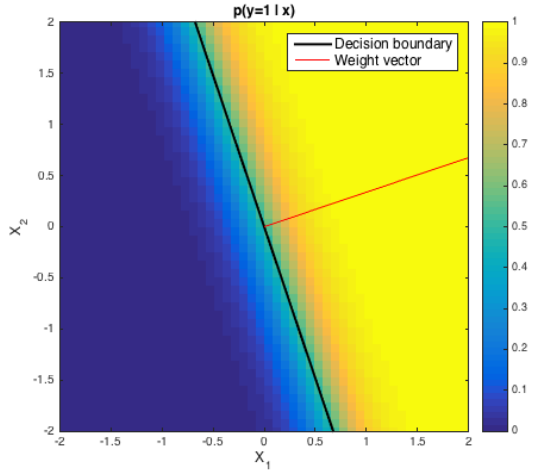


Figure 2.3: Shows a decision boundary of 0.5 on a sigmoidal shape [1].

2.2. Multi-layer perceptron

Single-layer perceptrons are only able to produce linear decision boundaries, regardless of whether the activation function is linear or non-linear. This is because the result is a weighted combination of the inputs and thus linear in the parameters. Applying a non-linear activation function such as a sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

adds a curvature to the resulting hyperplane. Figure 2.2 shows this in the case of a two-dimensional input $\mathbf{x} = \{x_1, x_2\}$. However, as the network is used to classify inputs, a threshold needs to be decided above which an input is considered to belong to class A and class B otherwise. This threshold is depicted as a line in Figure 2.3 and acts as a boundary between the two classes. Altering the weights would only stretch, squeeze and rotate the sigmoidal shape, but never contort the decision boundary. Thus, the activation function does not influence the linearity of a single-layer perceptron.

To introduce non-linearity to a perceptron, and increase the set of functions it can approximate, involves adding more intermediate neurons to the network between the input and output layers, called hidden layers. A hidden layer is defined in the same way as the output layer: $\mathbf{h} = \{h_1, h_2, \ldots, h_k\}$, except that they need not be the same length, i.e. $k \neq n$. In the case of a single hidden layer the result of the network would be:

$$\hat{y}_n = g(\sum_{i=1}^{k}(w_{i,n}h_i)) \tag{2.4}$$

$$where \quad h_i = g(\sum_{j=1}^{m}(w_{j,i}x_j)) \tag{2.5}$$

In these formulas $w_{a,b}$ refers to the weight between node a and b. Where a node can be either an input or a neuron. In essence, this says that in a multi-layer perceptron the output is a weighted linear combination of the hidden layers, which in turn are weighted linear combinations themselves of either another hidden layer or the input. If linear activation functions are used in multi-layer perceptrons it can be shown this is no different than using a single-layer perceptron and the network will again not be able to create non-linear decision boundaries. If on the other hand non-linear activation functions are used in the hidden layers, the network is able to approximate any function given enough hidden neurons. The proof for this is called the universal approximation theorem. If the network has more than around three hidden layers, we usually refer to it as a deep network, hence the term deep learning.

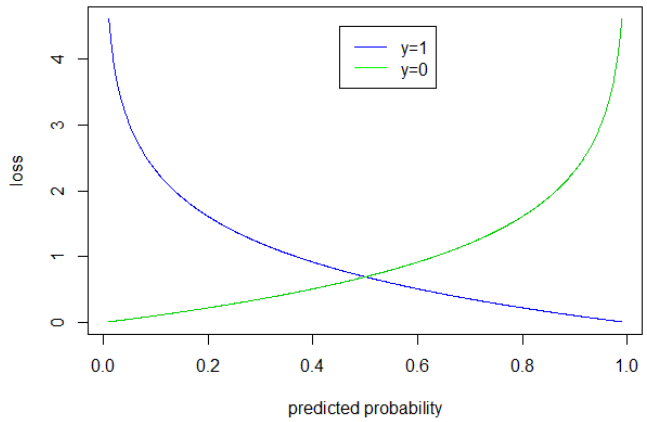


Figure 2.4: Shows the cross-entropy loss function in case of the label being 1 or 0 [46].

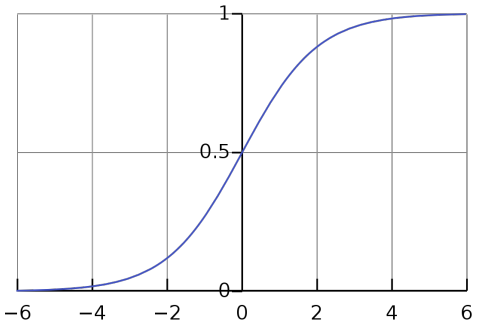


Figure 2.5: Shows the sigmoid function squashing inputs between 0 and 1.

2.3. Loss functions

In Equation 2.2 it was shown how a single-layer perceptron can be trained by iteratively updating the individual weights based on whether the network provided the correct output. This approach is wholly depended on the learning rate to determine the changes in the weights after each sample. A more nuanced approach is to treat the outcome as a minimization problem where the goal is to minimize the difference between the network's predictions and the correct labels. Gradient descent algorithms and its variants are widely used in deep learning to do exactly that:

$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta \mathcal{L}(f(\mathbf{x}; \theta_t), \mathbf{y}) \tag{2.6}$$

the equation represents the updating of the weights of the whole network given a single input sample, where θ_t are the network's weights, in the context of deep learning often referred to as parameters, at the current iteration t, γ the learning rate and $f(\mathbf{x}; \theta)$ the output of the network given an input vector $\mathbf{x}$ parameterized by the weights. $\mathbf{y}$ is the vector to denote the correct class label, it is a vector and not a scalar because it uses One-hot encoding which we will cover later. $\mathcal{L}(\cdot, \cdot)$ is the loss function which determines the penalty that the network incurs when it misclassifies samples. The chosen loss function can have a significant impact on the learning process of the network. Equation 2.6 has many similarities to Equation 2.2 but instead of simply updating the weights according to the learning rate, gradient descent updates the weights in line with the negative gradient $-\nabla_\theta \mathcal{L}(\cdot, \cdot)$ which is the vector of all partial derivatives of the loss with respect to θ.

One of the most commonly used loss functions in the context of classification is the cross-entropy loss, shown here for a binary classification setting:

$$\mathcal{L}_{CE} = -y_i log(\hat{y}_i) - (1 - y_i) log(1 - \hat{y}_i) \tag{2.7}$$

where y_i is the correct binary label of the input and $\hat{y}_i$ the prediction of the network. The advantage of using a loss function like this instead of simply taking the difference between the network's output and the label as we saw in Equation 2.2 is that we can now penalize 'very bad' outputs more than 'sort of bad' outputs. Figure 2.4 illustrates this effect, the x-axis shows the output of the network for a particular input sample and the y-axis the cross-entropy loss when the correct label is one or zero respectively: if the network predicts an output close to zero while the correct label is one, the error is much higher than if it had output a value near 0.5. The cross-entropy loss function also has better defined gradients across the board, whereas the sigmoid function as shown in figure 2.5 has increasingly smaller gradients as the output gets closer to zero or one while intuitively the gradient should be larger when the output is very incorrect. Note that the sigmoid function, or similar continuous activation functions, still often get used in the last layer of the network to squash the input between zero and one which then get fed to the loss function. In intermediate layers the ReLU activation function and its variants are a popular choice:

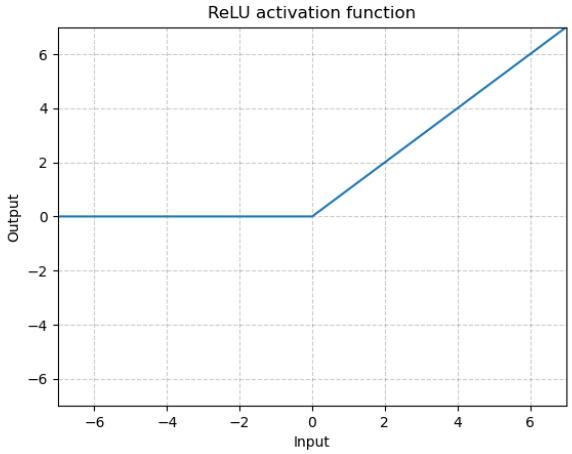


Figure 2.6: Shows the ReLU activation function [47].

$$h = max(0, a) \tag{2.8}$$

where h is the result of the hidden layer and $a = \mathbf{W}\mathbf{x} + b$, with a being the weighted combination of the layers' weights $\mathbf{W}$ and the input vector $\mathbf{x}$, plus an optional bias factor b. Figure 2.6 shows what a ReLU activation looks like, the main advantages over something like a sigmoid function are the fact that the gradient stays consistent when $a > 0$ no matter how large a becomes, whereas the gradients in the sigmoid function start to become smaller and smaller as $|a|$ gets larger, also referred to as the vanishing gradient problem. Secondly, ReLU introduces sparse results by setting every $a \leq 0$ result to zero. This helps with stability during the learning process.

In the last section it was mentioned how the sample labels are binary, even when there are often dozens of classes in a dataset. This is done by means of One-hot encoding, where given n classes, the label of an input sample is denoted by:

$$\mathbf{y} = \underbrace{(0, \dots, 0, 1, 0, \dots, 0)}_{\text{Entry i is set to 1 to denote class i}} \tag{2.9}$$

In tandem with this approach, the network's last layer will have n neurons. Instead of applying the sigmoid function to each neuron's output, a generalization of the logistic function is applied which also normalizes the outputs, called the softmax function:

$$\hat{\mathbf{y}} = softmax(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{n} e^{\hat{y}_j}} \; for \; i = 1, \dots, n \tag{2.10}$$

here $\hat{\mathbf{y}}$ represents the final output of the network, a vector of length n with values that add up to one. $\hat{y}_i$ is the output of neuron i in the final layer of the network. The value of each entry in $\hat{\mathbf{y}}$ after applying the softmax function now denotes the probability the network assigns to the input belonging to that class. Taking another look at the cross-entropy loss function $\mathcal{L}(f(\mathbf{x}; \theta_t), \mathbf{y})$ it now becomes clear how the output of the network $\hat{\mathbf{y}} = f(\mathbf{x}; \theta_t)$ and the One-hot label encoding of the input sample $\mathbf{y}$ each sum up to one and that the gradient of the cross-entropy loss increases when each entry of $\hat{\mathbf{y}}$ differs a lot from $\mathbf{y}$ which happens most often when the network predicts the wrong label: $argmax(\hat{\mathbf{y}}) \neq argmax(\mathbf{y})$. In conclusion, the network will adapt and alter its weight more when $\hat{\mathbf{y}}$ and $\mathbf{y}$ are more dissimilar, just as intuition would dictate.

2.4. Backpropagation

Updating the weights of the network in the case of a single-layer perceptron was done by adding a value to them proportional to the learning rate. When using gradient descent the weights move in the

opposite direction of the gradient of the loss function. The gradient is a vector that denotes all partial derivatives of the loss function with respect to each weight in the network:

$$\nabla_\theta \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \frac{\partial \mathcal{L}}{\partial \theta_2} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \theta_k} \end{bmatrix} \tag{2.11}$$

where $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ is the loss function with as inputs the network's output vector $\hat{\mathbf{y}}$ and the One-hot encoded class label vector $\mathbf{y}$. The gradient shows how much each weight contributes to the loss and the weights are altered accordingly during gradient descent. To calculate each derivative we can treat a network as a set of nested functions:

$$\begin{gathered} \mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y} log(\hat{\mathbf{y}}) - (1 - \mathbf{y}) log(1 - \hat{\mathbf{y}}) \\ \hat{\mathbf{y}} = g(\mathbf{z}_l) \\ \mathbf{z}_l = \theta_l \mathbf{z}_{l-1} + b_l \\ \mathbf{z}_{l-1} = \theta_{l-1} \mathbf{z}_{l-2} + b_{l-1} \\ \vdots \end{gathered} \tag{2.12}$$

where $g(\cdot)$ is any activation function, $\mathbf{z}_l$ the result of layer l, θ_l the weights at layer l and b_l the bias. When applying the derivative to a set of nested functions the chain rule applies:

$$f^{(2)}(f^{(1)}(x))' = f'^{(2)}(f^{(1)}(x)) f'^{(1)}(x) \tag{2.13}$$

Using this logic it can be shown what the partial derivatives of Equation 2.12 with respect to θ and b look like for the last layer l. For readability sake the equations are shown using Leibniz notation:

$$\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{d\hat{\mathbf{y}}} \frac{d\hat{\mathbf{y}}}{d\mathbf{z}_l} \frac{d\mathbf{z}_l}{d\theta_l} \ldots \tag{2.14}$$

$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{d\hat{\mathbf{y}}} \frac{d\hat{\mathbf{y}}}{d\mathbf{z}_l} \frac{d\mathbf{z}_l}{db_l} \ldots \tag{2.15}$$

if both of these partial derivatives are calculated separately, many nested partial derivatives become redundant as they have already been determined while calculating the other partial derivative. This observation is the essence of the backpropagation algorithm; by starting the calculation of nested partial derivatives from the back, i.e. from the input data, previous results can be reused for later calculations. This method is a form of dynamic programming and speeds up the learning process drastically. Virtually every deep learning library uses a heavily optimized version of the basics described here to alter the network's parameters during training. The only downside is that previously calculated results need to be cached for later use, taking up large portions of memory.

2.5. Convolutions

The networks we have looked at in the previous sections are multi-layer perceptrons with fully connected layers, also called dense connections, where every neuron in layer $l-1$ is connected with every neuron in layer l. Such a network has no problem taking images as inputs and performing classification. However, doing so with a fully connected network will lead to rather poor performance as a dense network is not spatially invariant. To show this, we can consider an image as a matrix of values $\mathbb{R}^{H \times W \times C}$ with W, H and C the width, height and number of image channels respectively. Each value dictates the intensity of the pixel in that channel. When $C = 1$ the image is in grey-scale. The input to a dense network is in the form of a vector $\mathbf{x}$, so a 1D matrix representing the image would have to be flattened into a vector either horizontally or vertically when being fed to the network. Either way, the relevant spatial information that is needed to classify the image is lost. Figure 2.7 shows what that would look like in the case of a horizontal flattening of the input matrix.

Conversely, a convolutional layer looks at local regions of the image by means of a striding window. During a convolution a window, also called a filter or kernel, is slid across the image transforming each scanned region into a value. The window, which itself is a matrix, is parameterized and subsequently

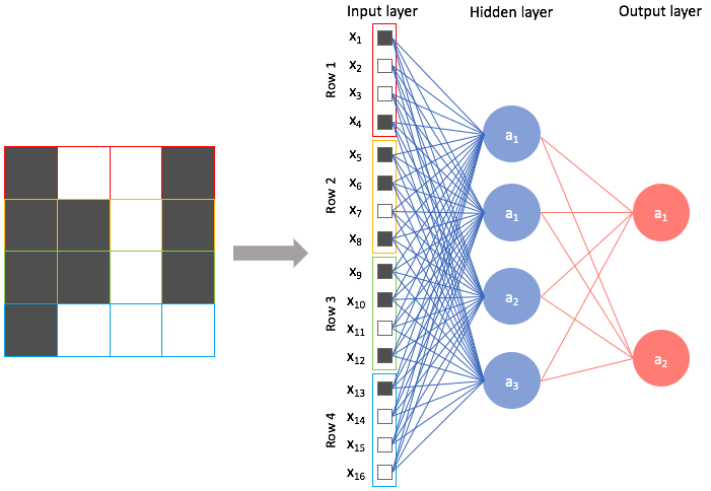


Figure 2.7: Shows how a one-dimensional image matrix is flattened horizontally to conform to the input of a fully connected network. The relevant spatial information in the image is lost in the process [22].

learned during training. The goal of the window is to scan for features, the resulting matrix from the convolution is for that reason called a feature map. An example of a convolution is shown in figure 2.8.

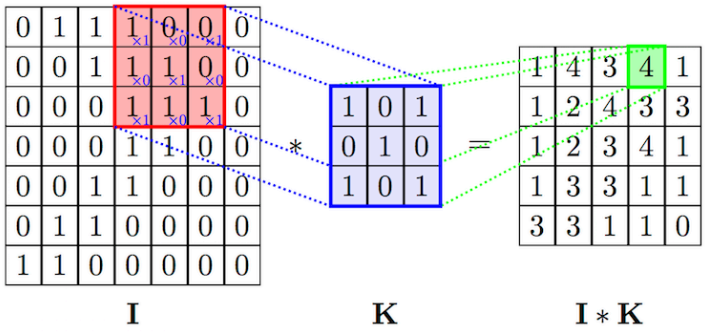


Figure 2.8: Shows the process of a convolution. I is the image or feature map of the previous layer, K the window and $I * K$ the resulting feature map [35].

The window K moves from left to right and top to bottom. After each step, or stride, it performs a convolution with the inputs. The result is a weighted combination of the inputs with the weights of the window. In other words, each input value is multiplied with the value of the window in the same position and summed up which results in the value depicted in green. Note that the resulting feature map's resolution is lower than the resolution of the input matrix because of the size of the filter. The size of the step the filter takes each time, also called stride, in this example is one, increasing the stride or increasing the size of the window lowers the resolution of the resulting feature map.

The network learns the parameters of the filters during training, intuitively the values of the resulting feature maps represent a measure of the presence of useful features found in that part of the image. If for example the network learns that straight lines are a good image statistic to classify the images in the dataset and the resulting feature map has high values in the top left corner, it signifies there is a strong straight line presence in that region of the image. Notice how this is invariant to the location of the straight lines, if there were straight lines in the bottom right corner of the images, the resulting feature map would show higher values in the bottom right corner.

By performing convolutions on previous feature maps, in figure 2.8 I would then not be the input image but a feature map, the network can learn increasingly complex features. As a result, later feature maps can "see" more of the input image than earlier feature maps. Figure 2.9 illustrates this effect: the green value in layer 2 contains information of the green region of layer 1. The yellow value in layer 3 contains information of the whole image, or feature map, of layer 1. This allows the filters in later layers to learn more nuanced features as they are able to quite literally see the bigger picture. We say

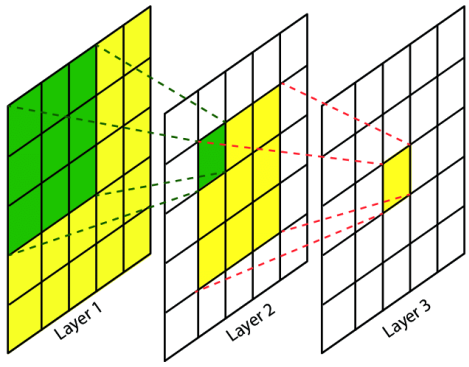


Figure 2.9: Shows how values of a feature map in later layers contain information from a larger resolution of the input as earlier layers due to the sub-sampling nature of convolutions. In this example a value from layer 3 contains information from all of layer 1, while a value from layer 2 only contains information from the green portion of layer 1 [36].

that the later layers have a higher spatial receptive field. In practice this leads to earlier feature maps learning basic shapes, such as lines and corners, while later feature maps learn complex geometries such as faces, ears, tires etc.

To get an understanding of what convolutions would look like from a architectural point of view, figure 2.10 shows how the same image from figure 2.7 is fed to a convolutional layer in a neural network. Instead of connecting every input to every neuron in the first layer, the neurons in a convolutional layer are only connected to the inputs which the filter coincides with. In this case, the filter is 2×2 and the stride is also 2. Notice also how each neuron shares the same parameters $\theta_l = (\theta_{11}, \theta_{12}, \theta_{21}, \theta_{22})$ as these represent the 2×2 filter. The outcome of the neurons in turn represent the resulting 2×2 filter map. This also demonstrates another key aspect of using convolutions as opposed to fully connected layers: sparse connectivity. Overall, the convolutional layer uses significantly less parameters to perform its function, namely 4 for the filter as opposed to 16 in the fully connected case of figure 2.7. While it may seem that less parameters leads to less capacity and less expressive power, in reality every extra parameter the network needs to learn requires more training data. This is what is known as the curse of dimensionality: an increase in dimensionality, which in this case refers to an increase in parameters, leads to sparsity of the available data requiring an exponential growth in data to counteract the effect. In other words, the more complex the function is the more parameters are needed to represent the function but also the more data is needed to learn said function. There is always a trade-off between the capacity of a convolutional neural network (CNN) and computational cost of training and deploying the network.

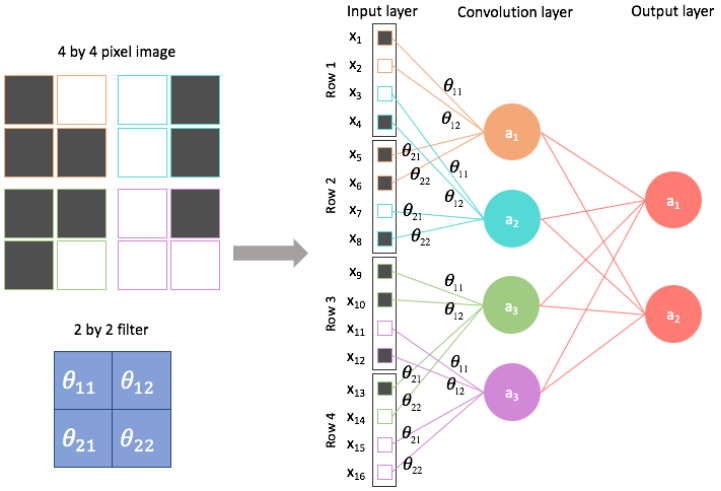


Figure 2.10: Shows how a one-dimensional image is fed to a convolutional layer with a filter of 2×2 and a stride of 2. The inputs are grouped according to how the filter slides across the image. All nodes share the same 4 parameters which represent the filter [22].

One common way to reduce the number of parameters in a CNN and at the same time increase the receptive field of a network is to introduce pooling layers. During pooling the feature maps are separated in several sections which are subsequently reduced to usually a single value according to a non-linear function. One of the most common pooling methods is max pooling, where the non-linear function is simply the max function. An example of max pooling is shown in figure 2.11 where a filter of 2×2 with a stride of 2 is used. Max pooling is a form of non-linear down-sampling and it collapses the relevant information to a lower resolution. The intuition behind it is that the resulting feature map retains the relevant local spatial information while discarding less relevant information, effectively reducing the number of parameters later in the network and increasing the receptive field of the resulting feature maps.

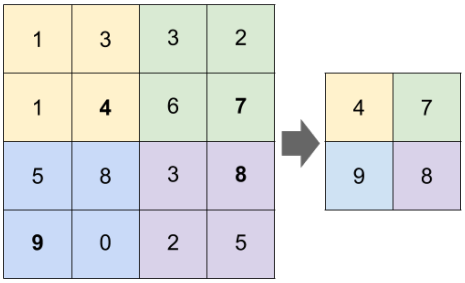


Figure 2.11: Shows the max pool function of a 2×2 filter with a stride of 2 [6].

The architecture of a convolutional neural network can be viewed as a combination of two parts: the feature extraction part and the classification part. Figure 2.12 shows the architecture of VGG16, a powerful image classifying network. The feature extraction is done by the convolutional layers which learn what features are relevant for the task at hand. These layers are comprised of the convolutional layer itself, a ReLU activation function and batch normalization. The term ”layer” is overloaded in the context of deep learning; often when referring to a convolutional layer, what is actually meant are the convolutional layer, the activation function and batch normalization combined, in the image shown in black. We will see later how the term layer can even include more elements. For now though, a number of convolutional layers are usually followed by a pooling layer, shown in red. Several of these convolutional layers and a pooling layer may be referred to as a block. Several blocks are stacked together to create the feature extraction part of the CNN. The classification part, more conventionally referred to as just the classifier, consists of several fully connected layers with intermittent ReLU functions, denoted by the blue blocks in the image, and finally a softmax function shown in brown.

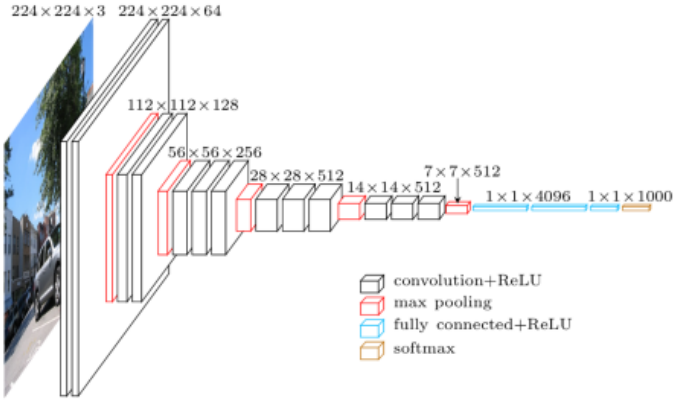


Figure 2.12: Shows the architecture of the VGG16 network. The white and red blocks represent the feature extraction part of the network while the blue and brown blocks denote the classifier. The network takes images of $224 \times 224 \times 3$ as input and outputs a probability vector of 1000 classes [51].

Notice how the input images are $224 \times 224 \times 3$ and every subsequent block down-samples the resolution but increases the number of feature maps, starting at 64 and ending at 512. As the receptive field of

the feature map grows, the complexity of the shapes the recognize increases and thus the number of shapes they can represent grows as well. The earlier feature maps only represent a few basic shapes and not many feature maps are needed to represent them. For this reason most modern CNN's increase the number of feature maps along the depth of the network. Finally, the softmax function at the end of the classifier has a size of $1 \times 1 \times 1000$ from which we can tell that the dataset the network is attempting to classify contains 1000 classes. These one-dimensional architectures, where all information flow takes place from left to right will henceforth be referred to as conventional or traditional convolutional neural networks as opposed to dynamic neural networks, which we will discuss in the next chapter.

3

Adaptive inference

Conventional neural networks have static computational graphs during inference, meaning the number of parameters do not change during the use of the network. Consequently, the amount of flops used by the network to come to a conclusion also does not change if the input dimensions stay consistent. While this computational robustness has its advantages in certain scenarios, not every input is consistent in terms of inference complexity. For this reason, performing adaptive inference on static neural networks, or in other words, turning them into dynamic neural networks, can lead to a number of potential advantages. The most notable favorable properties of dynamic networks are higher computational efficiency and more lenient computational flexibility.

This chapter covers dynamic neural networks and the inference settings they can operate in. In particular, we address how early-exiting works and the different ways it can be implemented, ending on Multi-Scale Dense networks which are optimized for adaptive inference. At the end, we go over how we define the concept of overthinking, what its consequences are and the motivation behind trying to understand it. Finally, we show how overthinking can be reduced.

3.1. Inference setting

Dynamic neural networks, as opposed to static ones, can alter their parameters or architecture on the fly. This allows them to selectively adapt to the complexity of the inputs during inference. By allocating less computation to less complex samples, dynamic networks can save time and resources on tasks where static neural networks would spend a fixed amount of computation on each sample. Furthermore, this computational adaptiveness leads to dynamic neural networks having the ability to be used in two inference settings that are out of reach for static networks: anytime prediction and budgeted batch classification. The goal in both settings is to maximize performance under a limited computational budget.

Anytime prediction. During anytime prediction, the network is asked to output a prediction given a finite positive computational budget B that exists for each sample $\mathbf{x}$ of test set D_{test}. The budget is only given during inference, varies from sample to sample and is non-deterministic. We can therefore model the event of a sample and its respective budget as a joint distribution $P((\mathbf{x}, \mathbf{y}), B)$. We assume x and B are not independent in most real life scenarios. As an example of such a situation, consider a self driving car containing a system that classifies traffic signs while driving. As soon as the system recognizes that a sign has appeared on the road in front of the car, the system will need to classify the type of sign to potentially alter the car's driving behaviour. If the system recognizes the appearance of a sign that is far down the road while the car is going at a moderate speed, the computational budget for the system to classify the sign is relatively high. Conversely, when the car is going fast down the same road, or a sign is spotted to be close to the car already, the computational budget might become significantly lower. The objective of the system is to output a prediction regardless of the computational constraints. The budget in this example is correlated with $\mathbf{x}$ as the closer the sign is, the easier it is to identify. At the same time, if a sign is close to the car it often needs to be identified quickly, so B

would be small. We let the loss of a network given a budget be:

$$\mathcal{L}(f(\mathbf{x}, B), \mathbf{y}) \tag{3.1}$$

Where $\mathcal{L}(\cdot)$ is any appropriate loss function, $f(\mathbf{x}, B)$ the network's prediction that depends not only on the input $\mathbf{x}$, but also the budget B associated with $\mathbf{x}$. $\mathbf{y}$ is the One-hot vector representing the sample's label. The goal of a network during anytime prediction is to minimize the expected loss under the joint distribution of the input and its budget:

$$\mathcal{L}(f) = \mathbb{E}[\mathcal{L}(f(\mathbf{x}, B), \mathbf{y})]_{P((\mathbf{x},\mathbf{y}),\mathbf{B})} \tag{3.2}$$

In practice, this is done by taking the average loss over all samples from the test set distribution $P((\mathbf{x}, \mathbf{y}), B) \in D_{test}$.

Budgeted batch classification. During budgeted batch classification the network is tasked with classifying all samples of a test set D_{test} given a computational positive finite budget B. This budget may be non-deterministic but is known in advance of inference. We denote the cumulative loss of the learner over the whole test set as:

$$\mathcal{L}(f(D_{test}, B), Y) \tag{3.3}$$

where $L(\cdot)$ is any appropriate loss, $f(D_{test}, B)$ an array representing all outputs of the classifier f over test set D_{test}, B the total computational budget granted to classify all samples and Y the label matrix of all samples. If we assume $|D_{test}| = M$, the model ideally spends less than $\frac{B}{M}$ of its computation on 'easy' samples and more than $\frac{B}{M}$ on 'hard' samples.

3.2. Early-exiting

Perhaps the most intuitive way of reducing computation of a network in an adaptive inference setting is to cut off inference of a sample as soon as the network is able to come up with a prediction with sufficiently high confidence, called early-exiting. If we view convolutional neural network architectures as a computational graph, the input data passes through a number of convolutional layers that look for relevant features, these features are then passed on to one or more fully connected layers that determine the likelihood of these features belonging to any of the relevant classes in the dataset. The idea is to not apply every layer in the network on the input data in line with the complexity of the input sample. If the input sample represents an 'easy' canonical data point, the full capacity of the network is not required to come to an accurate prediction. However, cutting of computation early leads to a number of obvious architectural issues including the absence of a classifier halfway along the network and the lack of sufficiently processed features to perform classification on. Not to mention the challenge of determining when a classifier is confident in its prediction and computation should be terminated. Inherently, early-exiting and traditional convolutional neural networks are at odds with each other; CNNs represent a linear process of feature extraction and subsequent classification which is heavily reliant on said features. Early-exiting interrupts this process and if implemented incorrectly can lead to poor performance.

In this section we shed some light on how early-exiting can be implemented. Subsection 3.2.1 addresses the architectural challenges of implementing early-exiting in neural networks and also covers some related work in this field. Subsection 3.2.3 discusses the second major challenge of early-exiting, namely how to ascertain when a network has performed enough computation to reach a trustworthy prediction. Finally, subsection 3.2.2 gives a detailed overview of how multi-scale dense networks operate, the main early-exiting network archetype that underpins the research laid out in upcoming chapters.

3.2.1. Architecture

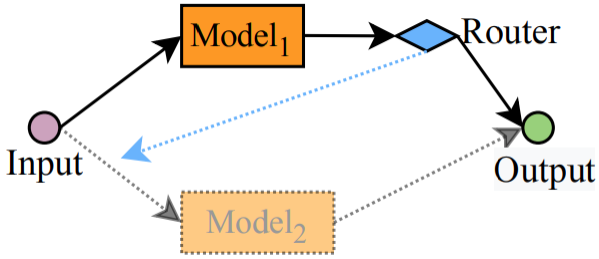


Figure 3.1: Shows the cascade of two models, where model 1 is the efficient, smaller model. The router is the abstract portrayal of some decision function that determines whether the output of model 1 suffices or if model 2 should be invoked. The cascade can be extended with as many models as desired [16].

Adding early-exiting capabilities to a traditional static neural network comes with a number of hurdles to overcome to maintain adequate performance in the face of limited computational budgets. It is then maybe no wonder that early-exit precursors chose to avoid those complications. Arguably the most straightforward way of acquiring early-exit abilities is by cascading several neural networks of increasing depth and performance. In the case of two networks, inference is first done by the smaller, efficient network. If the prediction of this network is deemed adequate, the prediction is returned as the final output. Otherwise, inference is done using the deep and slow network and its prediction is returned as the final result. Figure 3.1 shows an example of a two model cascade. In most cases the smaller network suffices and the bigger network is only required for the more complex inputs [44]. This scheme can be extended with multiple networks, where the inference is done by each model in a set order from smallest network to hardest [56], or a more nuanced approach is taken and certain models can be skipped if desired [4]. In the latter case a network choosing policy is trained to determine which network should be used next for inference if the current one does not suffice. Determining when a prediction suffices or when to perform extra inference is a non-trivial ordeal and is discussed in subsection 3.2.3. Implementing early-exiting by means of cascading several networks avoids architectural complications and can be done with virtually any CNN [28, 52, 50, 32, 17], granted they differ in performance and efficiency enough to make it worthwhile. Also, we assume Pareto optimality when choosing the networks for the cascade; if one of the networks is dominated in terms of both performance and efficiency, it adds no benefit to being in the cascade and should be left out.

The simplicity of cascading networks is an attractive attribute, but it comes with a number of notable downsides. The first being an excess of possible redundant computations. If the prediction of model 1 is not considered accurate enough, the next model in line has to perform inference from scratch. It is likely that both models will look for similar features in the input image, especially in the early layers of the network. Think for example of simple geometry like lines, corners and basic shapes. The second model may very well benefit from prior computation performed by the previous model. Secondly, using fully fledged pre-trained models hinders the early-exiting granularity. To gain a more fine grain early-exit granularity requires more, smaller networks that in terms of efficiency and performance are closer together. While this can be problematic in and of itself as it depends on the available models, it also runs directly contrary to the computational redundancy we just discussed. The closer the chosen models lie in terms of efficiency and performance, the more overhead the cascade will incur in the way of overlapping computations effectively reducing the ability of cascades to perform well in budgeted batch settings.

Cascades are better suited for the anytime prediction setting, however also here does the lack of early-exiting flexibility limit their use cases. Many practical instances require fine grain granularity when it comes to early exiting. If we consider the case of self-driving cars again, there are an unbounded amount of different budgets the early-exiting setup has to deal with as the speed of the car and distance to the signs are continuous variables. During anytime prediction, the result of the cascade, and by extent any early-exiting setup, can be described as follows:

$$\begin{aligned} \hat{\mathbf{y}} &= m_i(\mathbf{x}) \\ \textbf{s.t. } i &= argmax(P(M)) \\ T(m_i(\mathbf{x})) &\leq B \end{aligned} \tag{3.4}$$

where $\hat{\mathbf{y}} = m_i(\mathbf{x})$ is the cascade's output given by the prediction of model i on input $\mathbf{x}$. Model i is the model with the highest performance $P(\cdot)$ of all models in the cascade M. Finally, this model's efficiency has to satisfy the budget constraint B. In other words, in practical instances, an early-exit setup will output its best prediction that can be computed within the computational budget. If the cascade only has a limited number of models, it can happen that the budget is slightly smaller than the budget required to obtain a prediction from a well performing model, and the cascade has to make do with the model that precedes it. This model may perform significantly worse while part of the budget is left unused. For this reason, cascades depend heavily on how many pre-tained models it contains that fit well together in terms of efficiency and performance to do well in an anytime prediction setting.

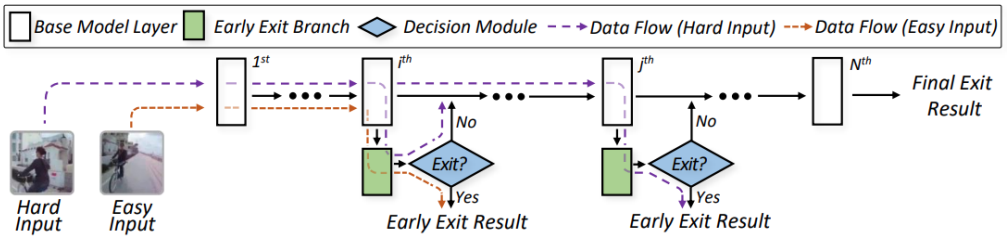


Figure 3.2: Shows the implementation of a convolutional neural network with several early-exit branches containing intermediate classifiers. Each prediction is fed to a decision module which determines whether the next part of the network should be invoked or to output the current prediction. Ideally, easier samples end up in earlier classifiers while more computation is spend on harder samples [11].

Models in cascades can not make use of previous computations leading to large computational overhead for every model invocation. This in turn prevents cascades from acquiring high early-exit granularity. A way to reduce computational overhead for every subsequent prediction is to let every subsequent classifier make use of previous computation. This naturally leads to a singular network design with multiple intermediate classifiers, an interpretation of such a design can be seen in figure 3.2. Several branches are connected to a network backbone which is shared by each classifier. These branches contain a number of extra convolutions, usually to downsample the feature maps and prepare them for classification. After a classifier at the end of the branch has made a prediction, the decision is made whether to accept the prediction or to proceed to the next classifier. When the prediction is rejected, the next part of the backbone network including the upcoming classifier is invoked until either the final classifier is reached or an adequate prediction is found. Note how the network computes only as much as it needs to and previous convolutions are used to feed the next part of the network.

The earlier convolutional layers of any chosen backbone model tend to have a low receptive field and as a consequence their derived features may not be suitable to attain high performance over a whole dataset. However, this concept of intermediate early-exits operates under the assumption that many datasets contain samples that can already be classified using features that are derived in those early layers [54, 31, 25, 11]. As long as the branches themselves containing the classifier do not require too much computational overhead, there is no harm checking if early and intermediate feature maps are already suitable for classification. If so, significant computation can be saved, if not, later layers might be able to come up with an accurate prediction regardless. The early-exit granularity is only limited by how many layers the backbone network has as an exit-branch can be connected after each layer. While this may not be ideal because of reasons related to training as we will see, it should be clear how we now have more control over what the early-exiting strategy looks like as opposed to when using model cascades.

Training a cascade of models involves training each model individually, as none of the models share parameters. In a network with multiple classifiers, all classifiers share parameters. We can train each

classifier in the network iteratively and individually, in that case the loss for each classifier would be:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}_{CE}(f_i(\mathbf{x}; \theta_i), \mathbf{y}) \tag{3.5}$$

where $\mathcal{L}_{CE}(\cdot)$ is the cross-entropy loss and $f_i(\mathbf{x}; \theta)$ the classifier at depth i applied to the input using its respective parameters θ_i. After the parameters that are associated with classifier i are updated, we can do the same for classifier j until we reach the final classifier. While this is a valid training scheme, the classifiers share parameters and updating them in favor of one of the classifiers may hinder the progress of another. A conflict in the optimization goal for the convolutional layers of the backbone will arise, namely to create filters that look for discriminative features for a nearby classifier while simultaneously maintaining information to create complex features for the later classifiers. It is therefore almost always advantageous to update the parameters in favor of all classifiers at the same time by means of a weighted cumulative loss function:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_i \gamma_i \mathcal{L}_{CE}(f_i(\mathbf{x}; \theta_i), \mathbf{y}) \tag{3.6}$$

here the loss is a summation of the losses of each individual classifier's prediction. γ_i is an optional factor that can be used to penalize specific classifiers more than others. Training the network this way changes the goal from maximizing the performance of each individual classifier in a vacuum to maximizing the average performance of all classifiers at the same time. The network benefits in both the anytime prediction and budgeted batch classification setting when jointly optimizing each classifier's subnetwork.

A vanilla backbone model with intermediate classifiers is already well suited for adaptive inference. Some 'easy' samples can be correctly classified with simple feature maps and a joint training scheme helps update the parameters in favor of all classifiers simultaneously. However, even simple samples benefit from more complex feature maps with larger receptive fields and classifiers do still tend to interfere with each other to some extent when trained jointly. Subsection 3.2.2 addresses how improvements can be made on the concepts laid out in this section, giving rise to hand-tuned end-to-end designed early-exiting architectures such as that of the Multi-scale Dense Network.

3.2.2. Multi-Scale Dense Networks

Injecting traditional models with intermediate classifiers trained in a joint fashion instills them with the ability to perform adaptive inference. Many datasets contain 'easy' samples that can already be correctly identified using primitive features present in the earlier stages of the network. However, if the classifiers in the early layers had access to coarser features, i.e. features with a higher receptive field, they would be more accurate in their predictions. While this may come across as stating the obvious, implementing it in a resource-aware way to be used in an early-exiting setting is non-trivial. Secondly, adding classifiers to regular networks can harm the effectiveness of classifiers positioned deeper in the network. In the last section we discussed how jointly training the classifiers in networks reduces the prevalence of optimization conflicts in the backbone network. In reality, earlier layers still tend to collapse in favor of the nearest classifiers, forsaking deeper layers in the process. Multi-scale Dense Networks (MSDNets) are an architectural archetype that address the lack of coarse level features in the earlier layers and the interference of intermediate classifiers by introducing multi-scale feature maps and dense connections, respectively [20]. MSDNets are considered state-of-the-art when it comes to architectures that specialize in resource-aware adaptive inference [30, 16].

Multi-scale architecture. Conventional CNN's process information from a fine to coarse scale. The early layers only see local details in an image while later layers mainly see features with full spatial context. In most state-of-the-art deep CNN's, regional information is propagated slowly across the spatial dimension of the feature maps in line with the growth of their receptive field. Forcefully increasing the receptive field too fast may lead to loss of crucial information needed to construct complex features. One school of thought suggests that the concept of a receptive field in the traditional, linear sense is outdated [26, 21]. Instead, spatial information exchange that leads to spatial shift invariance can be achieved with significantly fewer parameters if features maps of all scales could transfer information directly as opposed to only along the depth of the network. Figure 3.3 shows how this concept relates to standard CNNs. Scales normally represent the size of the feature maps and simultaneously its receptive

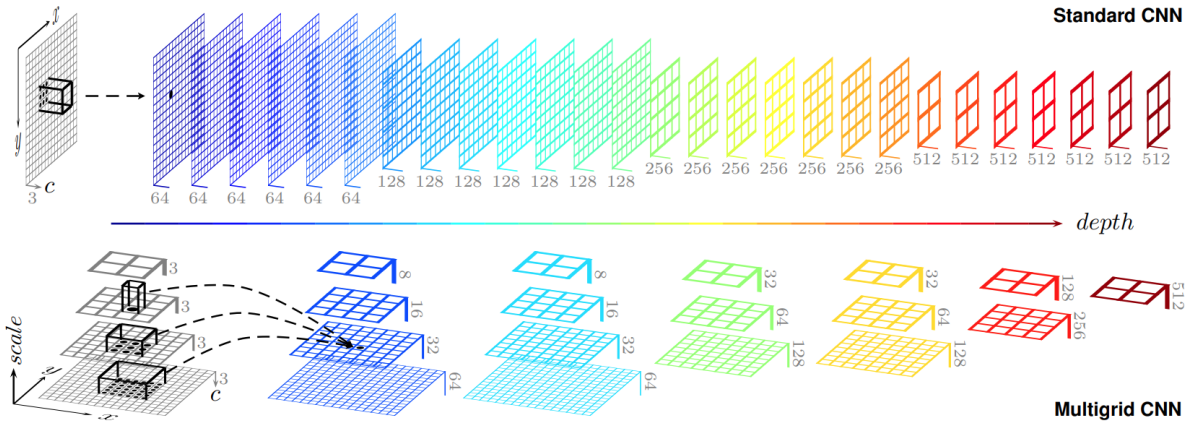


Figure 3.3: Shows a conventional CNN at the top. Only the deeper layers have full receptive field and propagating spatial information across feature maps takes many steps. The bottom shows the concept of a multigrid, or multi-scale, implementation. Scales are no longer linked to the depth of the network, instead every layer contains several scales that each contain information from every scale in the previous layer, incentivizing information transfer between coarse and fine scales [26].

field. The deeper a layer is positioned, the larger its receptive field on average and the smaller the size of the feature maps due to pooling operations. In multi-scale architectures, such as that of the multigrid CNN, all feature scales are already present in every layer, turning the one-dimensional architecture into a two-dimensional one. Depth now exists to exchange information between every scale and refine the feature maps. Mutltigrid does so by means of a multi-dimensional convolution along the scales of the scale pyramid. As each feature map in layer i consists of information of every feature map of layer $i-1$, this setup is reminiscent of an architecture with fully connected layers. Nevertheless, Multigrid uses significantly fewer parameters while maintaining performance when compared to traditional state-of-the-art CNNs.

Multi-scale setups have also been utilized in the research of shift invariance in CNNs [39] and the image segmentation setting [64], where in the latter case they also made use of upscaling and downsampling between feature maps in different layers. Taking the concept of multi-scale architectures to the extreme leads to setups with a fractal design where instead of a hierarchical and targeted architecture the network consists of many structured, interconnected, repeating convolutional layers that exchange information by upscaling, downsampling, pooling and convolutions. These type of networks essentially contain many different models within depending on which route is chosen through the network [48, 29].

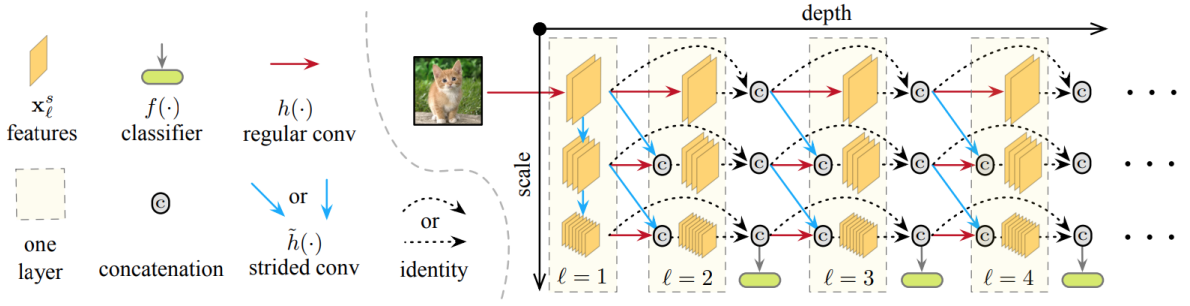
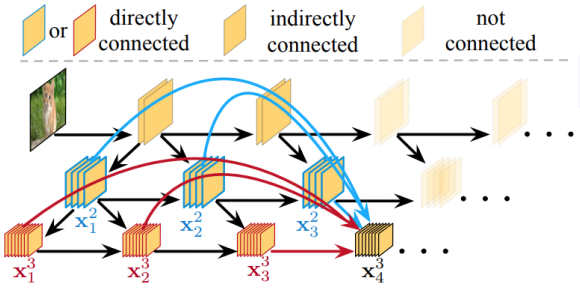


Figure 3.4: Shows the architectural overview of a basic Multi-Scale Dense network. It represents features of every scale along the depth of the network. All information flow is directed up the scales and deeper into the network. Classifiers are attached to the coarsest feature maps which represent complex features. After every convolution, the result is added to a concatenation, which is propagated along the depth of the network for every scale [20].

MSDNet employs a multi-scale setup not for the reasons we just covered, but specifically to maintain coarse-level features during every stage of the network. Figure 3.4 shows what the architecture of a MSDNet looks like. It is perhaps most similar to that of a Multigrid setup, but the main difference is the insight of MSDNets that it only needs the most accurate features at the highest scale, as this

is where the classifiers are connected to. Note that highest scale in this context means the coarsest features, portrayed as the bottom row in the image, normally associated with the depth of the network in traditional CNNs. Focusing predominantly on the coarsest features allows the information flow to only be directed to the bottom right; there is never a flow of information upwards. The first layer of the network is the only one that also includes direct vertical connections used for 'seeding' every scale. In many ways the first layer acts as a regular convolutional network with aggressive downsampling capabilities. The initial scale representations are subsequently refined along the depth of the network. Blue connections represent convolutions with a large stride that lead to downsampling of the resulting features maps; the first layers uses them to quickly present feature maps of every scale. The red connections are regular convolutions with padding, which creates feature maps of the same size as the input. It is worth mentioning that the classifiers denoted in figure 3.2.2 also contain convolutions to downsample the feature maps and prepare them for classification.



x_l^s	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s = 1$	$h_1^1(\mathbf{x}_0^1)$	$h_2^1(\mathbf{x}_1^1)$	$h_3^1(\mathbf{x}_1^1, \mathbf{x}_2^1)$	$h_4^1(\mathbf{x}_1^1, \mathbf{x}_2^1, \mathbf{x}_3^1)$
$s = 2$	$\tilde{h}_1^2(\mathbf{x}_1^1)$	$\begin{bmatrix} \tilde{h}_2^2([\mathbf{x}_1^1]) \\ h_2^2([\mathbf{x}_1^2]) \end{bmatrix}$	$\begin{bmatrix} \tilde{h}_3^2([\mathbf{x}_1^1, \mathbf{x}_2^1]) \\ h_3^2([\mathbf{x}_1^2, \mathbf{x}_2^2]) \end{bmatrix}$	$\begin{bmatrix} \tilde{h}_4^2([\mathbf{x}_1^1, \mathbf{x}_2^1, \mathbf{x}_3^1]) \\ h_4^2([\mathbf{x}_1^2, \mathbf{x}_2^2, \mathbf{x}_3^2]) \end{bmatrix}$
$s = 3$	$\tilde{h}_1^3(\mathbf{x}_1^2)$	$\begin{bmatrix} \tilde{h}_2^3([\mathbf{x}_1^2]) \\ h_2^3([\mathbf{x}_1^3]) \end{bmatrix}$	$\begin{bmatrix} \tilde{h}_3^3([\mathbf{x}_1^2, \mathbf{x}_2^2]) \\ h_3^3([\mathbf{x}_1^3, \mathbf{x}_2^3]) \end{bmatrix}$	$\begin{bmatrix} \tilde{h}_4^3([\mathbf{x}_1^2, \mathbf{x}_2^2, \mathbf{x}_3^2]) \\ h_4^3([\mathbf{x}_1^3, \mathbf{x}_2^3, \mathbf{x}_3^3]) \end{bmatrix}$

Figure 3.5: Shows how feature maps are concatenated and propagated though the network. l refers to the layer, s the scale and $h_l^s(\cdot)$ and $\tilde{h}_l^s(\cdot)$ to convolutions and strided convolutions, respectively. $[...]$ denotes the concatenation operator. The information on the right shows horizontal and vertical concatenations to differentiate between the concatenation of feature maps along the depth and scales, respectively. In reality, all concatenations take place along the channel dimension of the feature maps. The size of the concatenation grows linearly when $s = 1$ and according to $2k(l-1)$ when $s > 1$ [20].

Dense connectivity. When attaching intermediate classifiers to existing state-of-the-art models, joint optimization of the classifiers reduces the extent of earlier feature maps collapsing in favor of earlier classifiers. However, even so, earlier classifiers tend to interfere with later classifiers. The creators of MSDNet showed this by adding intermediate classifiers to both ResNet [17] and DenseNet [19]. The final classifier loses significant performance when intermediate classifiers are added in both networks. The effect is more pronounced when the first intermediate classifier is placed closer to the start of the network. This suggests that convolutional layers will collapse to tailor to upcoming classifiers in close proximity and not retain information that is needed to optimize the deeper layers. What is interesting to note here is that this effect is stronger in ResNet than DenseNet. DenseNet introduces dense connections, where layers belonging to the same block are all interconnected. So every convolutional layer takes as input the concatenation of every preceding feature map in a given block, while simultaneously feeding every successive layer, leading to $\frac{L(L+1)}{2}$ connections instead of $L-1$ connections in a traditional layout. Dense connections improve information flow between layers which, among other benefits, stimulates feature re-use and reduces parameters as fewer feature maps are needed per convolution. The increased information flow is also most likely the cause for the reduced loss in performance for later classifiers when intermediary classifiers are introduced. Dense connectivity makes it so early stage feature map information still ends up in later layers as they bypass intermediary transformations, preventing information loss due to the premature collapsing of feature maps. MSDNets implement dense connectivity on a scale by scale basis as can be seen in figure 3.5. If $s > 1$, the input to the convolution $h_l^s(\cdot)$ at scale s and layer l is the concatenation of all preceding feature maps $\{\mathbf{x}_1^s, \ldots, \mathbf{x}_{l-1}^s\}$ with all preceding feature maps of $s-1$ $\{\mathbf{x}_1^{s-1}, \ldots, \mathbf{x}_{l-1}^{s-1}\}$. This is visualized in the diagram with red and blue respectively. Note that the feature maps of $s-1$ are again connected to and thus influenced by $s-2$. In figure 3.4 dense connectivity is visualized with the dashed lines. Not every dense connection is visualized as this takes place implicitly via recursive concatenations. In essence, each feature map at layer l is added to a concatenation that is propagated along every scale s: $\{\mathbf{x}_1^s\} \rightarrow \{\mathbf{x}_1^s, \mathbf{x}_2^s\} \rightarrow \{\mathbf{x}_1^s, \mathbf{x}_2^s, \mathbf{x}_3^s\} \rightarrow \ldots$. $\mathbf{x}_0^s$ is not added as it either does not exist at $s > 1$ or it represents the input image at $s = 1$. The number of feature maps a convolutional layer takes in at layer l and $s > 1$ is equal to $2k(l-1)$ where

k denotes the fixed number of output channels set for every convolution, also referred to as the growth rate. Usually the number of channels in a deep neural network grows exponentially, as we saw in figure 2.12. Networks with dense connectivity on the other hand can inhibit the growth in channels hence the reduction in parameters compared to other state-of-the-art models.

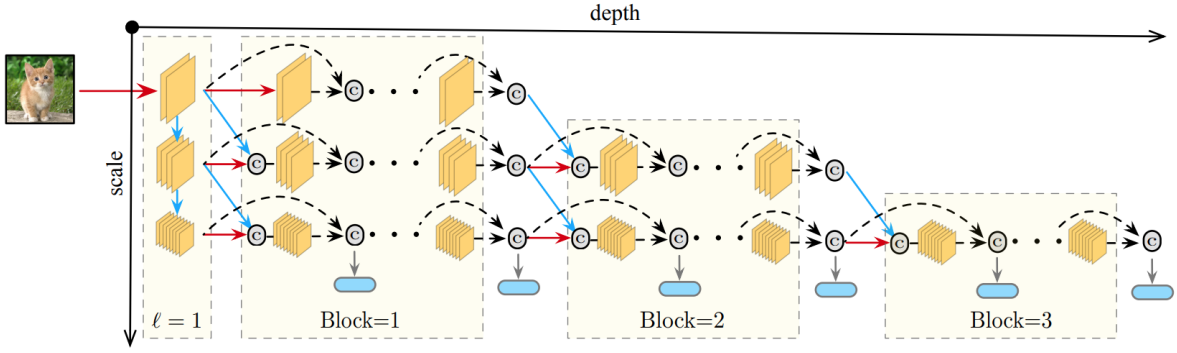


Figure 3.6: Shows a 3 block MSDNet with pruned subsections. The operations between blocks consist of 1×1 convolutions [20].

To further reduce parameters, MSDNets omit scales when they no longer contribute significantly to upcoming classifiers. Figure 3.6 shows what that would look like. Also, several layers are grouped together to form a block. Blocks are connected by means of a 1×1 convolution that halves the number of resulting feature maps. This has been shown to reduce computation without sacrificing performance [17, 53].

3.2.3. Exiting policies

During adaptive inference, processed features make their way to intermediate classifiers in the case of networks with early-exits, or to the final classifier of networks in the case of model cascades. Either way, in a budgeted batch setting, a decision has to be made at each stage whether the prediction of the current classifier suffices or if more computation is required to potentially reach a more accurate verdict. If the decision rule is too tight, inaccurate predictions might get returned. If the decision rule is too loose, unnecessary computation might be spent when previous classifiers already provided the correct answer. There are several methods to decide when to early-exit, each with their own advantages and shortcomings. In this section we will mainly focus on confidence based criteria as that is what we will use in our experiments. We touch on policy networks in section 4.1 as we aim to create a similar method to learn an early-exiting strategy directly.

Confidence based criteria rely on interpreting the output of the final layer of a classifier and comparing it to a predefined confidence threshold. If the criterion exceeds the threshold, the sample is exited at that classifier, otherwise computation resumes. As this is reminiscent of the stopping problem, some tangential methods are more closely related to reinforcement learning, where some sort of halting score is introduced [14, 12, 9, 15]. The halting score increases based on every prediction and when it crosses 1, the computation is stopped and the current classifier outputs its prediction. These type of methods often introduce additional modules that need to be added to the network for them to function. Regular confidence based criteria can be used without making changes to the network and are conceptually simple, making it a popular choice. The actual criterion used to determine the confidence of a prediction varies. The most commonly used statistic, and the one we use, is simply the maximum value of the softmax output [20, 61]:

$$\begin{aligned} \mathbf{z} &= f_i(\mathbf{x}) \quad i \in \{1, ..., n\} \\ \hat{\mathbf{y}} &= softmax(\mathbf{z}) \\ c &= max(\hat{\mathbf{y}}) \end{aligned} \tag{3.7}$$

where $\mathbf{z}$ is the output of classifier i, n the total number of classifiers in the network and c the measure of confidence in the prediction between 0 and 1. Another measure of confidence is the entropy of the

softmax output [60, 54]:

$$c = \sum_{j \in |\hat{\mathbf{y}}|} y_j log(y_j) \tag{3.8}$$

where y_j is the probability the classifier assigns to class j. High entropy indicates that the softmax values differ greatly and the classifier has ended up with assigning a high probability to a single class. Low entropy signifies that the classifier is unsure as class probabilities are closer in value. Lastly, a measure that is sometimes used is the ratio between the highest and second highest softmax values [23, 2]. It aims to do the same as the entropy measure, but in a more concrete way.

Regardless of which confidence criterion is used, a confidence threshold t_i needs to be determined for each classifier f_i. Let C_i denote the cost of invoking the network up to and including classifier i, q_i the probability of a sample exiting at classifier i and D_{test} the test set. Note that $\sum_i q_i = 1$. Given a budget constraint B during budgeted batch classification gives rise to the following inference constraint:

$$|D_{test}| \sum_i q_i C_i \leq B \tag{3.9}$$

The goal is to find the thresholds such that this constraint is met. There are many sets of $q_i \in Q$ that will satisfy the constraint. For this reason it is often easier to come up with several Q first and subsequently calculate the associated budget on the validation set D_{val}. Given Q, the resulting thresholds and the budget associated with them can be stored in a lookup table. In a real life budget batch scenario, the appropriate thresholds can then be picked that satisfy the budget constraint.

Algorithm 1 Pseudocode for the creation of confidence thresholds given a validation set D_{val}, a set of classifiers in a network F and a number of iterations *iter*.

1: **procedure** CREATETHRESHOLDS(D_{val}, F, *iter*)
2: $n \leftarrow |F|$
3: $m \leftarrow |D_{val}|$
4: $T[n] \leftarrow []$ ▷ Thresholds
5: $Q[n] \leftarrow []$ ▷ Exit probabilities
6: $C[n][m] \leftarrow []$ ▷ Confidence scores on all val samples for each classifier
7: **for** $i \leftarrow 1$ to n **do**
8: $C[i] \leftarrow max(softmax(f_i(D_{val})))$
9: $b \leftarrow 20$
10: **for** $p \leftarrow 1$ to *iter* **do**
11: $p \leftarrow p/b$
12: $Q \leftarrow e^{(log(p)*\{1,\ldots,n\})}$
13: $Q = Q/sum(Q)$ ▷ $\sum_i q_i = 1$
14: **for** $i \leftarrow 1$ to n **do**
15: $max_c = argmax_{C'_i \subset C[i], |C'_i| = Q[i]*m} \sum_{c \in C'_i} c$ ▷ Select samples with largest confidence until we reach exit quota q_i
16: $T[i] \leftarrow min(C[i][max_c])$ ▷ Threshold is lowest confidence that still exits
17: **if** i < n - 1 **then**
18: $C[i+1] \leftarrow C[i+1] \backslash max_c$ ▷ Remove samples from eligible samples for upcoming classifiers as they have already exited at this classifier
19: **return** T

Algorithm 1 shows the pseudocode for the derivation of thresholds T that we use whenever adaptive inference is involved. First, the confidence scores of each classifier on every sample is computed on line 8, we then create an exit distribution Q by use of the following formula:

$$q_i = e^{log(p/b)*i} \tag{3.10}$$

where p and b are any positive integer. If $p < b$ and given any $v < w$, $e^{log(p/b)*v} > e^{log(p/b)*w}$, conversely if $p > b$, $e^{log(p/b)*v} < e^{log(p/b)*w}$. Lines $9-13$ thus create a monotonic exit distribution for the classifiers

that first decreases, making more samples exit at earlier classifiers, and later increases as p grows, making more samples exit at later classifiers. b thus acts as a turning point and at $p = b$, every exit probability q_i is equal. If *iter* is chosen to be very large, p will eventually make it so nearly every sample will exit at the final classifier only, effectively asymptotically approximating a single classifier scenario, or an anytime prediction scenario where only the last classifier is used. If on the other hand b is chosen to be very large and p is still small, nearly every sample will exit at the first classifier. From Q we can then for each classifier find the $q_i * |D_{val}|$ samples it had the most confidence in and act as if they exit at that classifier, represented by lines 15 and 18, respectively. We set its confidence threshold to the lowest confidence of that set so that later during the evaluation on the test set D_{test} any sample that crosses that threshold will exit via that classifier, shown on line 16. As we assume that D_{val} and D_{test} are from the same distribution, this chosen threshold will approximate the exit probability q_i. The thresholds T are returned, and we can calculate the budget that is associated with this T by means of Equation 3.9 as we know the Q that was used to come up with the thresholds. We can also get the performance of the network for the obtained thresholds in a budgeted batch classification setting by making the test samples exit according to said thresholds. Algorithm 1 thus gives us a set of budget-performance pairs which will serve as an indication of the performance of an early-exiting network in the context of budgeted batch classification.

3.3. Overthinking

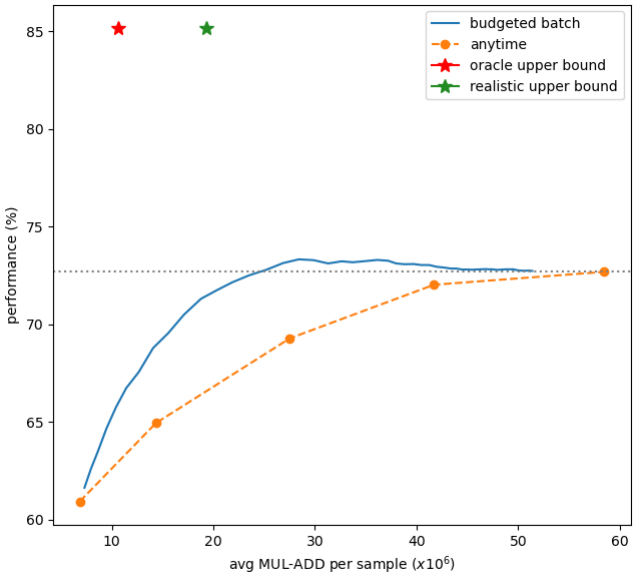


Figure 3.7: Shows anytime prediction and budgeted batch performance of applying a 5-block Multi-Scale Dense network on the CIFAR100 test set. The upper bounds represent results obtained when applying a perfect early-exit strategy. The oracle is clairvoyant and does not perform inference if no classifier can correctly classify the sample.

In subsection 3.2.1 we discussed how a setup with multiple classifiers would go about inference in an anytime prediction setting. Subsection 3.2.3 explained the strategy for the budgeted batch setting. We can show how these two metrics relate by applying an MSDNet on CIFAR100 [27], illustrated in figure 3.7. Appendix A shows similar results obtained when applying MSDNets on video data, it also describes the method used to prepare MSDNets for temporal data. The anytime prediction values represent the performance of each individual classifier. Note that higher performance can be obtained if instead of choosing the classifier whose computational requirements lies closest to the given budget, the 'best' classifier is picked based on confidence criteria. It is actually for this reason that the network's budgeted batch performance surpasses the that of the anytime prediction; early-exiting prevents network overthinking. Overthinking takes places when a later classifier incorrectly predicts a sample, while an earlier classifier already managed to come to the right conclusion. So even though

the average performance of each successive classifier is monotonically increasing, this need not be the case for individual samples; more computation can lead to worse results. More formally, we define overthinking as follows:

$$f_i(\mathbf{x};\theta_i) = \mathbf{y} \wedge f_{i+k}(\mathbf{x},\theta_{i+k}) \neq \mathbf{y} \; where \; k \in \{1,\dots,n-i\} \tag{3.11}$$

where $f_i(\mathbf{x};\theta_i)$ is the output of classifier i and n the total number of classifiers in the network. In the budgeted batch setting, the samples exit based on whether a classifier is confident in its current prediction, not based on a classifiers average performance. Because samples exist on which the network overthinks, not only flops can be saved by early exiting, but performance can be gained at the same time. As a consequence, the budgeted batch performance of an MSDNet is greater than a weighted combination of the average performance of individual classifiers. The early-exiting performance even surpasses the average performance of the final classifier, while using significantly fewer flops. Interestingly, because of the way the thresholds are chosen, the budgeted batch performance eventually decreases and approximates the performance of the final classifier. This is due to the chosen exiting distribution Q favoring the later classifiers as p increases, as explained in subsection 3.2.3. Eventually, all samples will exit at the final classifier as p gets closer to $iter$.

The ability to early-exit combats the overthinking phenomenon; in a way, exiting policies allow the network to chose the classifier best suited for the current sample, instead of relying on performance averages. Taking this to the extreme, figure 3.7 also shows the possible upper bounds in terms of performance and efficiency when perfect early-exiting is applied. The oracle represents a method that will always find the most efficient classifier that can correctly classify the sample and will not perform inference at all when no classifier exists which can do so. The realistic upper bound represents the same concept, except it is slightly less clairvoyant and will send a sample to the last classifier if no classifier can correctly predict it. It shows that the collective performance of all classifiers far surpasses the performance of the individual classifiers in the network and much is to be gained in terms of both performance and efficiency if early-exiting accuracy improves. Much work has already been done to increase the early-exiting accuracy of dynamic neural networks, some of which where briefly discussed in subsection 3.2.3. However, to our knowledge, there is only one body of work that explicitly addresses the concept of overthinking in an adaptive inference setting by Kaya et al [25]. Gaining a deeper understanding of why overthinking takes place could give us the ability to exploit it, leading to an increase in early-exiting accuracy and subsequently increasing the overall performance of the network while simultaneously reducing its computational footprint.

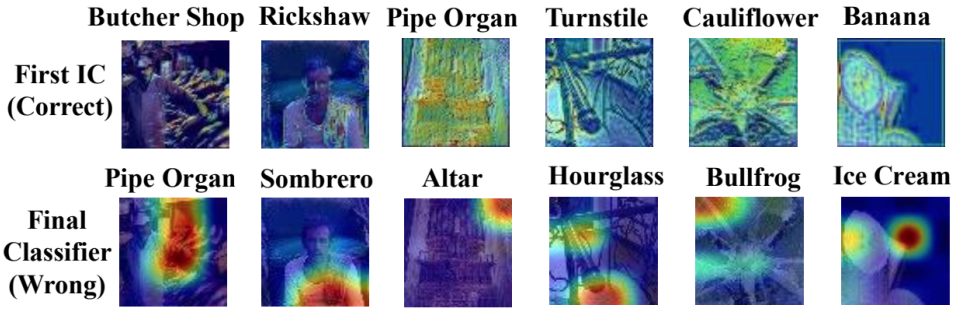


Figure 3.8: Shows how in a traditional CNN with multiple classifiers, the final classifier will find complex, specific features in images that do not contain them. Earlier classifiers, which make use of simpler features are able to classify the sample correctly [25].

Kaya et al. study the overthinking phenomenon in the context of traditional CNNs injected with intermediate classifiers. They note that the final classifier at times incorrectly classifies a sample while an earlier classifier is able to return the correct result. As this is done with traditional CNNs, specifically VGG, ResNet, Wide-ResNet [63] and MobileNet [18], they suggest the reason overthinking takes place is due to the feature maps used by the latest classifier being too specific and detail oriented. Figure 3.8 shows heatmaps of what the respective classifiers take into account to come to a conclusion. The simple features earlier classifiers make use of are present in the whole image, whereas the more complex features which are a result of increased receptive field are sometimes mistakenly found in subregions of images belonging to a different class. The final classifier assigns too much weight to these features

which is visible in the heatmaps. In other words, final classifiers occasionally have tunnel vision which leads them astray. Kaya et al. thus suggest there is a correlation between the spatial receptive field and overthinking. Producing similar heatmaps using GradCam [49] for MSDNets however does not lead to the same results as can be seen in figure 3.9. There seems to be no clear visual pattern that describes what each classifier focuses on. This result is not unexpected as we know from subsection 3.2.2 that MSDNets maintain complex features with full receptive field at every stage of the network. Overthinking can thus not be explained from the perspective of the receptive field when it comes to MSDNets.

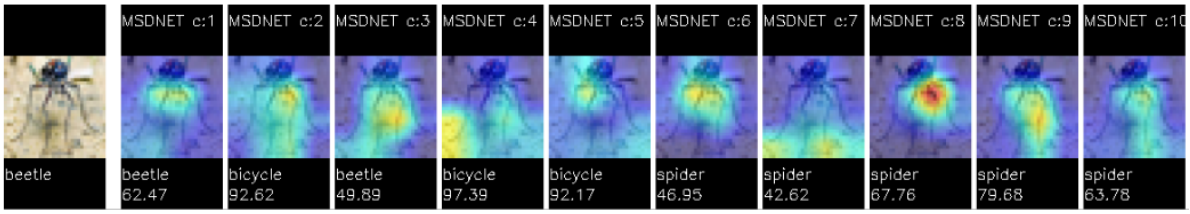


Figure 3.9: Shows an example of what classifiers focus on in a 10-block Multi-Scale Dense network. Each image also shows the respective classifiers' prediction and its softmax confidence in that prediction.

That being said, it does not rule out a correlation between the data and overthinking. There might very well be image statistics that can be used to predict whether a sample will exit at a given classifier or not. If it exists, it could provide us with insight on what each classifier focuses on and by extent what overthinking is. Most of chapter 4 is dedicated to exploring this avenue of trying to explain overthinking from the perspective of the data.

3.4. Reducing overthinking

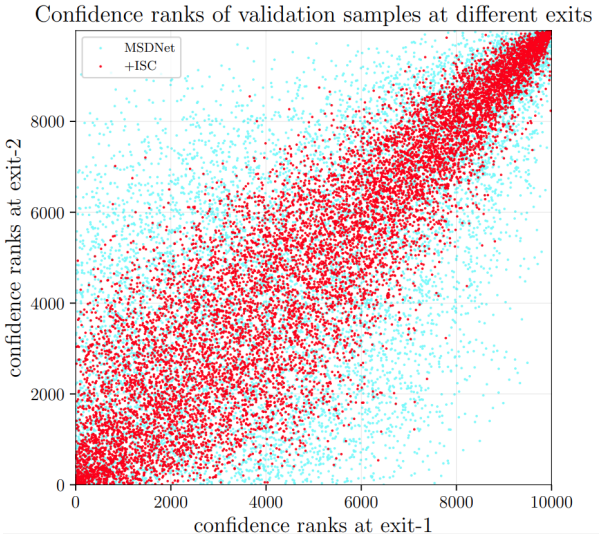


Figure 3.10: Shows how classifier 1 and 2 of a Multi-Scale Dense network rank each sample of the CIFAR100 test set according to confidence. When ISC is applied, these rankings become less dissimilar [34].

Part of overthinking is simply that neighbouring classifiers disagree on what the label of the input sample ought to be. The more they disagree, the more likely it becomes that overthinking will occur. A way to illustrate this disagreement is by obtaining the confidence two neighbouring classifiers have on a set of samples, sort the confidences, and plot them against each other. The result is shown in figure 3.10 as the blue dots in the case of classifier one and classifier two of an MSDNet on CIFAR100 [27]. The original authors of MSDNet propose several methods to increase the performance of classifiers in an MSDNet that can be applied during the training process [34]. Two methods that are particularly of note are Inline Subnetwork Collaboration (ISC) and One-for-all Knowledge Distillation (OFA). Both aim to

improve collaboration between classifiers during the training process to incentive knowledge transfer. The methods can be described as follows:

- **Inline Subnetwork Collaboration.** ISC adds connections between every classifier i and classifier $i+1$. These connections can consist of several convolutional layers or simply the identity function. In the latter case the results of classifier i are transferred to the next classifier without further alteration. The idea is that classifier $i+1$ can make use of what classifier i thinks, instead of just sharing the same parameters in the backbone of the network. This more direct connection stimulates analogous thinking between adjacent classifiers.

- **One-for-all Knowledge Distillation.** OFA allows intermediate classifiers to directly learn from the final classifier of the network. In general, we know that the final classifier performs best on average due to its increased capacity. This knowledge transfer is thus a form of teacher parent setup where the final classifier is the teacher who is trying to instill its knowledge into all other classifiers in the network. The loss for each intermediate classifier i becomes:

$$L_i(\mathbf{y}, f_i(\mathbf{x};\theta_i)) = \alpha\mathcal{L}_{CE}(\mathbf{y},\hat{\mathbf{y}}) + (1-\alpha)KLD_i \tag{3.12}$$

  where α determines how much OFA contributes to the total loss and KLD_i refers to the Kullback Leibler divergence between classifier i and the final classifier k:

$$KLD_i = -\sum_{c\in\mathbf{y}} p_k(c|\mathbf{x},\theta_k) log(\frac{p_i(c|\mathbf{x},\theta_i)}{p_k(c|\mathbf{x},\theta_k)}) \tag{3.13}$$

  with c is every class in $\mathbf{y}$ and $p_i(c|\mathbf{x},\theta_i)$ the probability classifier i assigns to sample $\mathbf{x}$ belonging to class c. In essence, KLD describes how much two probability distributions differ, the probability distributions in this case being the predictions of an intermediate classifier and the final classifier on some sample. OFA stimulates intermediate classifiers to think more along the same line as the final classifier of the network.

The effect of ISC applied to an MSDNet can be seen as the red dots in figure 3.10. The confidences of the adjacent classifiers line up significantly more so than before. On top of that, the authors show that this increases performance of individual classifiers by several percentage points in controlled settings, suggesting that increased collaboration benefits classifiers in multi-exit networks.

	Training methodology		
Dataset	**50 epochs**	**100 epochs**	**50 epochs and 50 epochs ISC + OFA**
CIFAR100	17.93	18.45	**14.33**
SHVN	15.32	15.50	**11.29**

Table 3.1: Shows the percentage of samples the network overthinks on for the test set of the respective dataset. Lower is better.

Considering the results of ISC, it begs the question whether more aligned confidences also results in less overthinking. To this end, we compare training an MSDNet regularly versus training it using ISC and OFA. When using the additional training techniques, we mostly follow the implementation of the original authors by first training a network as normal, and only fine-tuning it using ISC and OFA. The results are shown in table 3.1. It shows that overthinking can be reduced by introducing more explicit collaboration between classifiers. In a way, ISC and OFA help classifiers fill their knowledge gaps which they incur during training. It does however not explain why overthinking occurs and where this independence comes from in the first place. In the following chapter we attempt to find an explanation for overthinking itself.

4

The cause of overthinking

With only few exceptions, the average performance of classifiers in dynamic neural networks that utilize early-exiting increases monotonically in relation to their depth. This comes as no surprise as each consecutive classifier has a higher capacity. However, considering individual samples, a later classifier might misclassify a sample that an earlier classifier can classify correctly. We might expect that if any classifier arrives at the correct prediction, any more computation would not lead to adverse effects. However, this indeed tends to be the case for a significant portion of the test samples, about 18% for CIFAR-100 for example. In general, the set of samples S_j a classifier at depth j gets correct is larger than the set of samples S_i a classifier at depth i gets correct, where $i < j$, yet $S_i \not\subset S_j$. As a consequence, the collective performance of all classifiers of the network greatly surpasses that of any individual classifier of the network. Despite the inherent built-in dependencies that come with the MSDNet architecture, these results suggest a certain level of independence between classifiers. In this chapter we explore a possible explanation for this independence, namely that classifiers might specialize in certain inputs. We do so by attempting to train a network to predict where a sample should exit in a pre-trained MSDNet. If a separate network can do so successfully, it suggests there is a correlation between the input data and what each classifier in an MSDNet has learned.

In this chapter we first explain policy networks in section 4.1, a concept that motivated the consequent experiments. We then cover the overarching experiment setup in the introduction of section 4.2, which we revise in subsection 4.2.1 in light of the initial results. In subsection 4.2.2 we show that the experiment setup can successfully train a policy network to learn an early-exit strategy in MSDNets when the dataset provides sufficient room for classifier specialization. In subsection 4.2.3 we show that this does not hold in general. Finally, in section 4.3 we offer up a different theory for the occurrence of overthinking considering the results of the preceding subsection.

4.1. Policy networks

Even though classifiers in an MSDNet share the majority of their parameters, overthinking suggests that some level of independence among classifiers still exists. This independence is actually the reason why MSDNets, and also other early-exiting networks, tend to perform better in an adaptive inference setting than in a static inference setting; in the former setting the network is to some extent able to pick the right classifier for any given input sample. This decision is often based off of some type of confidence measure which in turn is correlated with performance. The fact that some of the earlier classifiers are able to correctly classify samples that later classifiers can not despite their increased capacity, suggests that these earlier classifiers might just be better at these type of inputs. This idea naturally gives rise to a more general version of this line of inquiry: Do classifiers in a MSDNet specialize in certain subtypes, classes or image statistics of a dataset to maximize performance? In other words, can we find a correlation between the input data and a trained MSDNet? If such a correlation can be found, it can provide insight into why overthinking takes place. This knowledge can subsequently be used to exploit the independence between classifiers even further. Figure 3.7 showed what could potentially be achieved if early-exiting performance of existing MSDNets were to improve.

To determine if a correlation exists between the input data and what an MSDNet learns, we will

attempt to learn it directly with an independent network. A separate network that decides what dynamic strategy should be used for the main inference network is also referred to as a policy network. Policy networks have been used to learn which layers to activate in ResNets based on a given input: [8] by Chen et al. and [59] by Wu et al. Note that policy networks themselves are convolutional neural networks. Their policy network is trained using associative reinforcement learning where the reward is based on minimizing active layers and maximizing performance of the main network. The policy network thus learns to conditionally activate parts of the network that it thinks have the best chance of providing an accurate result. Interestingly, Wu et al. conclude that their policy network will activate similar layers for samples belonging to the same class. This means that it is able find a correlation between the input data and parts of the inference network which might be biased toward specific classes. It can also be described as a form of specialization. Furthermore, they note that later layers are more important than early ones, likely due to later layers representing more complex and class specific features.

It seems that much of the success around policy networks is due to them being able to correlate the input data to parts of the main network that represent similar features. ResNet, which both papers used as their main network, is one-dimensional in its architecture. Feature maps become increasingly more complex along the depth of the network. It is likely due to this fact that policy networks will learn to ignore shallow layers and focus on deeper layers. Feature maps of an MSDNet do not grow in complexity along the depth of the network, or at least not in the same way as in traditional networks. In the same vain, the concept of a linear receptive field does not apply to MSDNets. It is for these reasons that is hard to say if a policy network will have the same success in the context of MSDNets as they do for traditional CNNs.

4.2. Experiment setup

We will take a different approach than Chen et al. and Wu et al. when it comes to training our policy network. As we are generally not interested in saving computation, a reinforcement approach is redundant. Instead, we aim to create a policy network that is able to predict which classifier of the MSDNet is the first to accurately classify the input sample. Such a policy network, if it were perfect, would be similar to the upper bounds depicted in figure 3.7.

Each experiment in the upcoming subsections, while similar, are slightly different in the way they are executed as they take into account the lessons learned of previous experiments. However, it is prudent to get across certain terminology that is relevant to each experiment as they each follow the same steps. These steps use terminology that is overloaded, for this reason we will denote the terms with their own syntax based on the context. Every experiment will perform the following 4 steps in order:

1. Choose a dataset D that the MSDNet will train on.
2. Train an MSDNet M on dataset D.
3. Create a dataset from M containing pairs of samples (d, l) with $d \in D$ and l representing the l^{th} classifier in M that is the first to correctly classify sample d. If no classifier is able to classify d correctly, we do not include d in the new dataset. We refer to this new early-exiting dataset as D_{ee}.
4. Train a traditional CNN M_{ee} on D_{ee} to learn the early-exit strategy of M. M_{ee} is thus the policy network.

The goal of this process is to determine whether M_{ee} can be successfully trained. We consider this to be the case if the performance of M_{ee} is higher than that of random guessing, or higher than if it learns to always return the same answer. The reason for the latter is that we will see that D_{ee} is not always balanced in terms of class labels, hence a network that is unable to learn anything meaningful will often end up returning the class that occurs most often in the training set to minimize the loss. If M_{ee} performs well, it suggests there is a correlation between input data statistics and specific classifiers in M that are able to learn, and specialize in, said statistics. It should be noted that the decision rule in step 2 for determining the labels of samples in dataset D_{ee} is somewhat arbitrary. As long as the label is that of a classifier that correctly classifies the sample, M_{ee} will end up trying to learn the existence of a connection between the input d and what classifier l in M has learned. We will see that our strategy for determining the label l will vary over the course of the experiments to suit our purpose.

For step 1 we choose several datasets each with different motivations, as will be described in their respective subsections. The architecture for MSDNet M in step 2 is the same for every experiment. It consists of 5 blocks, each block is followed by a classifier, meaning the network has 4 intermediate classifiers and 1 final classifier. The number of layers in each classifier grows linearly, the first block only contains the initial unique downsampling layer and every subsequent block i contains i layers. We will maintain 3 scales across the depth of the network to represent feature maps of multiple levels of coarseness which output 6, 12 and 24 feature maps, respectively. Each layer in a block makes use of 3×3 convolutions, batch normalization, max pooling and ReLU activation functions. The classifiers themselves use strided convolutions to downsample the input feature maps and a single fully-connected layer as its output. The classifiers are optimized jointly according to Equation 3.6. For more information on implementation specifics, we refer the reader to the Appendix of the MSDNet paper [20]. Each M is trained for 200 epochs, starting off with a learning rate of 0.1 which decreases by a factor of 10 at epochs 75 and 150. For M_{ee} we choose to use a version of ResNet [17] containing 34 layers. There are no concrete requirements for M_{ee}, it simply needs to have sufficient capacity while remaining manageable training-wise. Each ResNet was trained for a maximum of 100 epochs, at times stopped early if no progress was being made.

4.2.1. Initial experiments

Initially, we apply the strategy described in the introduction of **??** verbatim: We choose CIFAR-100 [27] as D, train our M on it, create D_{ee} by selecting the first classifier of M that is able to classify the sample correctly as the label for the new samples and lastly train M_{ee} on D_{ee}. Without belaboring the point, using this strategy, M_{ee} does not end up learning anything meaningful. However, by doing so, we uncover a variety of shortcomings in the basic strategy used which leave doubt in the veracity of the outcome. To eliminate all doubt we make a number of changes to the experiment setup that we will stick to for experiments to come.

The main issue that arises when the steps are applied without any further thought is that D_{ee} ends up egregiously imbalanced. The reason for this is the high accuracy of every classifier in M on D. In general, the training accuracy of a network is far higher than its validation accuracy on either the validation set or test set. This is because the latter 2 sets are left out of the training process. The improved accuracy on the training set is thus due to overfitting. If we base the training set of D_{ee} on that of the training set of D, selecting for the labels the first classifier in M that correctly classifies samples in D will lead to a significant bias towards the first physical classifier in M. If classifier 1 of M has a training accuracy of 80%, 80% of samples in the training set of D_{ee} will be labeled as 1 as a result. The imbalance will motivate M_{ee} to just return label 1 if it is unable to learn any correlations that improve its performance beyond 80%.

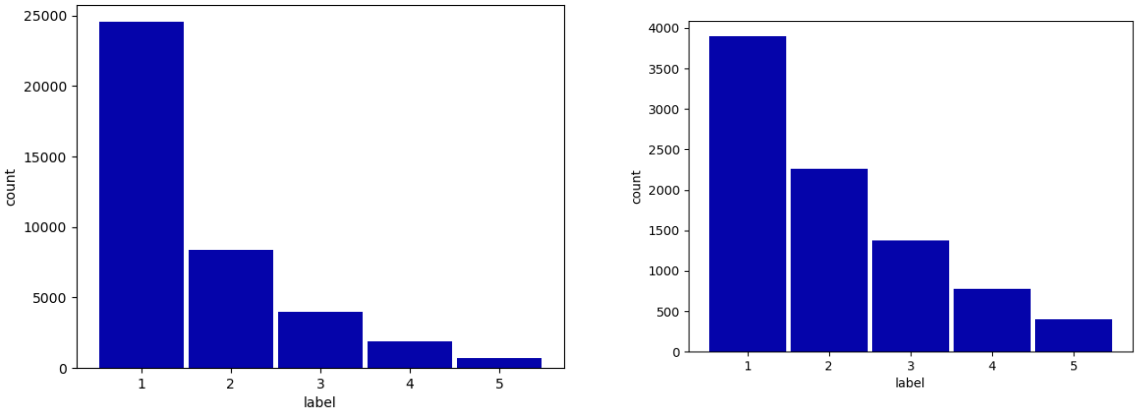


Figure 4.1: (**Left**) Shows the class distribution of D_{ee} on CIFAR-100 when using the training set of D. (**Right**) Shows the class distribution of D_{ee} on CIFAR-100 when using the validation set of D.

To start off, a straightforward way of tempering the imbalance is by using the validation set of D to create the training set of D_{ee}. The average performance of classifier 1 on the validation set is noticeably

lower than its performance on the training set, leading to a reduction in samples labeled 1 in D_{ee} and consequently an increase in remaining labels. To illustrate the point, in the case of CIFAR-100, the first classifier of M reaches a performance of 75.05% on the training set and 60.93% on the validation set. Figure 4.1 shows what effect this has on the distribution of D_{ee}. Basing the training set of D_{ee} off of the validation set of D does however come with a non-trivial downside; the validation set of D contains only a fraction of the samples of the training set. Conventionally, roughly $10 - 20\%$ of the samples in the training set are set aside to be used as validation. We opt for 20%, leaving enough training samples for the classifiers in M to learn image statistics, if it exists, while at the same time granting M_{ee} plenty of samples to prove it can find this correlation between classifiers and image statistics, again, if it exists. The test set of D is used to create the test set of D_{ee}, we omit the creation of a validation set in D_{ee} to preserve as many samples for training M_{ee}.

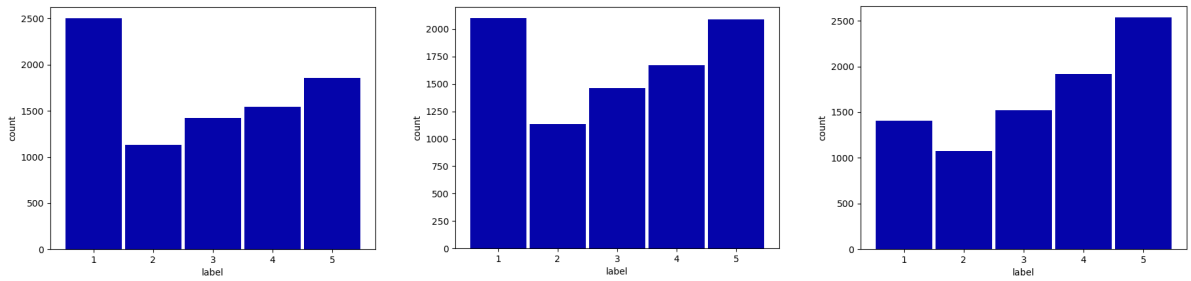


Figure 4.2: Shows the class distribution of D_{ee} on CIFAR-100 when using the validation set of D and a score based system with $\lambda = 5, 10, 50$ from left to right.

Secondly, we substitute step 2 of the process with a more nuanced approach. Instead of selecting the first classifier that classifies sample d correctly as the label, we assign each classifier a score according to the following formula:

$$score_i = \frac{1}{\lambda flops_i(\mathbf{x})10^{-6}} + conf_i \tag{4.1}$$

The score is a weighted sum of the inverse of the flops used and the confidence of classifier i. When given a sample input $\mathbf{x}$, a classifier's score is higher if it uses fewer flops and has a higher confidence in its prediction. We use the maximum of the softmax score to determine confidence, as shown in Equation 3.7. The label l in step 2 of sample d now becomes the classifier with the highest score:

$$score'_i = \begin{cases} score_i, & \text{if } f_i(\mathbf{x}, \theta_i) = \mathbf{y} \\ 0, & \text{otherwise} \end{cases}$$
$$l = argmax_{score'_i}(\dots, score'_i, \dots) \text{ for } i \in 1, \dots, n \tag{4.2}$$

where a classifier only gets assigned a non-zero score if it is able to classify the sample correctly in the first place. By altering λ we gain control over the label distribution of D_{ee}. If λ is smaller, the efficiency of the classifier is favoured and the average label value in D_{ee} becomes lower. Conversely, if λ becomes larger, confidence becomes a more influential factor favouring later classifiers, raising the average label value in D_{ee}. Figure 4.2 illustrates this effect. While it may seem as if we are changing the learning goal for M_{ee}, remember that the labeling of D_{ee} is arbitrary to begin with. As long as label l represents a classifier that correctly classifies sample d, $(d, l) \in D_{ee}$ will represent a dataset that contains information on the relation between samples and what classifiers in M have learned. Also, taking into account the confidence of a classifier when deciding on a label promotes the labeling of a classifier that in a sense is most suitable for that sample. If 2 classifiers correctly classify a sample, we might expect that the one with higher confidence has a stronger connection to that sample. In other words, that sample might be a better representation of what the classifier with higher confidence ends up learning, making it a more valuable asset than if the label were assigned to the other classifier.

Finally, even when using the validation set and a scoring method to create D_{ee}, it often still ends up imperfectly balanced. For this reason we will always make use of weighted learning, where a class

dependent weight is used after the negative log likelihood is applied to the softmax output of the classifier, which is just another way of referring to the cross-entropy loss in the case of multiple classes:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -w_y log(\frac{e^{\mathbf{z}_y}}{\sum_i^n e^{\mathbf{z}_i}}) \tag{4.3}$$

where w_y is the weight specific to class y and $\mathbf{z}_y$ the output of the classifier for class y. The weights are determined for the training set before training starts. While the weights can take any value, we will simply make them proportional to the occurrence of each class: If we consider an ordered array containing the occurrences of each class in the training set and denote it with C_{train}, the weights become $w_y = \frac{max(C_{train})}{C_{train}[y]}$ for $y \in \{1, ..., n\}$. In our case $n = 5$ as M contains that many classifiers. Effectively, this discourages M_{ee} from simply returning the most common class by penalizing incorrect classifications of minority samples more harshly.

After applying the new and improved concepts, we will end up with more balanced datasets and more confidence in the outcome of the experiments as a result. It should be noted that these additions and alterations to the initial experiment setup did not end up changing the outcome of the experiment on CIFAR-100; M_{ee} still does not end up learning much of anything. However, the outcome now serves as a more rigid result than the initial outcome. We will use the improved strategy in the subsequent experiments as well. To prove that the improved setup does work, i.e. if a known separation in D exists by which classifiers will differentiate then M_{ee} will be able to pick up on it, we will create a specific dataset called Max-Min MNIST.

4.2.2. Max-Min MNIST

In the previous subsection we showed how even using an improved experimental setup, M_{ee} will not be able to distinguish anything of note within D_{ee} on CIFAR-100. This can however have any number of reasons beyond the absence of an existing correlation between D and the classifiers of M. In other words, it is hard to prove the absence of classifier specialization by means of a single negative experimental result. It might be the case that our setup is simply not capable of uncovering classifier specialization in general. In this subsection we will show that using our experimental setup we can successfully train a policy network M_{ee} on D_{ee} if the data in D is sufficiently different, which results in classifiers in M learning distinguishable subsets of D. To do so we create a dataset called Max-Min MNIST (MMM) based on the dataset used in [9].

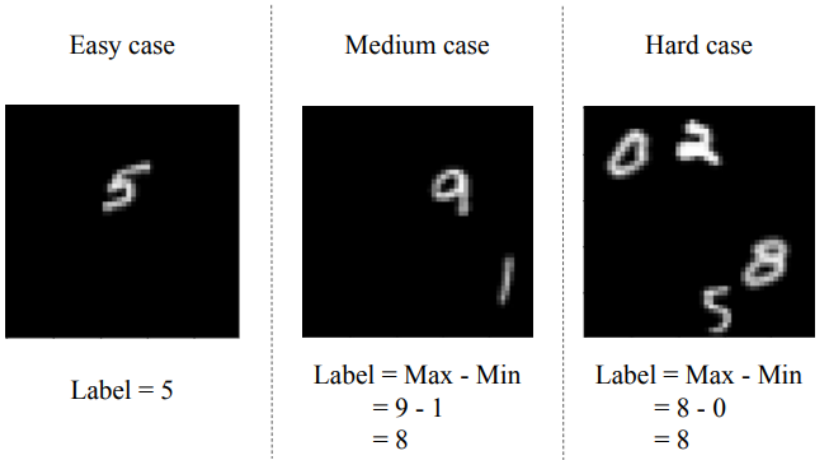


Figure 4.3: Shows examples of samples from the Max-Min MNIST dataset. The labels for the medium and hard cases are determined by subtracting the smallest digit from the largest digit in the image [9].

The core idea behind MMM is that it contains multiple levels of difficulty for each label. In this case we make use of 3 levels: easy, medium and hard. Every sample consists of a black plane of 64×64 containing 1 or more 20×20 digits sourced from the original MNIST [33] dataset. Figure 4.3 shows examples of each level. The easy case consists of single digits which are both sourced and placed uniformly randomly in the plane. The medium case requires the network to recognize both digits and

output the largest digit minus the smallest digit. The hard case asks the same of the network but now with 4 digits. The levels exist in a 2:1:1 ratio for each label in both the training and test sets. Furthermore, both sets are completely balanced in terms of class labels. There are a total of 100,000 training samples encompassing 10 classes, so there are 5000 easy samples and 2500 medium and hard samples for each class. The training set consists of 18000 samples.

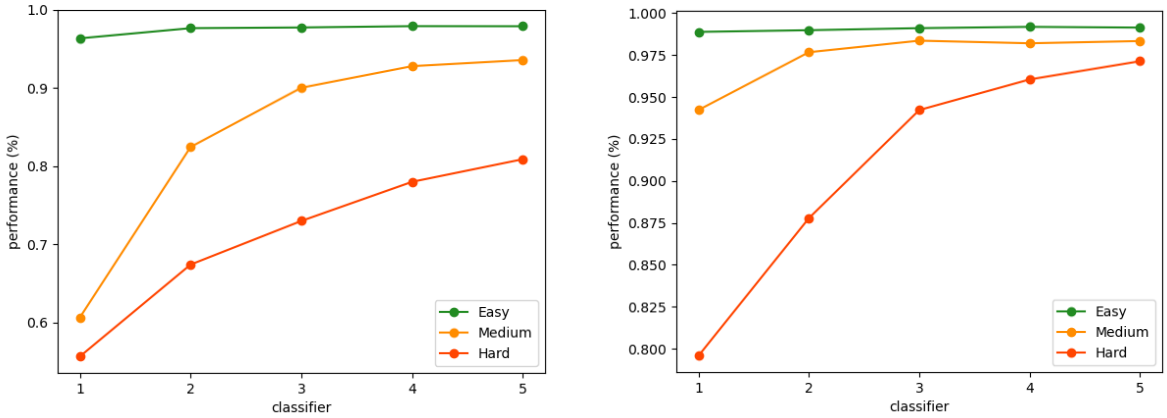


Figure 4.4: Shows the performance of each classifier in MSDNet M on dataset Max-Min MNIST D for each of the difficulty levels when M is trained for 5 and 200 epochs, respectively.

By using a dataset with baked in difficulty differentiation, we can enforce a specific type of specialization by exploiting the growth in capacity of each consecutive classifier in M. Later classifiers are better at classifying the harder samples than earlier classifiers. More importantly, the margin by which later classifiers are better at harder samples is larger than in the case of easier samples. We can see this effect in Figure 4.4. Both the earlier classifiers and later classifiers have almost perfect accuracy when it comes to classifying easy samples, but there is a large difference in accuracy for the hard samples. This effect becomes more pronounced when M is only trained for a few epochs. For this reason we will make use of the latter M to enforce further differentiation.

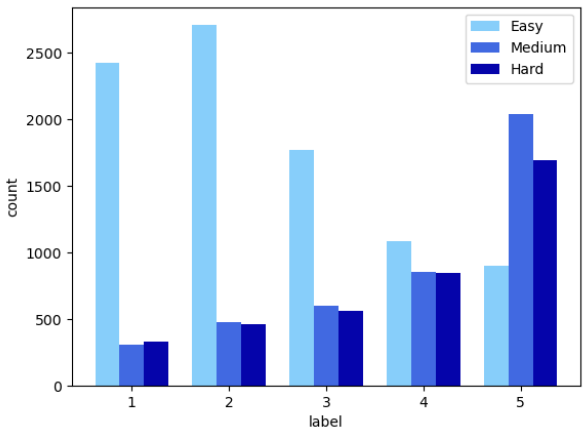


Figure 4.5: Shows the sample distribution of the training set of D_{ee} obtained from applying the score method with $\lambda = 5$ on M. M in this case is trained for only 5 epochs on MMM.

Now that we have trained our MSDNet M on the MMM dataset D, we can apply the concepts of the previous subsection to create D_{ee}. The label distribution of D_{ee}, as seen in Figure 4.5 now clearly shows the baked in sample differentiation; because later classifiers are significantly better at medium and hard samples than earlier classifiers, the score method will noticeably favour those classifiers for

labeling. Conversely, as the performance of early and late classifiers on easy samples is similar, the score method will favour earlier classifiers which are more efficient. Inherently this D_{ee} is not different from the D_{ee}'s we saw in the previous subsection, except for the knowledge that we now have about the existing 'specialization' within M, namely that earlier classifiers 'specialize' in easy samples, and later classifiers 'specialize' in medium and hard samples. We put specialize in quotation marks as its a rather artificial form of specialization and not necessarily the type we expect to see often in a more realistic scenario. Nevertheless, it is a form of specialization and we can train our M_{ee} on it.

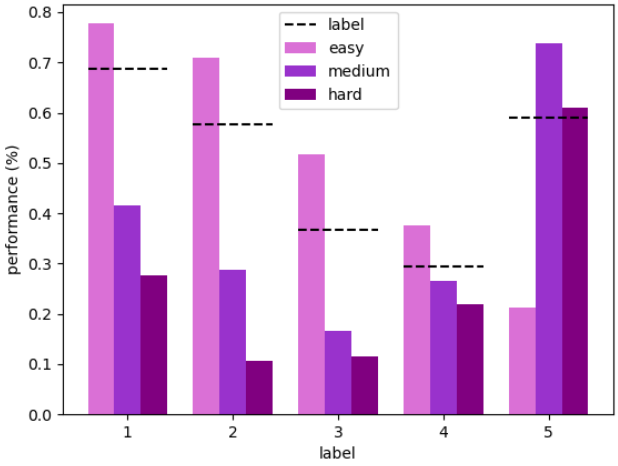


Figure 4.6: Shows the performance of M_{ee} on the test set of D_{ee} for each label and respective levels.

The training set of D_{ee} contains roughly 15,000 samples, and the test set likewise. We train M_{ee} for 60 epochs as it makes no progress after that. The overall performance over the whole test set of M_{ee} is 51.83%. The breakdown of how it performs for each label and difficulty level is shown in Figure 4.6. Several trends stand out, first of all, M_{ee} is clearly better at classifying samples at the tails of the label distribution. Also, it is poor at recognizing exceptions such as early samples that should end up at the final classifier, and medium and hard samples that should end up at intermediate classifiers. In general, it seems to not be able to conceptualize the nuances of classifiers that lie somewhere in the middle of the network. Naturally, as M contains more classifiers, the performance of M_{ee} will go down if we strictly look at the overall performance of it on the test set. Taking this into account and considering that random guessing would have lead to a performance of 20%, and the most common label, being label 5, appears 27% of the time, a performance of 51.83% shows M_{ee} is clearly able to learn a correlation between the data and classifiers of M.

What image statistic M_{ee} uses to determine what classifier in M the data belongs to is hard to say. We introduced artificial complexity to D according to what we consider to be more difficult to classify. Figure 4.4 shows that networks in this case agree with our intuition; hard samples are more difficult to classify than easy samples. What networks consider difficult and what sample complexity actually entails is a study in and of itself. One school of thought suggests one image statistic that describes complexity is how 'cluttered' an image is [45]. This has, among other things, been studied in the context of the learning order of networks [57, 43]. One measure of the 'clutterness' of an image is how many of the higher frequencies are needed to reconstruct the image within a given error bound when the image is decomposed into its frequency components. An image can be transformed into the frequency domain by using a Fourier transformation, but also by applying a Discrete Cosine Transformation (DCT) [41]. We explore the latter concept on several of the used datasets to see if the clutterness of an image serves as an explanation not only for the discrepancy in difficulty in the MMM dataset, but also to see if it can be used to explain specialization in MSDNets in general. An explanation of the method, as well as the results can be found in Appendix A.

4.2.3. CIFAR100 and SHVN

With the results of M_{ee} on MMM we have effectively shown that our 4 step method of constructing a policy network is capable of uncovering correlations between what classifiers in an MSDNet learn and the data they were trained on. In this subsection we will show that this does not hold for conventional datasets. Figure 4.6 illustrated how M_{ee} is better at differentiating samples that should go the first and last classifier. "When in doubt, send easy samples to the first classifier and hard samples to the last" is a sensible motto a policy network might live by. For this reason we simplify the original classification problem by only taking into account the first and last classifier of M. We train M as normal, but this time when we give each classifier a score according to Equation 4.2 we only do so for classifier 1 and 5. The other scores remain zero. This effectively turns the task of the policy network into one of binary classification.

We revisit the CIFAR100 dataset, one of the most commonly used classification sets in computer vision. It contains 100 classes consisting of 32×32 images of commonly found food, vehicles and animals. We also perform the exact same experiment on the SHVN dataset [42]. SHVN, another popular image dataset, consists of 32×32 images containing house numbers with labels 0 through 9. We choose CIFAR100 because it is as popular as it is and it contains a decent amount of training samples, 50,000 to be precise of which we turn 20% into validation samples for M. Having enough training samples to turn into validation samples is important as we tend to lose sample volume when creating D_{ee}; remember that we only include $d \in D$ into D_{ee} if at least 1 classifier in M classifies it correctly. As a result, small datasets on which M performs poorly to begin with will lead to questionably small D_{ee}'s. SHVN contains even more samples, ∼75,000 training samples and ∼500,000 extra samples that can be used as necessary. We supplement the validation and test set of SHVN with 25,000 samples each from the extra set. The reason we also supplement the test set is because the extra set is considered easier to classify. Remember that the validation set of SHVN will become the training set of D_{ee}, if this only contained easier extra samples it might hinder the ability of M_{ee} to do well on the test set which would then only contain the harder samples.

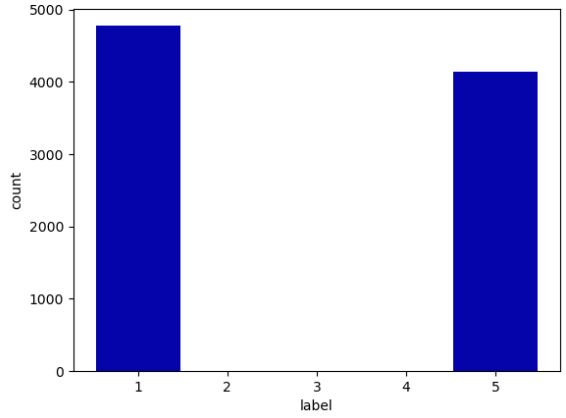


Figure 4.7: Shows the sample distribution of the training set of D_{ee} obtained by applying the score method with $\lambda = 5$ on M. M in this is case is trained on CIFAR100.

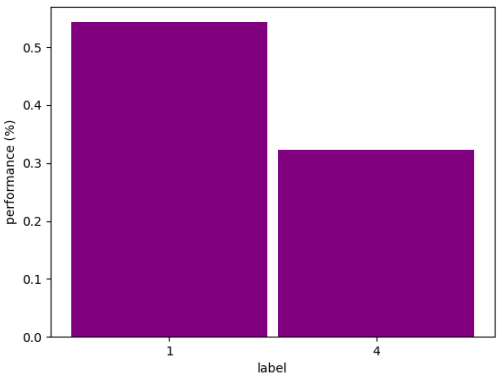


Figure 4.8: Shows the performance of M_{ee} on the test set of D_{ee} for both labels. D_{ee} is based on CIFAR100.

CIFAR100. Figure 4.7 shows the binary label distribution of applying the score method of Equation 4.2 on M. M in this case is an MSDNet trained for 200 epochs on the CIFAR100 training set. We use $\lambda = 5$ to maintain a relatively balanced dataset. To the right of it, in Figure 4.8 we see the average performance M_{ee} achieves on D_{ee}. The average performance is 43.25%, well below the cutoff of $\sim 50\%$ we consider to be the threshold for having learned anything.

SHVN. Figure 4.9 shows the binary label distribution of the training set of D_{ee}, it was created using $\lambda = 20$. M is trained for 200 epochs on the SHVN training set. In Figure 4.10 we see the average performance M_{ee} achieves on D_{ee}. The average performance is 49.92%; M_{ee} ends up returning labels evenly, and thus randomly. Again, there is no indication of M_{ee} having learned anything.

Even in a binary classification setting, M_{ee} is seemingly unable to discover a connection between

the data and the classifier in an MSDNet that is most suitable to classify it. M_{ee} either ends up along a path where it performs worse than random guessing, as we saw in the case of CIFAR100, or ends up randomly guessing in the case of SHVN. Either way, it becomes hard to argue that classifiers specialize in any specific subsets of the dataset when the dataset is not tailor made to introduce specializations as was the case with MMM. In the next section we offer up a different explanation for overthinking in light of these results.

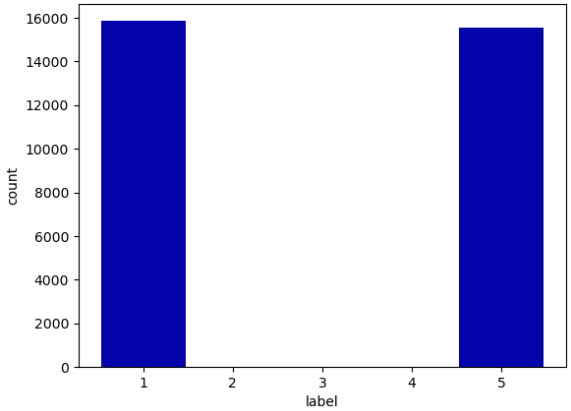


Figure 4.9: Shows the sample distribution of the training set of D_{ee} obtained by applying the score method with $\lambda = 20$ on M. M in this is case is trained on SHVN.

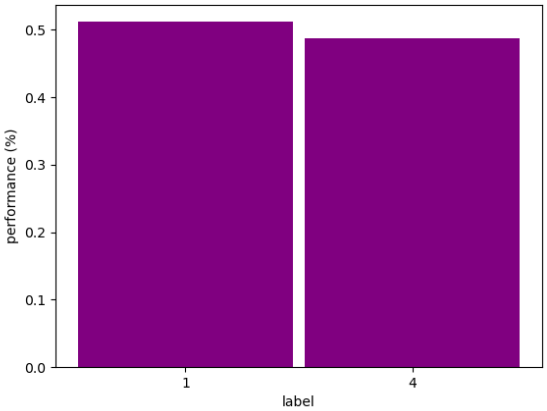


Figure 4.10: Shows the performance of M_{ee} on the test set of D_{ee} for both labels. D_{ee} is based on SHVN.

4.3. Alternative explanation for overthinking

In this chapter we have attempted to directly learn early-exiting strategies in MSDNets by means of a policy network. The idea is that if a policy network is capable of doing so, there must be a relation between the data and what classifiers in an MSDNet have learned. Policy networks would only be able to recognize these correlations if they are strong enough, in other words, if classifiers in an MSDNet tend to specialize in specific subsets of the dataset. This in turn would explain the overthinking phenomenon: Classifiers gravitate toward particular types of samples to maximize the overall performance of the network. The results show that policy networks struggle to ascertain the mentioned relations in conventional datasets. This suggests one of two things: a relation between the data and classifiers in an MSDNet exists but can not be picked up by policy networks or the relation does not exist to begin with. The latter in turn suggests that overthinking does not take place due to specialization. We have seen that policy networks are able to pick up on data and classifier relations when they are present in the case of a dataset like MMM, as was shown in **??**. We thus consider the likelihood of the former reason for the lack of success of policy networks in the context of MSDNets to be relatively low.

As we know there are samples that only a subset of the classifiers in an MSDNet can classify correctly, and as this does not depend on the data itself, it becomes more likely that it is the cause of stochasticity within the learning process itself. During training we optimize the joint loss of all classifiers at the same time. The joint loss does not take into account the collective performance of the network, it instead tries to minimize the average loss of each of the classifiers. This in turn results in a lack of collaboration between classifiers as we saw in section 3.4. Note that all of this is valid whether classifiers specialize or not, classifiers could specialize by tunnel-visioning on specific samples, which could be caused by a lack of collaboration. However, in the face of the results from this chapter, it seems more likely that the difference in what classifiers learn is due to the stochasticity in the learning process itself. If the network is in a particular state during training, the next training sample will cause the parameters of the network to update. As the goal is to minimize the average loss, which loss gets minimized makes no difference. Whether the loss of the final classifier becomes smaller, or the loss of the first classifier is of no concern to the network. It could thus happen that updating the parameters will lead to one of the classifiers not being able to classify the sample correctly, while the others can. So while there

is no explicit collaboration taking place, joint optimization specifically favours the greater good. The classifier that gets left behind in this example could be a classifier later in the network, resulting in overthinking. Depending on the state of the network before the new training sample gets introduced, the outcome of backpropagation can have different results. The learning order, so the order with which samples are fed to the network during training might thus already have a noticeable influence on the way overthinking is introduced in the training process. Hence the stochasticity; the training of a convolutional neural network involves a variety of stochastic processes, from training order and batch normalization to parameter initialization. It would be hard to tell which of these factors contribute and with what magnitude. We leave this research tangent to further research.

5

Discussion

This thesis has attempted to find an explanation for the overthinking phenomenon that takes place in Multi-Scale Dense networks [20]. In this chapter we will summarize the results and simultaneously address their shortcomings and limitations. We will conclude the chapter with potential future work in this regard.

5.1. Conclusions

Dynamic neural networks offer computational flexibility and adaptability as opposed to its static counterparts. This allows them to perform adaptive inference in the form of budgeted batch classification and anytime prediction: two computational settings which have real life equivalents in the form of image indexing and self driving vehicles. One of the more common and arguably the most straightforward way to instill a static network with adaptive capabilities is to add intermediate classifiers to it. Doing so allows a network to choose which exit to use for a given sample. Chances are an intermediate classifier can already correctly classify the sample and no further computation is required. An end-to-end trained network with multiple classifiers also does not suffer from redundant computation, as results from a previous classifier can directly be used for the prediction for the classifier next in line. However, implementing early-exiting naively into a traditional convolutional neural network will lead to poor performance. For this reason, hand-tuned specialized early-exiting architectures have become popular in recent years. Multi-Scale Dense networks is one of those architectures that has solved many of the shortcomings of its early-exiting predecessors.

Any early-exiting setup, MSDNets included, suffer from a counter-intuitive phenomenon: More computation does not guarantee better, or even equivalent, outcomes. While there is a monotonic increase in the average performance of each consecutive classifier in an early-exiting setup, this does not necessarily hold for individual samples. This phenomenon, which has been dubbed overthinking [25], was first noticed in conventional convolutional neural networks such as ResNet [17]. The explanation offered is that classifiers deeper in the network have access to more complex feature maps which the classifier mistakenly finds in images that do not contain them. One of the main reason later classifiers have access to complex feature maps is due to deeper layers having larger receptive fields. Shallow classifiers, positioned earlier in the network will only have access to feature maps with small receptive fields and it is exactly those primitive features that are required to accurately classify some samples. We showed in section 3.3 that this explanation, while possibly correct, does not hold for the case of MSDNets. The architecture of MSDNets and other multi-scale setups does not adhere to the conventional wisdom about receptive fields and the way they grow along the depth of a network. Instead, by aggressive early downsampling, every feature present in the largest scale of a multi-scale setup contains knowledge of the full spatial representation of the input image. Consequently, every classifier, no matter its depth, will have access to feature maps with full receptive field. Yet, overthinking still very much takes place in MSDNets.

In section 3.4 we have shown how overthinking in MSDNets can be reduced by making use of concepts brought forth by the original authors of MSDNet [34]. They showed how improving collaboration between classifiers improves both their individual and collective performance. This increased collabora-

tion in a way allows classifiers to fill each others knowledge gaps. Another consequence is that classifiers start to think more along the same lines, effectively reducing the occurrence of overthinking. However, this does not explain the underlying cause. Understanding why it does take place could provide insight on how to improve early-exiting. Improved early-exiting in turn leads to reduced computation and increased performance of dynamic neural networks.

Several works have shown that it is possible to train a separate policy network to dynamically alter the structure of the inference network to maximize performance or reduce computation. Successful conditional activation of parts of a network suggests that what these parts have learned is dependent on particular image statistics of the dataset. This dependency can thus be learned by a policy network. Their work focused on traditional convolutional neural networks and it is likely the policy network, a convolutional neural network itself, is able to find a correlation between the depth of the inference network and the input samples because what each feature map represents is highly correlated with its depth. It learns for example that certain parts of the network are useful for certain classes. We say that these parts specialize in those classes. Motivated by these works we attempted to determine whether classifiers in an MSDNet also specialize in particular subsets of a dataset.

First, to prove that our setup worked, we trained a policy network to predict what classifier is most suitable for classifying a given sample from the max-min MNIST (MMM) dataset. MMM contains artificial subsets in the form of difficulty levels. Later classifiers are better at classifying the more difficult samples, and early classifiers are more suitable for classifying the easy one. A policy network is able to discern this relation and it manages to attain a performance of 51.83%. In Appendix A we showed that the frequency domain is likely not a reliable indicator of the difficulty of a sample, nor is it a discriminative feature that can be used to differentiate between the subsets that classifiers in a MSDNet end up learning. Next, we attempted to train a policy network on more traditional visual datasets in the form of CIFAR100 and SHVN. In this case, there was no sign that the policy network is able to learn anything meaningful that enables it to reliably predict where a sample should end up in the respective MSDNet. The negative results suggest that, in general, classifiers do not specialize in specific subsets of the data. In light of these results, we offer an alternative possible explanation for overthinking. Overthinking is introduced randomly into a network due to stochastic processes inherent to training. As multiple classifiers are trained in joint fashion, the only goal is to maximize the average performance of all classifiers. The training process does not discriminate between classifiers, if a single classifier does not end up improving so the others can, then this is a desirable outcome. How the parameters of a network update after seeing a training sample all depends on the state of the network at that moment in time and many stochastic processes, such as the learning order, influence said state.

To conclude, we summarize our main findings and succinctly answer our initial research questions.

- *Can early-exiting in MSDNet be learned directly by means of a policy network?*
  Yes, but only under rather serendipitous circumstances, as was the case in MMM. It seems there needs to be inherent differentiation between subsets of the dataset.

- *Do classifiers in MSDNet end up specializing in specific subsets of the original dataset?*
  There is no reason to think that they do, the policy network is unable to uncover any substantial relations between the data and the classifiers of an MSDNet.

- *Are there any image statistics that can be used to discriminate between the subsets that classifiers in an MSDNet end up learning?*
  This question is to an extent already answered by the findings stated in the previous question. To add to that, we know that the frequency domain and discrete cosine transform analysis do not suffice as discriminatory features to separate the learned subsets of MSDNets.

- *What causes overthinking in MSDNets?*
  Overthinking does not seem to be caused by specialization of classifiers in an MSDNet. Instead, it seems more likely that it is caused by stochasticity inherent to the training process.

5.2. Limitations and future work

This thesis mainly ended up attempting to prove the absence of specialization of classifiers in MSDNets. Proving the absence of something is often more difficult, if not impossible, as opposed to proving the existence of a particular phenomenon. In the latter case you only need one instance to convincingly

prove the hypothesis. An attempt to disprove the existence of anything often leaves something to be desired; there is always another test that can be done or another experiment that can be run to add more credibility to the findings. The field of deep learning often exacerbates this issue due to its black box nature, hence the apparent reproducibility crisis. We have attempted to minimize the room for interpretation of the results when disproving the existence of specialization by first showing that the policy network setup functions, and subsequently by applying the exact same strategy to conventional datasets. On top of that, we simplified the problem by reducing it to only two classifiers of the MSDNet. All that being said, there is always cause for healthy skepticism and we have tried to convey this sentiment with our verbiage, often using 'most likely' and 'probably' instead of 'definitely' and 'guaranteed'. More concretely, when it comes to our policy network not finding any correlations between the data and classifiers in an MSDNet, it does not guarantee directly that specialization does not occur. It could be the case that specialization occurs but only to a non-noticeable degree. It could also be that it does occur but stochasticity muddles the signs which make it impossible for our policy network to pick up. Neither case is verifiable by our method, and should be kept in mind when interpreting the results laid out in this thesis.

When it comes to our alternative explanation on what causes of overthinking, it is worth noting that it assumes that specialization does not occur. Also, it is by all means simply a theory that seems most likely given the circumstances; there seems to be no reason to assume that the data itself causes specific overthinking and it happens in every early-exiting setup. This leaves us with the final horse of the three horsemen: the training process. As we have not proven that this is indeed the case in this thesis, we leave this as future work.

Finally, something that seems worth pursuing is research into training an early-exit setup specifically for early-exiting. Joint optimization promotes the maximization of individual classifiers, but they lack the ability to collaborate and coordinate well in an adaptive inference setting. Training the classifiers specifically for something like budgeted batch classification might be worthwhile future work.

Bibliography

[1] [user20160]. *Can a perceptron with sigmoid activation function perform nonlinear classification?* Feb. 2017. URL: https://stats.stackexchange.com/questions/263768/can-a-perceptron-with-sigmoid-activation-function-perform-nonlinear-classificati.

[2] Manuel Amthor, Erik Rodner, and Joachim Denzler. "Impatient dnns-deep neural networks with dynamic time budgets". In: *arXiv preprint arXiv:1610.02850* (2016).

[3] Mariusz Bojarski et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[4] Tolga Bolukbasi et al. "Adaptive neural networks for efficient inference". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 527–536.

[5] Tom B Brown et al. "Language models are few-shot learners". In: *arXiv preprint arXiv:2005.14165* (2020).

[6] Bert Carremans. *Classify butterfly images with deep learning in Keras*. Jan. 2019. URL: https://towardsdatascience.com/classify-butterfly-images-with-deep-learning-in-keras-b3101fe0f98.

[7] Guobin Chen et al. "Learning efficient object detection models with knowledge distillation". In: *Advances in neural information processing systems* 30 (2017).

[8] Zhourong Chen et al. "You look twice: Gaternet for dynamic filter selection in cnns". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9172–9180.

[9] 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*. 2020, pp. 235–244.

[10] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[11] Biyi Fang et al. "Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision". In: *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2020, pp. 84–95.

[12] Michael Figurnov et al. "Spatially adaptive computation time for residual networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 1039–1048.

[13] Yunchao Gong et al. "Compressing deep convolutional networks using vector quantization". In: *arXiv preprint arXiv:1412.6115* (2014).

[14] Alex Graves. "Adaptive computation time for recurrent neural networks". In: *arXiv preprint arXiv:1603.08983* (2016).

[15] Jiaqi Guan et al. "Energy-efficient amortized inference with cascaded deep classifiers". In: *arXiv preprint arXiv:1710.03368* (2017).

[16] Yizeng Han et al. "Dynamic neural networks: A survey". In: *arXiv preprint arXiv:2102.04906* (2021).

[17] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[18] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[19] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[20] Gao Huang et al. "Multi-scale dense networks for resource efficient image classification". In: *arXiv preprint arXiv:1703.09844* (2017).

[21] Jörn-Henrik Jacobsen et al. "Multiscale hierarchical convolutional networks". In: *arXiv preprint arXiv:1703.04140* (2017).

[22] Jeremy Jordan. *Convolutional neural networks*. Apr. 2019. URL: https://www.jeremyjordan.me/convolutional-neural-networks/.

[23] Ajay J Joshi, Fatih Porikli, and Nikolaos Papanikolopoulos. "Multi-class active learning for image classification". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2009, pp. 2372–2379.

[24] Nahua Kang. *Multi-layer neural networks with sigmoid function- deep learning for rookies (2)*. Feb. 2019. URL: https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f?source=----------4----------.

[25] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. "Shallow-deep networks: Understanding and mitigating network overthinking". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3301–3310.

[26] Tsung-Wei Ke, Michael Maire, and Stella X Yu. "Multigrid neural architectures". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 6665–6673.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[29] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. "Fractalnet: Ultra-deep neural networks without residuals". In: *arXiv preprint arXiv:1605.07648* (2016).

[30] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. "Adaptive Inference through Early-Exit Networks: Design, Challenges and Directions". In: *arXiv preprint arXiv:2106.05022* (2021).

[31] Stefanos Laskaridis et al. "HAPI: hardware-aware progressive inference". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2020, pp. 1–9.

[32] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[33] Yann LeCun, Corinna Cortes, and Chris Burges. *MNIST handwritten digit database*. 2010.

[34] Hao Li et al. "Improved techniques for training adaptive deep networks". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1891–1900.

[35] Hongfeng Li et al. "Skin disease diagnosis with deep learning: a review". In: *Neurocomputing* 464 (2021), pp. 364–393.

[36] Haoning Lin, Zhenwei Shi, and Zhengxia Zou. "Maritime Semantic Labeling of Optical Remote Sensing Images with Multi-Scale Fully Convolutional Network". In: *Remote Sensing* 9 (May 2017), p. 480. DOI: 10.3390/rs9050480.

[37] Shaohui Lin et al. "Towards optimal structured cnn pruning via generative adversarial learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2790–2799.

[38] Jian-Hao Luo et al. "Thinet: pruning cnn filters for a thinner net". In: *IEEE transactions on pattern analysis and machine intelligence* 41.10 (2018), pp. 2525–2538.

[39] Stéphane Mallat. "Understanding deep convolutional networks". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065 (2016), p. 20150203.

[40] Seyed Iman Mirzadeh et al. "Improved knowledge distillation via teacher assistant". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5191–5198.

[41] M Narasimha and A Peterson. "On the computation of the discrete cosine transform". In: *IEEE Transactions on Communications* 26.6 (1978), pp. 934–936.

[42] Yuval Netzer et al. "Reading digits in natural images with unsupervised feature learning". In: (2011).

[43] Guillermo Ortiz-Jimenez et al. "Hold me tight! Influence of discriminative features on deep network boundaries". In: *arXiv preprint arXiv:2002.06349* (2020).

[44] Eunhyeok Park et al. "Big/little deep neural network for ultra low power inference". In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2015, pp. 124–132. DOI: 10.1109/CODESISSS.2015.7331375.

[45] Iuliia Pliushch et al. "When Deep Classifiers Agree: Analyzing Correlations between Learning Order and Image Statistics". In: *arXiv preprint arXiv:2105.08997* (2021).

[46] David Refaeli. *Cross entropy*. URL: https://themaverickmeerkat.com/2020-11-15-Cross-Entropy/.

[47] *Relu*. URL: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html.

[48] Shreyas Saxena and Jakob Verbeek. "Convolutional neural fabrics". In: *Advances in neural information processing systems* 29 (2016), pp. 4053–4061.

[49] Ramprasaath R Selvaraju et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 618–626.

[50] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[51] Kousai Smeda. *Understand the architecture of CNN*. Nov. 2019. URL: https://towardsdatascience.com/understand-the-architecture-of-cnn-90a25e244c7.

[52] Christian Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015). DOI: 10.1109/cvpr.2015.7298594. URL: https://arxiv.org/abs/1409.4842.

[53] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[54] 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)*. IEEE. 2016, pp. 2464–2469.

[55] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[56] Xin Wang et al. "Idk cascades: Fast deep learning by learning not to overthink". In: *arXiv preprint arXiv:1706.00885* (2017).

[57] Zifan Wang et al. "Towards frequency-based explanation for robust cnn". In: *arXiv preprint arXiv:2005.03141* (2020).

[58] Yi Wei et al. "Quantization mimic: Towards very tiny cnn for object detection". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 267–283.

[59] Zuxuan Wu et al. "Blockdrop: Dynamic inference paths in residual networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8817–8826.

[60] Ji Xin et al. "Deebert: Dynamic early exiting for accelerating bert inference". In: *arXiv preprint arXiv:2004.12993* (2020).

[61] Le Yang et al. "Resolution adaptive networks for efficient inference". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 2369–2378.

[62] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. "Designing energy-efficient convolutional neural networks using energy-aware pruning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5687–5695.

[63] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[64] Yisu Zhou, Xiaolin Hu, and Bo Zhang. "Interlinked convolutional neural networks for face parsing". In: *International symposium on neural networks*. Springer. 2015, pp. 222–231.

Appendix

A

The frequency domain as a discriminative feature

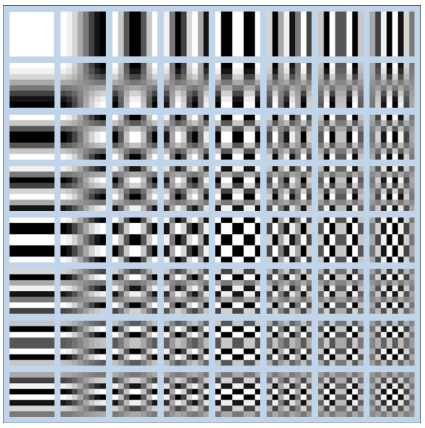


Figure A.1: Shows the performance of M_{ee} on the test set of D_{ee} for each label and respective levels.

Every image can be decomposed into individual frequencies via frequency transformations. The original image can be reconstructed by combining the individual frequencies again. On average, images consist mostly of lower frequencies, meaning the majority of information contained in an image can be described by these lower frequencies. Compression algorithms such as JPEG make use of this fact to compress images; they discard information about higher frequencies which only account for a small percentage of the image energy.

Discrete cosine transforms (DCTs), like Fourier transforms (FTs), transform a signal into the frequency components. DCT only uses real numbers as opposed to FT. Given a 1 channel image I in $\mathbb{R}^{H\times W\times C}$, we can denote a pixel in this image with I_{hw}. The resulting frequency decomposition of this image obtained by applying a 2D DCT is given by:

$$F_{vu} = \frac{1}{4} C_v C_u \sum_{h=0}^{N-1} \sum_{w=0}^{N-1} I_{hw} cos(v\pi \frac{2h+1}{2N}) cos(v\pi \frac{2w+1}{2N}) \tag{A.1}$$

$$C_u, C_v = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases} \tag{A.2}$$

where F_{vu} denotes a value, also called the coefficient, in the resulting matrix of F. If the image is $H \times W$, the resulting decomposition will also be $H \times W$. Each coefficient in the decomposition denotes

the contribution of a vertical and horizontal cosine signal with frequency h and w. If we consider an $8 \times 8 \times 1$ image, a decomposition would look like figure A.1, with a weighted combination of each of those frequencies you could reconstruct any $8 \times 8 \times 1$ image. 2D DCT assigns weights to those frequencies that show much they contribute to the original image. The top left coefficients represents lower frequencies, whereas the bottom right coefficients represent the higher frequencies.

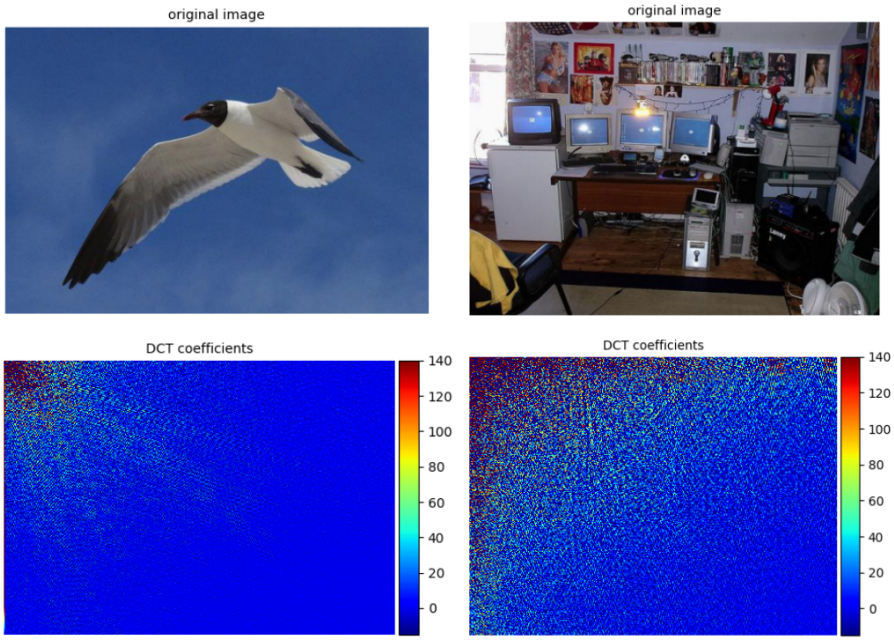


Figure A.2: Shows the performance of M_{ee} on the test set of D_{ee} for each label and respective levels.

When we apply 2D DCT to regular images and plot the corresponding coefficients in the form of a heatmap you get what is depicted in figure A.2. First, it shows that the majority of the image energy is present in the lower frequencies. Secondly, more "cluttered" images on average tend to contain more energy in the higher frequencies. Clutterness has also be associated with image complexity [45] when studying the learning order of CNNs. The idea is that the more higher frequencies are needed to reconstruct the image to a certain degree, the more cluttered and thus more complex the image is. We use these concepts to determine whether DCT can be used to distinguish the learned subsets of classifiers in an MSDNet.

We flatten the coefficient matrix resulting from applying a 2D DCT, horizontally. We then select the first n coefficients that add up to 98% of the total sum of coefficients, presented as a percentage of the total number of coefficients. The higher the value is, the more high frequencies are needed to reconstruct the image with sufficient accuracy. We do this for every subset of images each classifier gets correct. The results are presented in figures A.3, A.4 and A.5. The datasets used are SHVN, a subset of Imagenet called Imagenette[1] and MMM, respectively. On SHVN and Imagenette we use a 2-classifier setup, for MMM a 6-classifier. There does not seem to be a way to distinguish learned subsets using DCT as the variance in the results is significantly larger than the differences themselves. We therefor suggest that the frequency domain is likely not useful as a discriminatory feature to base an early-exiting policy on.

[1] https://github.com/fastai/imagenette

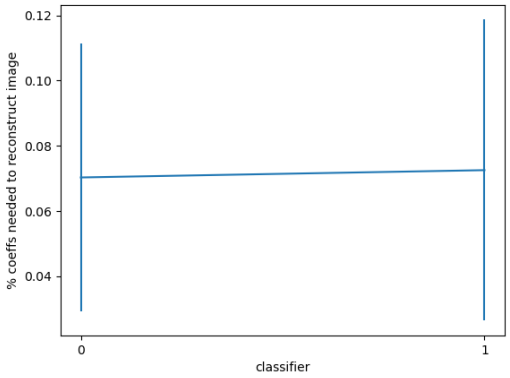


Figure A.3: **(SHVN).** Shows the average percentage of coefficients that are required to reconstruct the images of the learned subset of each classifier with a 2% error rate. A higher value indicates that the images are more cluttered. The vertical bars represent the variance.

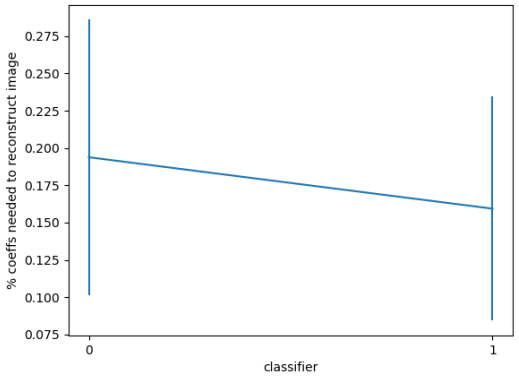


Figure A.4: **(Imagenette).** Shows the average percentage of coefficients that are required to reconstruct the images of the learned subset of each classifier with a 2% error rate. A higher value indicates that the images are more cluttered. The vertical bars represent the variance.

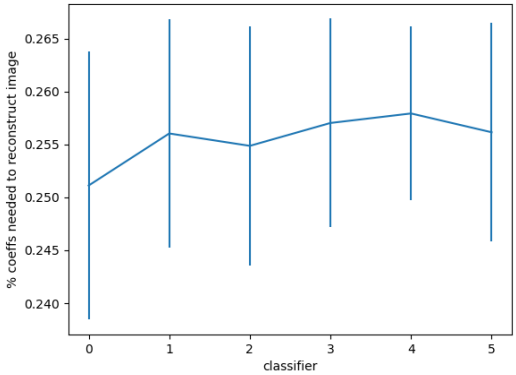


Figure A.5: **(MMM).** Shows the average percentage of coefficients that are required to reconstruct the images of the learned subset of each classifier with a 2% error rate. A higher value indicates that the images are more cluttered. The vertical bars represent the variance.

B

Paper

Explaining overthinking in Multi-Scale Dense networks

D. Voorhout, X. Liu, J. van Gemert
Delft University of Technology
D.M.Voorhout@student.tudelft.nl, {X.Liu-11, j.c.vangemert}@tudelft.nl

Abstract

Dynamic neural networks are becoming more popular as they are able to adapt to changing computational constraints. One of the more common paradigms in this regard is early-exiting, a technique which allows networks to iteratively process the input until either the computational budget has been spent or the network has become confident enough in its prediction. The average performance of each classifier in such a setup increases monotonically as their capacity grows, yet this does not have to hold for individual inputs. Network overthinking has been documented in several traditional convolutional neural networks with added intermediate classifiers. One possible explanation is that later classifiers make use of more complex features due to the increased receptive fields which no longer represent easy samples. However, we have observed overthinking in Multi-Scale Dense networks for which the given argument in relation to the receptive field does not hold due to its unique architecture. This paper shows that policy networks are unable to successfully learn an early-exit strategy for trained Multi-Scale Dense networks, suggesting a lack of classifier specialization and subsequent correlation between overthinking and the data. Instead, we offer up a different theory, that overthinking in Multi-Scale Dense networks is caused by the inherent stochasticity of the learning process.

1. Introduction

Convolutional neural networks (CNNs) have shown to be exceedingly successful on a number of visual recognition tasks [11, 13, 18, 25]. State-of-the-art models, such as ResNet [11], DenseNet [13] and GoogleNet [25], are deeper than ever before to break performance benchmarks. However, with increased depth comes an increase in computation which limits their use cases in practical settings where computational constraints play a role.

A sizable body of work has already been dedicated to the optimization of deep neural networks, ranging from knowledge distillation [3, 22] and parameter reduction [20, 21, 32] to weight quantization [8, 28]. These type of methods are often model-agnostic and can even improve performance. However, whether or not these methods are implemented, traditional deep neural networks still have static computational graphs and fixed parameters. As a consequence, a static neural network will always spend the same amount of computation if the input dimensions remain fixed. Some samples do however not need as much processing as others for the network to come to an accurate prediction; some samples are easier than others. Spending unnecessary computation on easy samples increases both the network's inference time and power consumption. Ideally, we would have the network only spend as much computation as it needs to. This is the essence of early-exiting; the input is processed iteratively until either the computational budget has been spent or the network has become confident enough in its prediction. Not only do these type of dynamic networks enjoy an overall reduction of computation, they are also compatible with existing efficiency optimization techniques and more widely applicable in real world settings due to their flexibility.

There are two settings where dynamic neural networks have a distinct advantage over their static counterparts: During anytime prediction where a network is asked to output a result at a moments notice, and budgeted batch classification where a group of samples need to be processed within a singular fixed computational budget. In the latter case, a dynamic network will be able to maximize its performance by spending more of the budget on harder samples. Search engines indexing images is an example of a budgeted batch classification scenario. For a real world example of anytime prediction, think of self driving cars [1] where road signs need to be classified to potentially alter the vehicle's driving behaviour. Road signs can appear around corners, the car can be driving at various speeds and there might be stochasticity in the speed with which signs are recognized in the first place. All these variables lead to variability in time constraints imposed on the classification network of the vehicle. Static networks will not be able to adapt to the changing circumstances, whereas dynamic networks can.

Classifiers in a multi-exit network have the advantage of

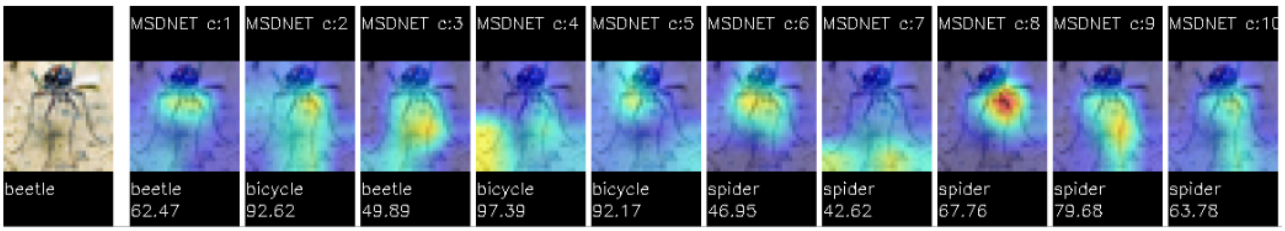


Figure 1. Shows the heatmap of what classifiers focus on in a 10-block Multi-Scale Dense network in a case where it overthinks. The image on the left shows the original input. Each heatmap also shows the respective classifiers' prediction and its softmax confidence in that prediction. There seems to be no pattern to what each classifier focuses on.

being able to make use of computation performed by previous classifiers. Deeper classifiers, with their increased capacity, therefore perform better than their shallower counterparts on average. However, deeper classifiers can misclassify samples that shallower classifiers classify correctly. This overthinking phenomenon has been documented in multi-exit networks that use traditional convolutional networks as the backbone by Kaya et al. [15]. A potential reason for this phenomenon occurring under these circumstances is that deeper classifiers make use of layers with larger receptive fields. These layers thus represent more complex features, features which are no longer suitable to classify the easier samples of the dataset. The deeper classifiers end up finding these complex features in samples that do not contain them and subsequently assigning a high confidence to them, leading to misclassification. These easy samples can however be classified by shallower classifiers which make use of more primitive feature maps. More computation can thus lead to worse results. However, we have observed overthinking taking place in Multi-Scale Dense networks (MSDNets) [12], a hand-tuned end-to-end architecture optimized for early-exiting. Multi-scale architectures do not conform to the common notion of how receptive fields, and with it the complexity of the features, grow along the depth of the network. Instead, aggressive downsampling in the first layer seeds every scale with feature maps that differ in their size of receptive field. The highest scale contains the coarsest features that have full receptive field and multi-scale architectures maintain these scales along the depth of the network. For this reason, the argument for overthinking in relation to the receptive field does not hold for MSDNets. Figure 1 shows an example of what each classifier in an MSDNet focuses on in a case where it overthinks. There seems to be no clear relation to be found between the depth of an MSDNet and what a classifier at that respective depth focuses on. This relation can be found in multi-exit networks with a traditional backbone as was shown by Kaya et al. due to the more linear nature of receptive fields in such networks.

Understanding overthinking is closely tied to understanding why some classifiers correctly classify particular samples and why other classifiers can not. This knowledge can subsequently be used to improve upon existing early-exiting strategies, improving adaptive inference performance and reducing computation of dynamic neural networks.

In this paper we show that a separate network can under serendipitous circumstances learn to find a correlation between the data and what a classifier in an MSDNet learns, but not in general. This suggests that classifiers in an MSDNet do not specialize in particular subsets of the dataset. In other words, the overthinking patterns in MSDNets are likely not tied to the data itself, suggesting instead that is caused by stochasticity in the learning process.

We first train a policy network on Max-Min MNIST [5] to show that policy networks can learn early-exit strategies for MSDNets when the respective dataset leads to a strong dependency between the data and the classifiers in the networks. We then show that this does not hold in general by unsuccessfully training policy networks on CIFAR100 [17] and the SHVN [23] datasets.

2. Related Work

Resource Efficient Neural Networks. Much of the existing literature on reducing the computational burden of deep neural networks has been focused on networks after they have been trained. Knowledge can be distilled from a larger network into a smaller network, effectively reducing the number of parameters while maintaining performance [3, 22]. Another way is to prune redundant parameters directly by iteratively removing parameters that contribute the least to the result [20, 21, 32]. A more implementation focused method of reducing computation is to collapse parameter data into a more primitive form by means of quantization, effectively reducing computational overhead [8, 28]. Most of these methods can be used in tandem as well as with adaptive inference.

Adaptive Inference. Early-exiting in its most trivial form can be performed by making use of two networks with vary-

Dataset	Training methodology		
	50 epochs	100 epochs	50 epochs and 50 epochs ISC + OFA
CIFAR100	17.93	18.45	**14.33**
SHVN	15.32	15.50	**11.29**

Table 1. Shows the percentage of samples the network overthinks on for the test set of the respective dataset. Lower is better. It shows that using learning techniques which stimulate classifier collaboration reduces overthinking.

ing capacity and computational requirements and turning them into a model cascade. In most cases the smaller network suffices and the bigger network is only required for the more complex inputs [24]. This scheme can be extended with multiple networks, where the inference is done by each model in a set order from smallest network to hardest [27]. A more nuanced approach can also be taken and certain models can be skipped if desired [2]. Model cascades can be distilled down into a single network where the backbone consists of any regular convolutional neural network and any number of intermediate classifiers are added as early-exits. This concept of intermediate early-exits operates under the assumption that many datasets contain samples that can already be classified using features that are derived in those early layers [6, 15, 19, 26].

Information exchange that leads to spatial shift invariance can be achieved with significantly fewer parameters if features maps of all scales could transfer information directly as opposed to only along the depth of the network [14, 16]. Multi-Scale Dense networks make use of these concepts to represent feature maps with maximum receptive fields along every layer of the network [12]. Furthermore, MSDNets make use of dense connections [13] to prevent shallow feature maps from collapsing in favour of intermediate classifiers. Together, these techniques allow MSDNets to produce state-of-the-art results in the context of adaptive inference.

Exiting Policies. Deciding when a sample should exit in a multi-exit network is reminiscent of the stopping problem. Many techniques for this reason make use of reinforcement learning, where some sort of halting score is introduced that weighs potential gain in performance versus the cost of extra incurred computation [5, 7, 9, 10]. Another popular method is to make use of confidence based criteria, if the confidence of a classifier crosses a predetermined threshold, the sample exits at that classifier. Measures of network confidence include the maximum of the softmax [12, 31] and the entropy [26, 30] of the output.

Chen et al. [4] and Wu et al. [29] show that policy networks are able to select layers of a ResNet that should be activated to maximize inference performance and efficiency. Wu et al. in particular note that the policy network selects similar parts of the inference network for the same class, signifying a relation between parts of the inference network and the data. Motivated by these works we attempt to train a policy network to learn to early-exit in MSDNets to determine whether a deterministic relation exists between the data and what classifiers in an MSDNet learn. Such an existence could in turn be an explanation for overthinking.

3. The Overthinking Phenomenon

Overthinking takes place when a classifier positioned deeper in the network misclassifies a sample whereas a classifier placed before it can classify it correctly. More formally we define overthinking as follows:

$$f_i(\mathbf{x};\theta_i) = \mathbf{y} \wedge f_{i+k}(\mathbf{x},\theta_{i+k}) \neq \mathbf{y} \; where \; k \in \{1,\ldots,n-i\}, \tag{1}$$

where $f_i(\mathbf{x};\theta_i)$ represents the prediction of the classifier at depth i, parameterized by θ_i on input $\mathbf{x}$. $\mathbf{y}$ is the label vector of the input and n the total number of classifiers in the multi-exit network. In general, the set of samples S_j a classifier at depth j gets correct is larger than the set of samples S_i a classifier at depth i gets correct, where $i < j$, yet $S_i \not\subset S_j$. As a result, the adaptive performance of a multi-exit network like MSDNet exceeds that of the average performance of its individual classifiers. This effect is shown in figure 2. The anytime prediction shows the performance of each individual classifier, whereas the budgeted batch performance indicates what the network can achieve when it is allowed to early-exit. A sample exits the network when the respective classifiers' confidence crosses a predetermined threshold. Early-exiting allows the network to output a prediction before it has a chance to overthink, directly combating the overthinking phenomenon. This is why the budgeted batch performance is larger than a weighted combination of individual classifiers. If a perfect early-exiting strategy could be achieved, both efficiency and adaptive performance of dynamic neural networks would increase drastically, as indicated by the upper bounds.

Reducing Overthinking. Krizhevskyet al. [17] showed that improving collaboration between classifiers in a multi-exit network improves their performance. They introduce Inline Subnetwork Collaboration (ISC), a technique that implements direct connections between classifiers, and One-for-all Knowledge Distillation (OFA), a teacher parent training technique where the final classifier acts as the teacher for intermediate classifiers. We show that these techniques can be used to reduce overthinking in MSDNets by training a network conventionally for 50 and 100 epochs and comparing the rate of overthinking when the network is trained using the improved training techniques. We follow

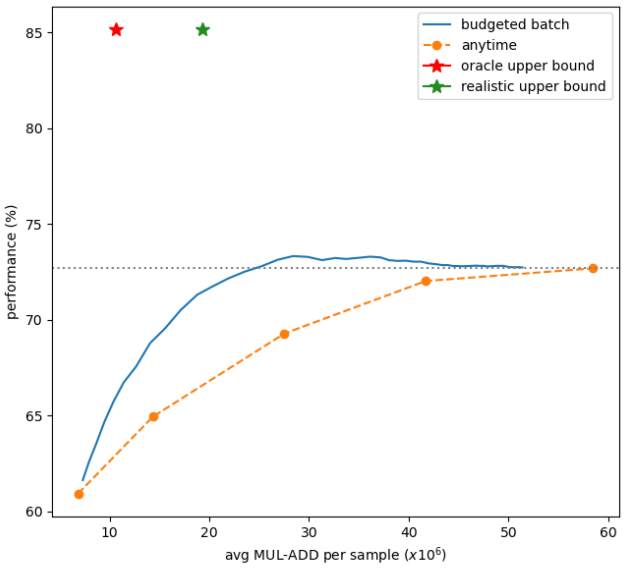


Figure 2. Shows anytime prediction and budgeted batch performance of applying a 5-block Multi-Scale Dense network on the CIFAR100 test set. The upper bounds represent results obtained when applying a perfect early-exit strategy. The oracle is clairvoyant and does not perform inference if no classifier can correctly classify the sample. The performance of budgeted batch classification surpasses that of anytime prediction due to early-exiting.

the convention of Krizhevskyet al. to only use the techniques to fine-tune the network, which we do for the final 50 epochs of the training process. The results on the CIFAR100 [17] and SHVN [23] datasets can be seen in table 1. So while increased classifier collaboration reduces overthinking and increases performance, it does not explain where overthinking and classifier independecy stems from in the first place.

4. Method

To show that there are likely no relations between the data and overthinking in MSDNets in general, we train a policy network to directly learn an early-exit policy for the respective MSDNet. We evaluate the policy network on three datasets and show that in general, the policy network is unable to successfully learn the early-exiting strategy of an MSDNet.

Every experiment consists of the 4 following steps:

1. Choose a dataset D that the MSDNet will train on.
2. Train an MSDNet M on dataset D.
3. Create a dataset from M containing pairs of samples (d, l) with $d \in D$ and l representing the l^{th} classifier in M that is most suitable to classify sample d. We call this new early-exiting dataset D_{ee}.
4. Train a traditional CNN M_{ee} on D_{ee} to learn the early-exit strategy of M. M_{ee} is thus the policy network. M_{ee} is subsequently evaluated on the test set of D_{ee}.

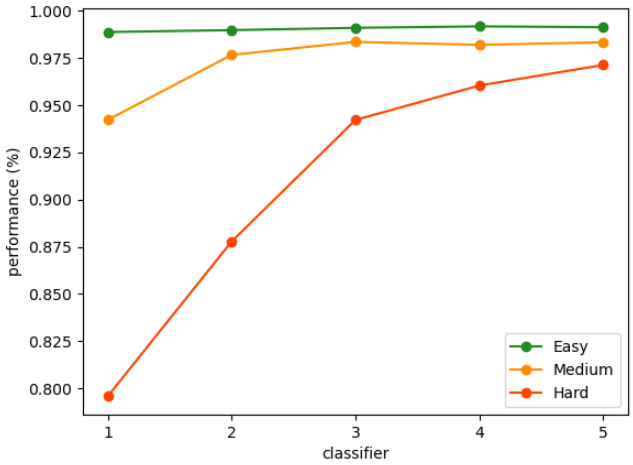


Figure 3. Shows the performance of a 5-classifier Multi-Scale Dense network on Max-Min MNIST for each of its respective difficulty levels. The final classifier is significantly better at classifying harder samples than shallow classifiers, whereas the performance on easy samples is nearly identical.

For D we use Max-Min MNIST (MMM) [5], CIFAR100 and SHVN. We use the special dataset MMM to show that the experiment setup is able to bring to light correlations between learned subsets by classifiers of M and the data as long as D allows for enough classifier specialization to begin with. With the experiments on CIFAR100 and SHVN we show that in more conventional datasets the policy network is not able to learn of any such relations, even in a simplified setting.

M consists of a 5-classifier MSDNet trained for 200 epochs with a learning rate of 0.1 which decreases by a factor of 10 at epoch 75 and 150. The network has 3 scales and a linear growth in layers, we follow the conventions of [12] when it comes to other architectural considerations.

M_{ee} is a ResNet with 34 layers, trained with a learning rate of 0.1 for 100 epochs which decreases tenfold at epoch 50 and 75. At times, training was cut short if no progress was being made.

Creating D_{ee}. The goal of D_{ee} is to capture the specific subsets that classifiers in an MSDNet learn. Every label l ideally represents the classifier for whose learned subset the sample d is most representative. As it often happens that multiple classifiers in M can correctly classify the sample d, choosing l becomes ambiguous. However, it can be ar-

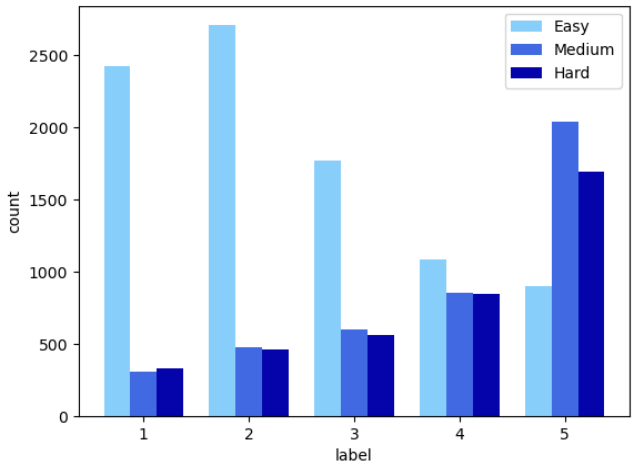


Figure 4. Shows the sample distribution of the early-exiting training set obtained from applying the scoring method with $\lambda = 5$ on an MSDNet that was trained on Max-Min MNIST. As the final classifier in the MSDNet is significantly better at classifying harder samples than shallow classifiers, the scoring method labels those samples accordingly.

gued that in such occasions, the classifier with the highest confidence represents the classifier for which the sample is most emblematic. We therefor make use of the following scoring system to determine the labeling of D_{ee}:

$$s_i = \frac{1}{\lambda m_i(\mathbf{x}) 10^{-6}} + c_i \text{ for } i \in 1, \dots, n. \tag{2}$$

The score s_i each classifier i gets assigned is determined by its confidence c_i in its prediction and the amount of computation $m_i(\cdot)$ it requires to come to a conclusion on input $\mathbf{x}$ in terms of MUL-ADDs. n is the total number of classifiers in the network. We use the maximum value of the softmax output of a classifier's prediction as a measure of confidence. Taking into account the efficiency of each classifier is important to not skew the distribution of D_{ee} too much. Without it, most samples would be labeled according to the last classifier in the network as it has the highest performance and confidence on average. γ allows us to have more control over the distribution of D_{ee}. We use this to make D_{ee} as balanced as possible. Each classifier i only gets assigned a score if it is able to classify the sample correctly in the first place. It gets assigned 0 otherwise:

$$s'_i = \begin{cases} s_i, & \text{if } f_i(\mathbf{x}, \theta_i) = \mathbf{y} \\ 0, & \text{otherwise} \end{cases} \text{ for } i \in 1, \dots, n, \tag{3}$$

where s'_i is the final score classifier i gets assigned. $f_i(\mathbf{x}; \theta_i)$ represents the prediction of the respective classifier, parameterized by θ_i on input $\mathbf{x}$. $\mathbf{y}$ is the label vector of the input. If all classifiers are unable to correctly classify sample d, d is left out of D_{ee} as it does not represent any of the learned subsets of the classifiers of M. The label then becomes the classifier with the highest score:

$$l = argmax_{s'_i}(\dots, s'_i, \dots) \text{ for } i \in 1, \dots, n. \tag{4}$$

Despite using the scoring method to create the training and test set of D_{ee}, it often still ends up imbalanced. For this reason, we will make use of weighted learning when training M_{ee}:

$$\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -w_y log(\frac{e^{\mathbf{z}_y}}{\sum_i^n e^{\mathbf{z}_i}}), \tag{5}$$

where $\mathcal{L}_{CE}(\hat{\mathbf{y}}, \mathbf{y})$ is the cross-entropy loss between the prediction $\hat{\mathbf{y}}$ and the label vector $\mathbf{y}$, w_y is the weight specific to class y and $\mathbf{z}_y$ the output of the classifier for class y. The weights are determined for the training set before training starts. While the weights can take any value, we will make them proportional to the occurrence of each class: If we consider an ordered array containing the occurrences of each class in the training set and denote it with C_{train}, the weights become $w_y = \frac{max(C_{train})}{C_{train}[y]}$ for $y \in \{1, \dots, n\}$. In our case $n = 5$ as M contains that many classifiers. Effectively, weighted learning discourages M_{ee} from simply returning the most common class by penalizing incorrect classifications of minority samples more harshly.

For the training set of D_{ee} we use the validation set of D. We set aside 20% of training samples of D each time for validation. The test set of D_{ee} is created from the test set of D. We omit the validation set for D_{ee} to maximize the amount of training samples.

5. Experiments

The policy network M_{ee} is first trained on Max-Min MNIST (MMM) [5] to show that it can pick up on classifier specialization when a data dependency is explicitly introduced. We then show that without this explicit differentiation, policy networks are seemingly unable to find any correlation between the data and classifiers in am MSDNet by evaluating the policy network on CIFAR100 [17] and SHVN [23].

5.1. Evaluation on Max-Min MNIST

We first evaluate the performance of M_{ee} that aims to predict the early-exiting strategy of MSDNet M in the context of the MMM dataset, a dataset consisting of 10 labels, each with 3 difficulty levels. Figure 5 shows an example of each of the difficulty levels. The performance of M on this dataset can be seen in figure 3. The final classifier is significantly better at classifying hard samples than the first classifier, whereas their performance on easy samples is identical. This discrepancy leads to a noticeable differentiation in the

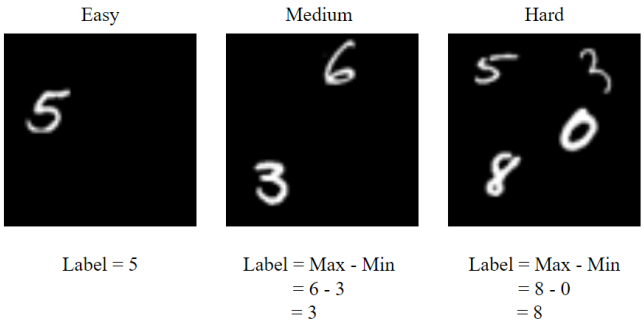


Figure 5. Shows examples of samples from the Max-Min MNIST dataset. The labels for the medium and hard cases are determined by subtracting the smallest digit from the largest digit in the image.

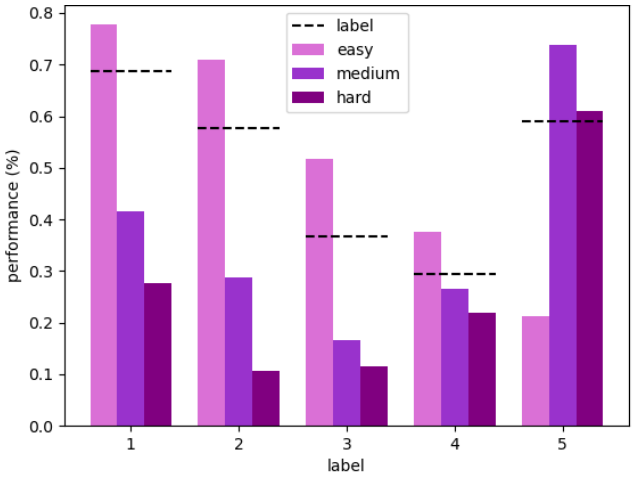


Figure 6. Shows the performance of the Resnet34 policy network on the test set of the Max-Min MNIST D_{ee} dataset for each label and respective difficulty levels. The policy network is better at distinguishing samples that should go to the first or last classifier than classifiers inbetween.

label distribution of D_{ee}, as can be seen in figure 4. D_{ee} in this case thus suggests that easy samples resemble the learned subsets of earlier classifiers more and medium and hard samples resemble learned subsets of the last classifier more, representing a form of classifier specialization.

The training set of D_{ee} contains roughly 15,000 samples, and the test set likewise. We train M_{ee} for 60 epochs as it makes no progress after that. The overall performance over the whole test set of D_{ee} is 51.83%. The breakdown of how it performs for each label and difficulty level is shown in figure 6. Several trends stand out, first, M_{ee} is clearly better at classifying samples at the tails of the label distribution. Also, it is poor at recognizing exceptions such as early samples that should end up at the final classifier, and medium and hard samples that should end up at intermediate classifiers. In general, it seems to not be able to conceptualize the nuances of classifiers that lie somewhere in the middle of the network.

5.2. Evaluation on CIFAR100 and SHVN

From the results on MMM we can conclude that policy networks are able to learn an early-exiting strategy for MSDNets in the case where classifier differentiation is introduced artificially. We will now show that policy networks are not successful at doing so when it comes to two, more conventional, datasets: CIFAR100 and SHVN. To add veracity to the results, we will simplify the task of the policy network. We saw in the results of the policy network on MMM that it is better at distinguishing samples that should go to either the first or last classifier. We let M_{ee} ignore classifiers in the middle of the network and only focus on the two classifiers on either end. M_{ee}'s task thus effectively turns into binary classification. More concretely, we change equation 3 to:

$$s'_i = \begin{cases} s_i, & \text{if } f_i(\mathbf{x}, \theta_i) = \mathbf{y} \wedge (i = 1 \vee i = n) \\ 0, & \text{otherwise} \end{cases}$$
$$\text{for } i \in 1, \ldots, n. \tag{6}$$

Every test was executed on both datasets 3 times, starting with learning rates 0.1, 0.01 and 0.001. Learning rates were decreased by a factor of 10 at epochs 75 and 150. We only discuss the results in the case where the learning rate is 0.01 as the results for other learning rates are similar.

CIFAR100. Figure 7 shows the binary label distribution of applying the scoring method on M. M in this case is an MSDNet trained for 200 epochs on the CIFAR100 training set. We use $\lambda = 5$ to maintain a relatively balanced dataset. The average performance M_{ee} reaches on the test set of D_{ee} is 43.25%, well below the cutoff of $\sim 50\%$ we consider to be the threshold for having learned anything which can be achieved by random guessing.

SHVN. Figure 8 shows the binary label distribution of the training set of D_{ee}, it was created using the scoring method with $\lambda = 20$. M is trained for 200 epochs on the SHVN training set. The average performance of M_{ee} in this case is 49.92%; M_{ee} ends up returning labels evenly, and thus randomly. Again, there is no indication of M_{ee} having learned anything.

Alternative Theory For Overthinking. In the case of CIFAR100 and SHVN, there is no sign that the policy network

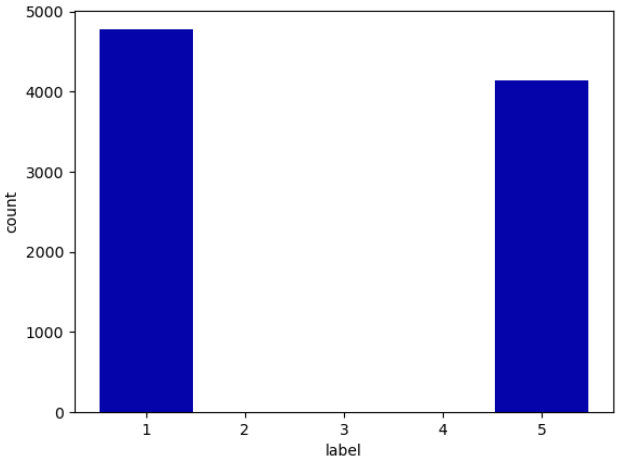


Figure 7. Shows the sample distribution of the training set of D_{ee} obtained by applying the scoring method with $\lambda = 5$ on M. M in this is case is trained on CIFAR100. The lambda is chosen to balance the dataset as much as possible. Internal classifiers are ignored to simplify the subsequent learning process of M_{ee}.

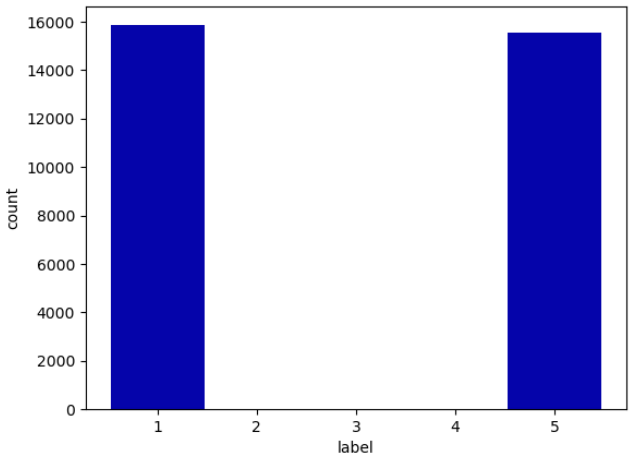


Figure 8. Shows the sample distribution of the training set of D_{ee} obtained by applying the scoring method with $\lambda = 20$ on M. M in this is case is trained on SHVN. The lambda is chosen to balance the dataset as much as possible. Internal classifiers are ignored to simplify the subsequent learning process of M_{ee}.

is able to learn anything meaningful that enables it to reliably predict where a sample should end up in the respective MSDNet. The negative results suggest that, in general, classifiers in MSDNets do not specialize in specific subsets of the data. In light of these results, we offer an alternative possible explanation for overthinking. Overthinking is introduced randomly into a network due to stochastic processes inherent to training. As multiple classifiers are trained in joint fashion, the only goal is to maximize the average performance of all classifiers. The training process does not discriminate between classifiers; if a single classifier does not end up improving so the others can, then this is a desirable outcome. How the parameters of a network update after seeing a training sample all depends on the state of the network at that moment in time and many stochastic processes, such as the learning order, influence the state. This would explain why a policy network is unable to pick up on image statistics that tie learned subsets of classifiers in MSDNets together; they are not deterministic and instead caused by stochastic processes.

6. Conclusion

In this paper we have attempted to directly learn early-exiting strategies in MSDNets by means of a policy network. The idea is that if a policy network is capable of doing so, there must be a relation between the data and what classifiers in an MSDNet have learned. Policy networks would only be able to recognize these correlations if they are strong enough, in other words, if classifiers in an MSDNet tend to specialize in specific subsets of the dataset. This in turn could explain the overthinking phenomenon: Classifiers gravitate toward particular types of samples to maximize the overall performance of the network. The results show that policy networks struggle to ascertain the mentioned relations in conventional datasets. This suggests one of two things: a relation between the data and classifiers in an MSDNet exists but can not be picked up by policy networks or the relation does not exist to begin with. The latter in turn suggests that overthinking does not take place due to specialization. We have seen that policy networks are able to pick up on data and classifier relations when they are present in the case of a dataset like Max-Min MNIST. We thus consider the likelihood of policy networks not being able to pick up on the data correlations for the lack of success of policy networks in the context of MSDNets to be lower than these relations not existing. However, the results can not guarantee directly that specialization does not occur. It could be the case that specialization occurs but only to a non-noticeable degree. It could also be that it does occur but stochasticity muddles the signs which make it impossible for our policy network to pick up. Neither case is verifiable by our method, and should be kept in mind when interpreting the results laid out in this paper.

Finally, as policy networks seem in general unable to ascertain the existence of classifier specialization, we offered an alternative possible explanation for overthinking. Overthinking is introduced randomly into a network due to stochastic processes inherent to training. As we have not proven this, we leave it as future work.

References

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 1

[2] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pages 527–536. PMLR, 2017. 3

[3] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. *Advances in neural information processing systems*, 30, 2017. 1, 2

[4] Zhourong Chen, Yang Li, Samy Bengio, and Si Si. You look twice: Gaternet for dynamic filter selection in cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9172–9180, 2019. 3

[5] 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*, pages 235–244, 2020. 2, 3, 4, 5

[6] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 84–95. IEEE, 2020. 3

[7] 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. 3

[8] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 1, 2

[9] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016. 3

[10] Jiaqi Guan, Yang Liu, Qiang Liu, and Jian Peng. Energy-efficient amortized inference with cascaded deep classifiers. *arXiv preprint arXiv:1710.03368*, 2017. 3

[11] 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. 1

[12] 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. 2, 3, 4

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 1, 3

[14] Jörn-Henrik Jacobsen, Edouard Oyallon, Stéphane Mallat, and Arnold WM Smeulders. Multiscale hierarchical convolutional networks. *arXiv preprint arXiv:1703.04140*, 2017. 3

[15] 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. 2, 3

[16] Tsung-Wei Ke, Michael Maire, and Stella X Yu. Multigrid neural architectures. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6665–6673, 2017. 3

[17] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 2, 3, 4, 5

[18] 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. 1

[19] Stefanos Laskaridis, Stylianos I Venieris, Hyeji Kim, and Nicholas D Lane. Hapi: hardware-aware progressive inference. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020. 3

[20] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2790–2799, 2019. 1, 2

[21] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. Thinet: pruning cnn filters for a thinner net. *IEEE transactions on pattern analysis and machine intelligence*, 41(10):2525–2538, 2018. 1, 2

[22] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, Nir Levine, Akihiro Matsukawa, and Hassan Ghasemzadeh. Improved knowledge distillation via teacher assistant. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5191–5198, 2020. 1, 2

[23] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011. 2, 4, 5

[24] Eunhyeok Park, Dongyoung Kim, Soobeom Kim, Yong-Deok Kim, Gunhee Kim, Sungroh Yoon, and Sungjoo Yoo. Big/little deep neural network for ultra low power inference. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 124–132, 2015. 3

[25] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. 1

[26] 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. 3

[27] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, Fisher Yu, and Joseph E Gonzalez. Idk cascades: Fast deep learning by learning not to overthink. *arXiv preprint arXiv:1706.00885*, 2017. 3

[28] Yi Wei, Xinyu Pan, Hongwei Qin, Wanli Ouyang, and Junjie Yan. Quantization mimic: Towards very tiny cnn for object detection. In *Proceedings of the European conference on computer vision (ECCV)*, pages 267–283, 2018. 1, 2

[29] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8817–8826, 2018. 3

[30] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020. 3

[31] Le Yang, Yizeng Han, Xi Chen, Shiji Song, Jifeng Dai, and Gao Huang. Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2369–2378, 2020. 3

[32] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017. 1, 2