Uncertainty in Deep Learning

Curtis Murray

November 1, 2019

Thesis submitted for the degree of
Honours of Mathematics
in
Statistics
at The University of Adelaide
Faculty of Engineering, Computer and Mathematical Sciences
School of Mathematical Sciences

Contents

List of Tables

List of Figures

Signed Statement

Acknowledgements

Firstly, I would like to thank my supervisor Dr. Jono Tuke for his expert advice and continued support throughout the year, always making his office available to me. I would also like to thank Assoc. Prof. Anthony Dick for his guidance, and suggestions.

I would like to express my gratitude to the Department of Veterans Affairs for their financial support, without which this year would have been all the more challenging.

Finally, I would like to thank my family for providing me with the opportunity to follow my goals. To my girlfriend for her unending support, and my friends for always being there. A special thanks to those friends in the honours room, you've made this challenging year into an enjoyable one, with never a dull moment.

Abstract

Deep learning is an emerging form of machine learning that has achieved state of the art results in many areas, particularly in computer vision and natural language processing [Schmidhuber, 2015]. Although it has achieved great success in these areas, conventional models lack the ability to quantify uncertainty in their predictions [Gal, 2016]. As an example, a model trained to predict whether a picture is of a cat or a dog may be highly effective at doing so, but when given an image of an elephant, instead of the model saying "I'm not sure" by making a prediction with low confidence, it may predict with certainty that the image is of a cat. While this example may be innocent, the implications could be deadly if a deep learning model used to diagnose and treat medical conditions were to fail this spectacularly. The need to develop methods to quantify uncertainty without compromising performance is critical in such sensitive areas. This thesis will consider techniques that facilitate uncertainty quantification by diagnosing potential misclassifications, without the need for formulating new models.

Chapter 1

Introduction

Deep learning is a subfield of machine learning, primarily concerned with predicting relationships between data. Given enough information about something, can we effectively predict an unknown quantity in that thing? Deep learning and other machine learning techniques demonstrate that this is possible, even when the relationship between the information we have and the quantity we are trying to predict is incredibly complex. Sometimes, these relationships *seem* simple; as an example, humans are able to effortlessly classifying whether an image contains either a cat or a dog. However, how can we teach a computer to recognise the difference? An approach may be to tell the computer to look for a set of pointy ears with whiskers, if these features are present, predict that the image is a cat! Now the problem is, how can we teach the computer to recognise pointy ears and whiskers? This in of itself seems just as complex as teaching the computer to detect the cat. Consider now trying to apply this technique to all of the countless objects humans can recognise; it seems impossible. Nevertheless, here, deep learning thrives and has been able to surpass human-level performance [He et al., 2015], with progress continuing to be made at an incredible pace. In the above example, we recognised that it might be useful to break an intricate pattern, such as an image of a cat, up into simpler patterns, such as ears and whiskers. Deep learning takes a similar approach, breaking down an input into what it views as its most basic components, for example, edges in an image. It then combines these patterns to build more complex patterns. Multiple *layers* of patterns are built up hierarchically until a model can accurately detect abstract objects such as cats and dogs, all without the need for manually defining any of the patterns in the hierarchy.

While deep learning is able to make powerful predictions, it is often likened to a black-box because it lacks interpretability. Interpretability can mean many different things, such as the ability to reveal a causal structure in data to explain relationships [Lipton, 2016], or the ability to make trustworthy predictions [Ribeiro et al., 2016]. The necessity of trustworthy predictions is critical in sensitive areas, such as those in medicine, the criminal justice system, and financial markets. Knowing when our model is likely to fail can allow us to implement precautions. As an example, predictions made with low confidence could be verified by a human, or an alternative model.

In this thesis, we look to quantify uncertainty in model predictions. An approach taken by Gal [2016] was to disturb model predictions with some random noise, allowing a distribution of predictions to be obtained. The method of adding noise to the prediction process implemented by Gal [2016] was to apply a process known as dropout. Dropout is a process traditionally associated with preventing overfitting, that works by randomly blocking out a proportion of the patterns the model has been developed to look for. Casting dropout as a noise injection technique to arrive at a distribution of predictions was shown to be effective at quantifying model uncertainty. This thesis follows a similar approach, considering other avenues of noise injection, instead of adding noise to the model. Instead, we look towards image augmentation as a means of noise injection, where the inputs to the deep learning model are randomly transformed through various methods, such as rotating, scaling. We demonstrate that these methods are also effective in quantifying model uncertainty, and propose methods for diagnosing potential misclassifications.

Before doing so, we address the mathematics of deep learning in Chapter 2. We formally identify what machine learning is, and the different sets of data we make use of. We step through the details of how we train models in machine learning, and cast this framework in the setting of deep learning. The structure of the models used in deep learning, known as neural networks are looked into, followed by a detailed description of how deep learning predictions are made in a process known as forward propagation. We then look into ways of measuring the accuracy of these predictions, and how we *tune* models to improve their performance by stochastic gradient descent. In doing so, we cover a key step that must occur known as back propagation.

Chapter 2

Mathematical Background

To adequately describe what deep learning is and how it works, we first look at the more general process of **machine learning**. In the following sections, the main goals of machine learning are identified, as well as the structure of machine learning algorithms. In doing so, we define what a variable is and their different types. The notion of training and test data is presented, and we explore how machine learning models train, and common challenges that need to be overcome.

We then move onto deep learning, looking at the structure of a simple neural network, and building the foundations for the mathematics and algorithms underpinning deep learning.

2.1 Machine Learning

Machine learning is the process of developing models that can explain relationships between data, or make predictions from data, using algorithms designed for pattern recognition. Consider the problem of determining causes for heart disease. This type of problem is addressed with an explanatory model as we are trying to explain a relationship.

If we instead wanted a model that would be able to determine whether a particular person may have heart disease, we would use a predictive model as we are trying to predict something.

In both cases, we need to look at certain qualities in people that may be indicative of heart disease, such as their diet, age, and sex. In certain combinations, these may make an individual more or less likely to have heart disease. These qualities are commonly known as **variables**, as they are subject to change. Usually, variables that we use to identify others (such as diet, age, and sex in this case) are called **inputs**, **predictors**, or **features**. The variables that we are trying to predict or explain (in this case heart disease status) are known as **outputs**, **response variables**, or **labels**.

There are multiple areas of machine learning. The area to be explored in this thesis is known as **supervised learning**. In supervised learning, machine learning models are given a set of labelled training data consisting of input-output pairs, similar to our heart disease example. That is, we have a collection of both predictor variables and associated response variables. We call this training data a **training set**, as we directly train the model using this set of variables.

Machine learning models may be described by complicated functions that map predictor variables to predictions of what the underlying response may be. We say that a model is **parametric** if there is a family of function characterised by **parameters**, governing the way in which predictions are made. This family of functions acts as a template for the model, providing a vast range of functions that can be tuned to find the desired results by finding the right parameters. These parameters are typically denoted $\boldsymbol{\theta}$ and live inside a space of possible parameters Θ.

A typical training process in machine learning involves the model starting with randomly initialised parameters, making improvements to them until accurate results are found. To do so, the model makes predictions on the training set, then evaluates its performance by comparing the predictions to the true labels. Following is an update of the parameters in such a way that further improves performance on the training set, using optimisation techniques. In some models, such as linear regression, training is simple and occurs in a single step. Other models, such as those used in deep learning, do not instantly find the best parameter. Instead, they go through an iterative training process.

While models are training, it is often helpful to see their progress after each training iteration, to confirm progress is being made. Instead of using the training set to do so, we make use of another set known as the **validation set**. The validation set should come from the same distribution as the training set, but consist of new data. The validation set is useful as it tracks the performance of the model on data that it has not directly been trained on, providing an estimation of the **generalisability** of the model, how well the model may perform on data that it has not been trained using.

A key struggle in machine learning is developing powerful models without having them memorising the training data. More formally, this is a struggle between **underfitting** and **overfitting**. We say that model's whose performance on the training set closely resembles the performance on the validation set are **underfit** models. These models may have poor performance but are generalisable to unseen data, meaning that underfit model's performance on the training set is a good indication of what it would be on unseen data - they have not started to memorise the training data. Underfit models may benefit from further training to increase their performance.

Contrastingly, models that perform particularly well on the training set but have deteriorated performance on the validation set are known to be **overfit** models. They may have learned insignificant patterns in the training data that are not present in the overall distribution that the data is drawn from, or perhaps even memorised the mapping from the predictors in the training set to their respective labels. Overfit models are not generalisable to unseen data; we may naively look to the training set performance to get a misguided view that an overfit model is better than it truly is.

Finding a balance between underfitting and overfitting is a central challenge to machine learning. To build a powerful model, we need excellent performance on the training set, but more importantly, we need excellent performance on all possible data coming from the distribution we are interested in — this why we use the validation set. As the model never directly trains on the validation set, it represents unseen data, and so the performance of the model on the validation set indicates model performance on unseen data. Seeking a model that maximises its performance on the validation set yields a generalisable model. If we notice that a models performance has stopped improving on the validation set, we should stop training to avoid overfitting.

At this point, we need to consider if our model is sufficiently powerful. More often than not, we try different strategies, changing certain **hyper-parameters** (parameters not changed in training) of a particular model, or trying a different model altogether. From all models trained, we select that with performed best on the validation set, even if this model does not perform as well on the training set.

We must be careful in doing so, however, as models may become overfit to the validation set. Overfitting models to the validation set may seem counter-intuitive as the model never directly trains on the validation set; we choose a model that performs best on the validation set. The use of this selection process; however, coupled with excessively measuring the performance on the validation set facilitates this process of overfitting to the validation set.

The remaining set of data is known as the **test set**. This set is unused in both the training and validation steps. Instead, when we believe we have a finished model that has the best performance on the validation set, we evaluate the performance of the model on the test set. The performance of the model on the test set gives us an unbiased estimation of how we expect the model will perform on unseen data. For a well-trained model, we hope to see the performance on the test set closely resembling the performance on the validation set. If this is not the case, and the model performs particularly worse on the test set than the validation set, then it may be the case that the model has indeed overfit to the validation set. It is essential to use the test set sparingly so that it represents truly unseen data.

The quality of the predictions made on training data arising from the use of a particular parameter value can be assessed by comparing predictions $\hat{Y}$ and true labels Y under some metric, or **loss function**, denoted $\mathcal{L}$. Loss functions will be covered in more detail in Section 2.8, for now, they can be thought of as functions that measure how accurate predictions are by having a low value when predictions are accurate, and high value when they are not. The overall performance for the given parameter can be evaluated by considering the total loss, which is the sum of the loss for each training point. To help prevent overfitting, we sometimes combine this with a penalty term, that pushes parameters of the model towards that which result in a model with lower complexity. We denote this overall performance as the **cost**, using a cost function J, where $J : \Theta \to \mathbb{R}$, maps parameters to real numbers. Machine learning algorithms provide computationally efficient ways of searching through Θ to find values of $\boldsymbol{\theta}$ that aim to minimise $J(\boldsymbol{\theta})$.

In an ideal scenario, we would know the distribution that the data is drawn from. This distribution is referred to as the **data-generating distribution**, and denoted p_{data}. Knowing p_{data} would allow us to minimise the generalisation error, known as the **risk**. This quantity is the expected cost of the model over the entire distribution the data comes from. Letting $\left[\hat{Y}\right]_i = \hat{\mathbf{y}}_i = f(\mathbf{x}_i, \boldsymbol{\theta})$ be the model prediction on the input $\mathbf{x}_i$ using parameters $\boldsymbol{\theta}$, the **risk** is,

$$J(\boldsymbol{\theta}) = E_{p_{data}}\left[\mathcal{L}(Y, \hat{Y})\right],$$

Minimising this quantity provides us with a model that is highly generalisable to unseen data, as it minimises the cost over the whole distribution.

Since we typically have access to a subset of data from the data-generating distribution, we train our model to find parameters that minimise the **empirical risk**. The empirical risk is given as the expectation of the cost over the training set, otherwise known as the **emperical distribution**. Developing a model with sufficiently small cost on this distribution will hopefully provide a model which performs well on unseen data. However, having a model that performs well on a training set is not necessarily an indication that it will always perform well on unseen data. To ensure that we develop a generalisable model, we validate and test our model on data that the model has not been trained on before deploying it.

A typical machine learning training process is summarised in the flow chart Figure 2.1.

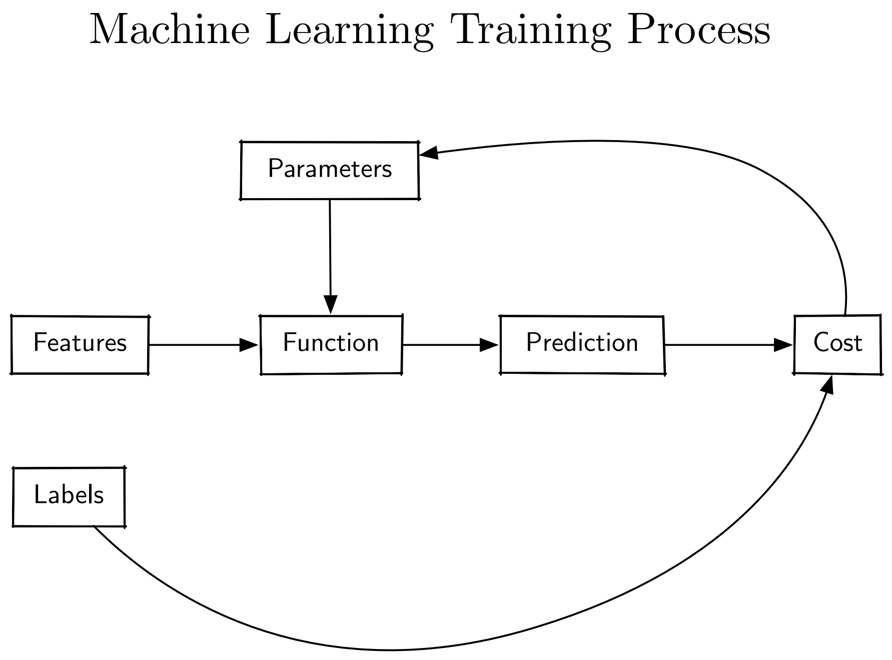


Figure 2.1: The training process in machine learning. Parameters and features are fed into a function to make a prediction on the response variables. The accuracies of these predictions are measured by comparing the predictions to the labels using a cost function. This cost is then used to update the parameters in a way that reduces the cost function.

2.2 Neural Networks

Deep learning is a sub-field of machine learning, loosely inspired by how the brain functions. The models that are built using deep learning are known as neural networks. In the following section, we summarise work from Goodfellow et al. [2017], and explore the most straightforward kind of neural networks, known as **multilayer perceptron networks**, or **deep feedforward networks**.

2.3 Multilayer Perceptron

A multilayer perceptron (MLP), or deep feedforward network (example in Figure 2.2) is a class of deep learning networks that has three types of **layers**: an **input layer**, **hidden layers**, and an **output layer**. Data propagates forward through the network, starting at the input layer, making its way through the hidden layers, and finally through the output layer. Each layer consist of multiple **nodes**, or **neurons**. With the exception of the input layer, data flows into these neurons as an affine transformation of previous layers outputs, or **activations**, or in other words a weighted sum of the previous layer's activations, with an additional term known as a bias term. Each neuron then applies a non-linear activation function to the affine transformation, which is then output as an activation to the next layer. In the case of the input layer, its activations are simply the inputs themselves. The weights and biases form the parameters $\boldsymbol{\theta}$ of the model.

Each node in the network may be responsible for learning a type of pattern in the input, that helps in making a prediction. Early layers may be responsible for learning low-level patterns, such as recognising edges in an image. Deeper layers may be able to learn much more abstract representations by combining the patterns recognised in early layers.

For the model to make useful predictions, the parameters of the model must be tuned. Parameter are tuned by first performing a process known as **forward propagation**, which first evaluates the model predictions, then the cost of those predictions. A process known as **back propagation** is then conducted which calculates the gradient of the cost function with respect to the parameters. This gradient is then used in a **gradient descent** method to minimise the cost function. The details of these processes are in the proceeding sections, summarised in Figure 2.3.

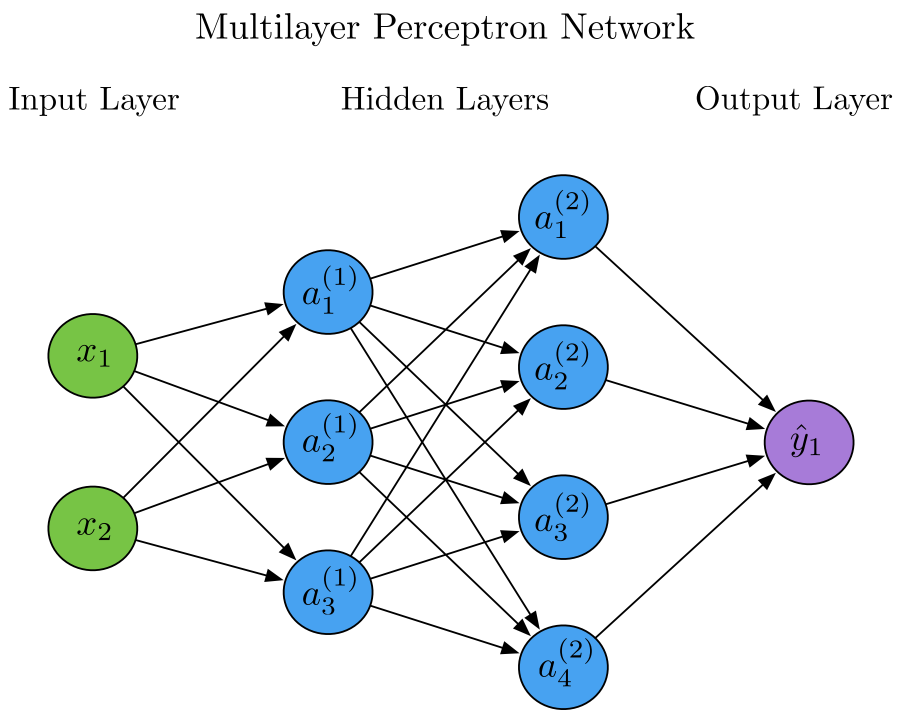


Figure 2.2: Example multilayer perceptron network with two hidden layers that maps two features x_1, x_2 to a single prediction $\hat{y}_1$. Each node computes an activation $a_j^{(k)}$ for the k^{th} node in the j^{th} hidden layer, that is sent to nodes in the next layer.

2.4 Forward Propagation

Forward propagation is the process of making predictions on the response variables using features. When we are training the model, we also measure how accurate our predictions are using a cost function. The details of this process are described in this Section.

We first develop the forward propagation algorithm in the simple case where the input layer takes a column vector of features $\mathbf{x}$, corresponding to individual features $x_1, \cdots, x_n$. These are the activations of the input layer, which we denote as the 0^{th} layer. To provide a general framework to work with, we may define the activations of the k^{th} layer with n_k nodes as $\mathbf{a^{(k)}} = (a_1^{(k)}, \cdots, a_{n_k}^{(k)})^T$. For every node in the k^{th} layer, with $k \geq 1$ we take the weighted sum of activations $a_i^{(k-1)}$ of the previous layer with weights $w_{i,j}^{(k)}$, as well as a bias $b_j^{(k)}$. We say that this affine transformation

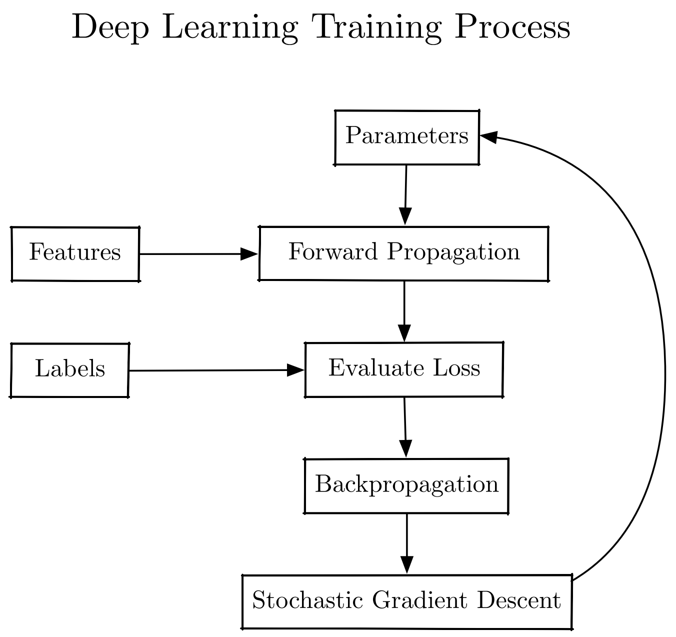


Figure 2.3: Flow chart describing the iterative deep learning training process. Forward propagation is conducted to make predictions on the response variables corresponding to features, using parameters. These predictions are compared to the labels using a cost function. Backpropagation finds the gradient of the cost function to be used in stochastic gradient descent to update the parameters to reduce the cost function.

is the **weighted input** $h_j^{(k)}$ to the j^{th} node in the k^{th} layer, calculated as:

$$h_j^{(k)} = \sum_{i=1}^{n_k} w_{i,j}^{(k)} \cdot a_i^{(k-1)} + b_j^{(k)},$$

and can be written more compactly in a vector form as

$$h_j^{(k)} = \mathbf{w_j^{(k)}}^T \mathbf{a^{(k-1)}} + b_j^{(k)}.$$

A non-linear activation function $f^{(k)}$ is applied to $h_j^{(k)}$ for each node in the layer to obtain the activations $a_j^{(k)}$ for layer k.

$$a_j^{(k)} = f^{(k)}(h_j^{(k)}).$$

The information propagates forward through the network in this fashion, reaching the output layer ℓ, where the outputs gives the model's prediction $\hat{\mathbf{y}}$ of the true label $\mathbf{y}$. The details of forward propagation for a single node are depicted in Figure 2.4.

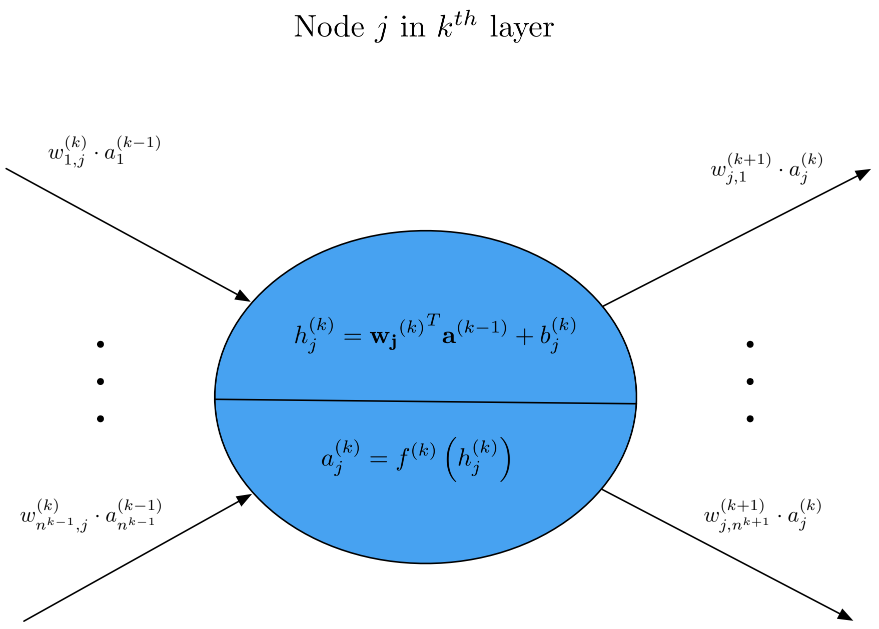

Node j in k^{th} layer
$w_{1,j}^{(k)} \cdot a_1^{(k-1)}$
$w_{j,1}^{(k+1)} \cdot a_j^{(k)}$
$h_j^{(k)} = \mathbf{w_j}^{(k)^T} \mathbf{a}^{(k-1)} + b_j^{(k)}$
$a_j^{(k)} = f^{(k)}\left(h_j^{(k)}\right)$
$w_{n^{k-1},j}^{(k)} \cdot a_{n^{k-1}}^{(k-1)}$
$w_{j,n^{k+1}}^{(k+1)} \cdot a_j^{(k)}$

Figure 2.4: Visualisation of the j^{th} node in the k^{th} layer, computing its weighted input $h_j^{(k)}$ as an affine transformation of previous activations, before computing its own activation $a_j^{(k)}$ that is sent as a weighted input to nodes in the next layer.

This process of forward propagation gives predictions, but it is a seemingly abstract process, and it is not necessarily clear why it is so powerful. A brief intuition that provides a reasoning for this is now given. A theoretical justification makes use of the Universal Approximation Theorem [Cybenko, 1989], which is out of the scope of this thesis.

Each node in a neural network can be thought of as being responsible for detecting specific patterns. Nodes will have large activations if the patterns are present and small or negative otherwise.

Consider a single node, the weights of into this node can be thought of as parameters that describe how significant the respective nodes in the previous layer are in making up the patterns that this node is concerned with. We can also think of the bias as imposing a threshold on the activation of the node, making it more or less sensitive to the pattern.

As an example, consider the activations in the layer immediately before the output layer of a neural network that classifies pictures of cats and dogs, where the response variable is a one if a cat is present and zero if a dog is present. There may be nodes in this layer that have large activation for particular features of cats and dogs, such as pointy ears, floppy ears, long whiskers, and long nose, amongst others. When an image of a cat is given to the neural network, the pointy ears node, and long whiskers node should have strong activations, as these patterns are present. To have the neural network classify the images as a cat, we use large positive weights from the pointy ear node and long whiskers node to the output layer node. Conversely, if a picture of a dog is given the floppy ears node and long nose nodes should have large activations, meaning that we want large negative weights from these nodes to the output layer node to classify the picture with predicted label zero, a dog.

This example is an idealisation of how neural networks work; they will not necessarily look for patterns familiar to humans. Instead, patterns that neural networks make use of can be much more abstract. However, this interpretation provides useful intuition of how basic patterns can be used to learn patterns with higher levels of abstraction.

Returning to forward propagation, we can compact the process down into an equivalent vectorised version by using weight matrices;

$$W^{(k)} = [\mathbf{w_1^{(k)}}, \cdots, \mathbf{w_{n_k}^{(k)}}]^T,$$

and bias vectors;

$$\mathbf{b^{(k)}} = (b_1^{(k)}, \cdots, b_{n_k}^{(k)})^T.$$

It is then possible to compute the weighted input of all nodes in the k^{th} layer as;

$$\begin{aligned}\mathbf{h^{(k)}} &= (h_1^{(k)}, \cdots, h_{n_k}^{(k)})^T \\ &= W^{(k)}\mathbf{a^{(k)}} + \mathbf{b^{(k)}}.\end{aligned}$$

This allows us to succinctly express the activations of the k^{th} layer as;

$$\begin{aligned}\mathbf{a^{(k)}} &= f^{(k)}(W^{(k)}\mathbf{a^{(k-1)}} + \mathbf{b^{(k)}}) \\ &= f^{(k)}(\mathbf{h^{(k)}}).\end{aligned}$$

Here $f^{(k)}$ is the activation function of the k^{th} layer, acting element-wise.

Until this point, we have made a prediction $\hat{\mathbf{y}}$ for a single set of features $\mathbf{x}$. To exploit the computational benefits gained from parallel computing, we can generalise this model further to compute the matrix

$$\hat{Y} = [\hat{\mathbf{y}}^{[1]}, \cdots, \hat{\mathbf{y}}^{[m]}]$$

where m is the number of training examples in our data-set and the upper right subscript within square brackets indicates the index of the training example. For example; $\hat{\mathbf{y}}^{[t]}$ is the models prediction for the label $\mathbf{y}^{[t]}$ of the t^{th} training example $\mathbf{x}^{[t]}$. We also similarly define the model's activations;

$$A^{(k)} = [\mathbf{a^{(k)[1]}}, \cdots, \mathbf{a^{(k)[m]}}].$$

Writing the bias vectors as a matrix of m repeated columns;

$$B^{(k)} = [\mathbf{b^{(k)}}, \cdots, \mathbf{b^{(k)}}].$$

We can compute the matrix of model activations for layer k as follows,

$$H^{(k)} = W^{(k)} A^{(k-1)} + B^{(k)},$$

$$A^{(k)} = f^{(k)}(H^{(k)}).$$

In this case $f^{(k)}$ acts element-wise across the matrix so that $A^{(k)}$ is the same dimension as $H^{(k)}$. This allows us to compute the activations across all neurons in the k^{th} layer for all training examples in one step.

This forward propagation technique is an iterative process, calculating the activations of successive layers;

$$\begin{aligned}
\hat{Y} =& f^{(\ell)}\left(H^{(\ell)}\right) \\
=& f^{(\ell)}\left(W^{(l)} f^{(\ell-1)}\left(H^{(\ell-1)}\right) + B^{(\ell)}\right) \\
&\vdots \\
=& f^{(\ell)}\left(W^{(\ell)} f^{(\ell-1)}\left(\cdots\left(W^{(2)} f^{(1)}\left(H^{(1)}\right) + B^{(2)}\right)\cdots + B^{(\ell-1)}\right) + B^{(\ell)}\right) \\
=& f^{(\ell)}\left(W^{(\ell)} f^{(\ell-1)}\left(\cdots\left(W^{(2)} f^{(1)}\left(W^{(1)} X + B^{(1)}\right) + B^{(2)}\right)\cdots + B^{(\ell-1)}\right) + B^{(\ell)}\right).
\end{aligned}$$

We can express the underlying function of the neural networks as $f(\mathbf{x}, \boldsymbol{\theta})$, that maps inputs to predictions

$$\hat{\mathbf{y}} = f(\mathbf{x}, \boldsymbol{\theta}),$$

Or more generally, mapping the matrix of features to the matrix of predictions;

$$\hat{Y} = f(X, \boldsymbol{\theta}).$$

After making predictions $\hat{Y}$, the final step in forward propagation is to evaluate the cost function. Here, we define the cost function $J(\boldsymbol{\theta})$ to be the empirical risk of the network. This is the average loss over each example in our training set. We may also choose to add a regularisation term to this cost function in order to prevent the model from overfitting. A regularisation term helps to constrain the parameter space by penalising parameters that lie far from the origin. By effectively constraining the parameter space, we reduce the set of possible parameters the model may learn, reducing the **complexity** of the model. Typically, we denote this regularisation term by $\Omega(\boldsymbol{\theta})$. The calculation of the cost function is then seen to be:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{training}\big[\mathcal{L}(f(X,\boldsymbol{\theta}),Y)\big] + \lambda\Omega(\boldsymbol{\theta}) \\ &= \mathbb{E}_{training}\big[\mathcal{L}(\hat{Y},Y)\big] + \lambda\Omega(\boldsymbol{\theta}) \\ &= \frac{1}{m}\sum_{t=1}^{m}\mathcal{L}(\hat{\mathbf{y}}^{[t]},\mathbf{y}^{[t]}) + \lambda\Omega(\boldsymbol{\theta}), \end{aligned}$$

where λ is some constant.

How we measure the performance of the model is dependent on the choice of the loss function, which usually depends to the type of problem we are dealing with, and is explored in more detail in Section 2.8. Overfitting and regularisation techniques are explored in Section 2.9.

2.5 Gradient Descent

For a model to make good predictions that are close to the response variables, we tune the parameters $\boldsymbol{\theta}$ of the model according to how they affect the model's predictions via the cost $J(\boldsymbol{\theta})$. Models with a lower cost make more accurate predictions on the training set. We hope that this result is generalisable to other data, and hence seek,

$$\boldsymbol{\theta}^* := \arg\min_{\boldsymbol{\theta}}\{J(\boldsymbol{\theta}) \mid \boldsymbol{\theta} \in \Theta\},$$

where Θ is the parameter space of $\boldsymbol{\theta}$.

One such way of tuning $\boldsymbol{\theta}$ is by **gradient descent**. Gradient descent takes the simple optimisation strategy of adjusting $\boldsymbol{\theta}$ to be $\boldsymbol{\theta} - \varepsilon\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ where ε is a small constant known as the learning rate. By taking the gradient of the cost with respect to $\boldsymbol{\theta}$, we find the direction of steepest ascent in the cost with respect $\boldsymbol{\theta}$. Taking the negative of this gradient gives the direction of steepest descent, giving us a direction gives of how we might make a small update to $\boldsymbol{\theta}$ to give a reduced cost in the model. For highly non-linear models this is problematic as higher-order components might influence the cost when moving a whole step of size ε if ε is not sufficiently small. However, if ε too small, the training process may become too computationally expensive, as many iterations of gradient descent are needed. Using mostly-linear activation functions in the model together with small ε ensures that higher-order terms do not strongly influence this update.

A single step of gradient descent, otherwise known as **batch gradient descent** involves the assignment $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ where $J(\boldsymbol{\theta})$ is calculated from the entire training set. This gives the true gradient of the cost function, however is computationally expensive.

2.5.1 Stochastic Gradient Descent

One way of reducing the computational complexity of gradient descent, while following a similar principle is by using a more general method named **minibatch stochastic gradient descent**, otherwise known as **stochastic gradient descent**. In this process, we take the cost of the model as the cost of random sample (without replacement) of the training set $\mathbb{B} = \{\mathbf{x}^{[s_1]}, \cdots, \mathbf{x}^{[s_{m'}]}\}$, where s_i are the indices of the sample in the training set. We call this subset $\mathbb{B}$ a **minibatch**, where m' is the minibatch size, typically chosen for powers of 2, from 1 to 256, using $m' = 1$ corresponds to true SGD. Smaller batch sizes used in stochastic gradient descent provide a significantly less expensive optimisation strategy than batch gradient descent as they use a smaller subset of data to compute the gradient, for the price of providing only an unbiased estimate of the true gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, instead of the true gradient as in batch gradient descent.

As well as reducing the computational complexity of finding the gradient, this estimation of the gradient may also regulatory effect on the model, adding noise that helps to prevent the model from learning insignificant patterns which if learned may cause the model to overfit [Goodfellow et al., 2017, Section 8.1.3].

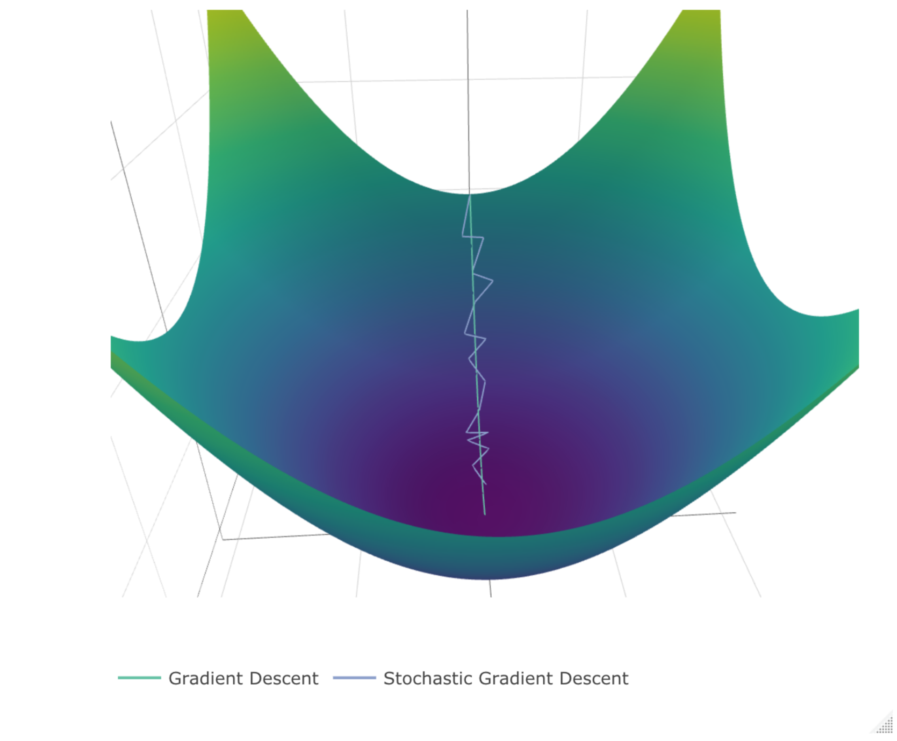


Figure 2.5: Comparison between stochastic gradient descent and gradient descent. Notice that gradient descent takes steps aligning with the true gradient, whereas stochastic gradient descent takes steps in a random but similar direction to the true gradient.

2.6 Backpropagation

Section 2.5 shows a training strategy that improves the performance of the model using stochastic gradient descent. To perform stochastic gradient descent, we must find a way to compute the gradient of the cost function with respect to the parameters $\nabla_{\boldsymbol{\theta}} J$. To find $\nabla_{\boldsymbol{\theta}} J$, we utilise a process known as backpropagation is implemented. Note that we only need to compute the cost function of $\boldsymbol{\theta}$ on a subset of the training data; however, for notational brevity, we will demonstrate this result for the entire training set.

The backpropagation algorithm, or commonly **backprop**, computes $\nabla_{\boldsymbol{\theta}} J$ by recursively applying the chain rule to obtain partial derivatives of the cost function with respect to the activations and weighted inputs of preceding layers.

Recall the cost of a neural network is given by

$$J(\boldsymbol{\theta}) = \mathbb{E}_{training}\big[\mathcal{L}(f(\mathbf{x},\boldsymbol{\theta}),\mathbf{y})\big] + \lambda\Omega(\boldsymbol{\theta})$$

The gradient of the cost decomposes across its terms;

$$\nabla_{\boldsymbol{\theta}} J = \nabla_{\boldsymbol{\theta}}\mathbb{E}_{training}\big[\mathcal{L}(f(\mathbf{x},\boldsymbol{\theta}),\mathbf{y})\big] + \nabla_{\boldsymbol{\theta}}\lambda\Omega(\boldsymbol{\theta})$$

Since the expectation is taken over a finite training set, the gradient can be taken inside of the expectation:

$$\nabla_{\boldsymbol{\theta}} J = \mathbb{E}_{training}\big[\nabla_{\boldsymbol{\theta}}\mathcal{L}(f(\mathbf{x},\boldsymbol{\theta}),\mathbf{y})\big] + \nabla_{\boldsymbol{\theta}}\lambda\Omega(\boldsymbol{\theta}).$$

Depending on the choice of regularisation term $\Omega(\boldsymbol{\theta})$, it is relatively simple to explicitly calculate $\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta})$ as it is a function of the parameters and not the activations. This will be shown in Section 2.9 where we define appropriate regularisation terms. The loss however, is given as the composition of many functions, where the parameters of the model are buried within these compositions. To then find $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\hat{\mathbf{y}},\mathbf{y})$, we must recursively apply the chain rule to unravel all of the compositions.

We recall the multivaraible chain rule: if $\mathbf{y} \in \mathbb{R}^n$, and $\mathbf{x} \in \mathbb{R}^m$ with functions $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^m \to \mathbb{R}^n$, with $\mathbf{y} = g(\mathbf{x})$ then the chain rule states:

$$\nabla_{\mathbf{x}} f = \left(\frac{\partial g}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} f,$$

where $\left(\dfrac{\partial g}{\partial \mathbf{x}}\right)$ is the Jacobian of the function g.

The process of determining $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\hat{\mathbf{y}},\mathbf{y})$ is now illustrated. For brevity, we use the notation $\mathcal{L} := \mathcal{L}(\hat{\mathbf{y}},\mathbf{y})$ and consider $\hat{\mathbf{y}}$ to be the activation of the ℓ^{th} layer, *i.e.* $\mathbf{a}^{(\ell)} = \hat{\mathbf{y}}$.

The gradient of the loss $\mathcal{L}$ with respect to the prediction $\hat{\mathbf{y}}$, $\nabla_{\hat{\mathbf{y}}}\mathcal{L} = \nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}$, is first computed directly. This is dependent on the choice of loss function (Section 2.8), but can be explicitly calculated using basic calculus.

Recursively, the chain rule is applied to obtain the gradient of the loss with respect to the weighted input and activations preceding layers. This propagates backward

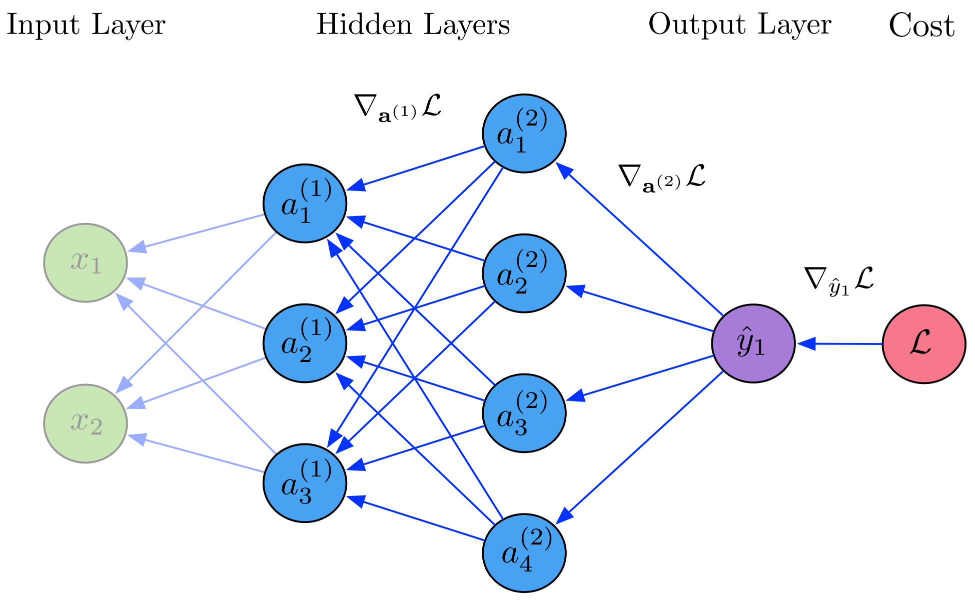


Figure 2.6: Backpropagation for the network. Within each layer the gradient of the activation with respect to the parameters is calculated by using the chain rule to backpropagate.

through the layers, starting with layer $k = \ell$, down to $k = 1$, as illustrated in Figure 2.6

$$\nabla_{\mathbf{h}^{(k)}}\mathcal{L} = \left(\frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{h}^{(k)}}\right)^T \nabla_{\mathbf{a}^{(k)}}\mathcal{L},$$

$$\nabla_{\mathbf{a}^{(k-1)}}\mathcal{L} = \left(\frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{a}^{(k-1)}}\right)^T \nabla_{\mathbf{h}^{(k)}}\mathcal{L}.$$

It then becomes possible to calculate $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ by computing each of the partial derivatives $\nabla_{W^{(k)}}\mathcal{L}$ and $\nabla_{\mathbf{b}^{(\ell)}}\mathcal{L}$ for $k \in \{1, ..., \ell\}$, as $\boldsymbol{\theta}$ is just a collection of all of the weights and biases.

$$\nabla_{W^{(\ell)}}\mathcal{L} = \left(\frac{\partial \mathbf{h}^{(\ell)}}{\partial W^{(\ell)}}\right)^T \nabla_{\mathbf{h}^{(\ell)}}\mathcal{L},$$

$$\nabla_{\mathbf{b}^{(\ell)}}\mathcal{L} = \left(\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}\right)^T \nabla_{\mathbf{h}^{(\ell)}}\mathcal{L}.$$

2.7 Activation Functions

2.7.1 Hidden Layer Activations

We have seen the basic structure of a deep feedforward network, where an input undergoes many transformations as it progresses through the layers of the network. For these transformations to be meaningful, we must not consider solely affine transformations in each layer, as the whole network could then be reduced down to a single affine transformation since the composition of affine functions yields an affine function. For this reason, we make use of non-linear activation functions in each node. Doing so allows us to build up a complicated non-linear function, capable of making accurate predictions.

There are certain properties desirable of activation functions, helping to guide which should be used. For gradient descent to be effective at finding a minimum, it becomes beneficial to use activation functions that are not highly non-linear. Using highly non-linear activations would cause higher order derivatives to have a strong influence in small neighbourhoods, mitigating the effectiveness of gradient descent, which is based on only the first derivative. Figure 2.7 shows common choices for activations, as described below.

Traditionally, the **logistic sigmoid** function, which squashes the real number line onto the interval $[0, 1]$, was a common choice for hidden layer activations. The sigmoid function is defined as;

$$\sigma(x) := \frac{1}{1+e^{-x}}.$$

A closely related function is the **hyperbolic tangent** function;

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

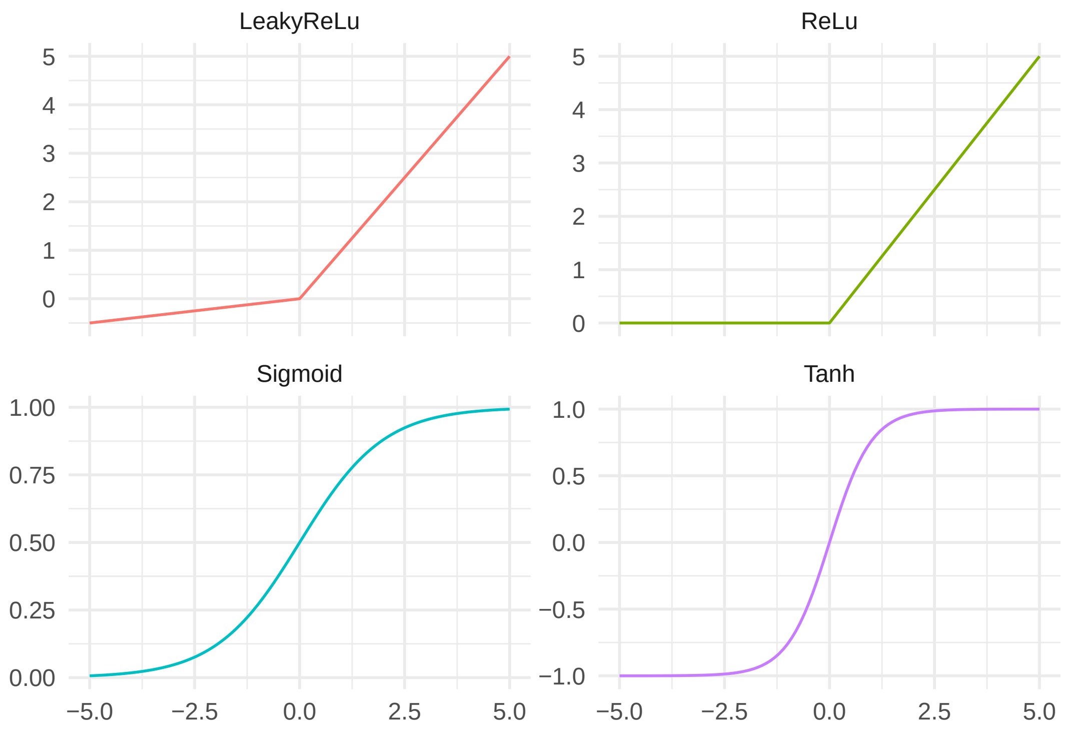


Figure 2.7: Typical choices for hidden layer activation functions.

Simple algebra shows that the hyperbolic tangent is a translated and scaled sigmoid function:

$$\begin{aligned}
\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
&= \frac{2e^x - e^{-x} - e^x}{e^x + e^{-x}} \\
&= \frac{2e^x}{e^x + e^{-x}} - 1 \\
&= \frac{2}{1 + e^{-2x}} - 1 \\
&= 2\sigma(2x) - 1.
\end{aligned}$$

This transformation benefits the hyperbolic tangent as an activation function as it centres it at zero, as well as having near unit first derivative so that it closely resembles the identity function near zero. As such, this transformation improves the training of the neural network [Goodfellow et al., 2017, Section 6.3.2]. However, this is out of the scope of this thesis.

A downside of using sigmoid or hyperbolic tangent functions is that they have near-zero slope when their input is far from the origin, which we refer to as becoming **saturated**. When this is the case, it is particularly difficult for gradient descent to update parameters to give a reduced cost, as adjusting the parameters by a step in the negative direction of the gradient has almost no effect. This saturation motivates the use of the **rectifier**, presently, a popular choice for hidden layer activations [Goodfellow et al., 2017, Section 6.3.1]. The rectifier is defined to be the identity function on the positive reals, and 0 elsewhere;

$$f(x) := \max(0, x).$$

A neuron that uses a rectifier as its activation function is know as a **rectified linear unit**, or **ReLu**. While it may be concerning that the rectifier is not differentiable at zero and that gradient descent relies on finding derivatives, it has both left and right derivatives defined, and it is sufficient to use either. Goodfellow et al. [2017] justifies this use by considering that numerical precision of computers is imperfect, reasoning that tiny non-zero numbers are often numerically underflowed to zero.

Generalisations of the rectified linear unit exist, such as the **Leaky ReLu**, an activation function similar to ReLu, with the only difference being instead of mapping negative numbers to zero, it maps them to a small fraction of their value. Usually, ReLu is a sufficient choice. Through experimentation with different activation functions, small improvements may be made.

2.7.2 Output Layer Activations

The output layer activation function is the final function applied to data as it propagates through the neural network, so that its output gives the predictions, and so must be chosen carefully to ensure that the model makes predictions useful for classification or regression. In the case of binary classification, we can represent the labels with 0's and 1's. A useful model for binary classification output a probability that an input is of a particular class, for example $\hat{y} = P(y = 1|\mathbf{x})$. For such outputs to be valid probabilities, the output layer activation must map to $[0, 1]$. In this setting, predictions near 0 have a low probability of being in Class 1, and can accordingly be classified as Class 0. Predictions nearer to 1 can be classified as Class 1. The distance between the model prediction and class label is used in the cost function as an accuracy measure of the prediction. This is explored in Section 2.8. In particular, models performing binary classification are well suited to using the sigmoid function (previously discussed in Section 2.7.1) as an output layer activation, as this ensures that predictions will lie in $[0, 1]$.

For multi-class classification, we extend binary classification to give probabilities for each possible class. For a classification problem with n classes, the prediction could be represented by an n-dimensional vector, where the i^{th} element of the vector represents the probability that the input is of Class i,

$$\hat{\mathbf{y}} = \left(P\left(y = 1|\mathbf{x}\right), \ldots, P\left(y = n|\mathbf{x}\right)\right).$$

Ensuring this is a valid probability requires that each element of the vector is within the interval $[0, 1]$, with the additional requirement that the elements of $\hat{\mathbf{y}}$ sum to unity. A generalisation of the logistic sigmoid function known as the **softmax** function has such behaviour, defined with i^{th} element as;

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{i=1}^{n} z_i},$$

for $i = 1, \ldots, n$, given an input $\mathbf{z} = (z_1, \ldots, z_n)$ with $z_i \in \mathbb{R}$.

Regression models may have response variables in $\mathbb{R}$. Therefore, a suitable output layer activation does not need the same restrictions that those for classification tasks do. In fact, the identity function is a suitable choice for regression, see [Goodfellow et al., 2017, Section 6.2] for further details.

2.8 Cost Functions

In order to improve a model's performance, we need a practical way of measuring performance. Typically model performance encapsulates the accuracy of predictions, as well as the complexity of the model. Penalising overly complex models helps to prevent overfitting (Section 2.1). Model performance is measured using a **cost function**. A cost function takes model predictions based on a given parametrisation of the model, combined with true labels to find how 'far' the predictions are from the truth. To measure how 'far', we can use different measures of dissimilarity, with the general idea that predictions close to the truth should incur a small cost, whereas those far from the truth should have a high cost.

A cost function usually includes a penalty term $\Omega(\boldsymbol{\theta})$, to aid in the prevention of overfitting by penalising the complexity of models through their parameters. We explore this strategy in Section 2.9.

A useful tool for choosing the 'best' parameter of a model is **maximum likelihood estimation**. The **likelihood** of a model is the probability of observing the training data labels Y, given training data X and parameter $\boldsymbol{\theta}$. Suppose that the training labels are distributed according to some distribution. We define the likelihood as,

$$L(\boldsymbol{\theta}\,; Y) = p(Y|X, \boldsymbol{\theta}).$$

The goal of maximum likelihood estimation is to find a parameter $\boldsymbol{\theta_{ML}}$ that maximises the likelihood $L(\boldsymbol{\theta}; Y)$. The idea being that the parameter that most likely generated our observations is the best.

We denote this parameter as the maximum likelihood estimate.

It is often preferred to work with the **log-likelihood**, defined as

$$\ell\,(\boldsymbol{\theta}; Y) = \log\, L(\boldsymbol{\theta}\,; Y).$$

The log-likelihood resolves the issue of numerical underflow that the likelihood is prone to due to repeated multiplication of small numbers, by replacing products with sums. Since likelihoods are non-negative, and the logarithm is monotonically increasing on the non-negative reals, the argument that maximises the likelihood is the same that maximises the log-likelihood.

A common metric used for a loss function in regression problems is known as the **mean-squared error**, defined for a model $f(X, \boldsymbol{\theta})$ as;

$$\begin{aligned} MSE(\boldsymbol{\theta}) &= ||\mathbf{Y} - f(X, \boldsymbol{\theta})||_2^2 \\ &= ||\mathbf{Y} - \hat{\mathbf{Y}}||_2^2. \end{aligned}$$

It can be shown that the minimiser of the mean squared error yields equivalent results to maximum likelihood estimation for linear regression under the assumption $\mathbf{y} \sim \mathrm{N}(X\beta, \Sigma)$ with $\Sigma = \mathrm{diag}(\sigma^2)$, and σ^2 known. A short proof of this follows. The likelihood of observing $\mathbf{y}$ under the parameterisation β, defined using the probability density function for a normal distribution can be seen to be;

$$L(\beta; \mathbf{y}) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} \exp\left(\frac{-||\mathbf{y} - X\beta)||_2^2}{2\sigma^2}\right).$$

As we are assuming σ^2 known, the maximum likelihood estimate β_{ML} is;

$$\beta_{ML} = \underset{\beta}{\mathrm{argmax}} \left[\exp\left(\frac{-||\mathbf{y} - X\beta)||_2^2}{2\sigma^2}\right)\right].$$

Since the exponential function is monotonically increasing, the maximum occurs at the maximum to its argument;

$$\beta_{ML} = \underset{\beta}{\operatorname{argmax}} \left[\frac{-||\mathbf{y} - X\beta||_2^2}{2\sigma^2} \right].$$

Again, σ^2 is known, so the maximum likelihood estimate is;

$$\beta_{ML} = \underset{\beta}{\operatorname{argmin}} \; ||\mathbf{y} - X\beta||_2^2.$$

This shows the maximum likelihood estimate minimises the mean squared error with predictions $\hat{\mathbf{y}} = X\beta$.

In the setting of binary classification, **binary cross-entropy** is used as a loss function, defined for a prediction p of class label y as;

$$\text{CE} = -(y \cdot \log(p) + (1-y) \cdot \log(1-p))$$

Minimising a binary cross-entropy loss function can also been seen as equivalent to maximum likelihood estimation, which will now be shown. Consider an outcome following a Bernoulli distribution $y \sim \text{Bernoulli}\,(p)$. The likelihood is,

$$L(\theta\,; y) = \theta^y (1-\theta)^{1-y}.$$

Hence the log-likelihood is;

$$\begin{aligned} \ell(\theta; y) &= y \cdot \log(\theta) + (1-y)\log(1-\theta) \\ &= -\text{CE}. \end{aligned}$$

So minimising a binary cross entropy loss function is equivalent to maximum likelihood estimation for Bernoulli observations.

As the primary concern of this thesis is in binary classification, we omit the proof that the loss function used for multiclass classification gives maximum likelihood estimates.

2.9 Regularisation

As the complexity of a model increases, so too does its power to recognise patterns. Powerful models are able to be developed this way but at a cost. Increasing the complexity of a model makes it more prone to overfitting, thereby reducing the effectiveness of the model on unseen data. Finding the right model complexity is crucial to minimising generalisation error. Preventing model overfitting is done using a process known as **regularisation**.

The **bias** of an estimator T of θ denoted $\text{Bias}_T(\theta)$ as;

$$\text{Bias}_T(\theta) = \mathbb{E}(T) - \theta.$$

This relationship between model complexity and generalisation error is well described by the **bias-variance trade-off**, illustrated in Figure 2.8.

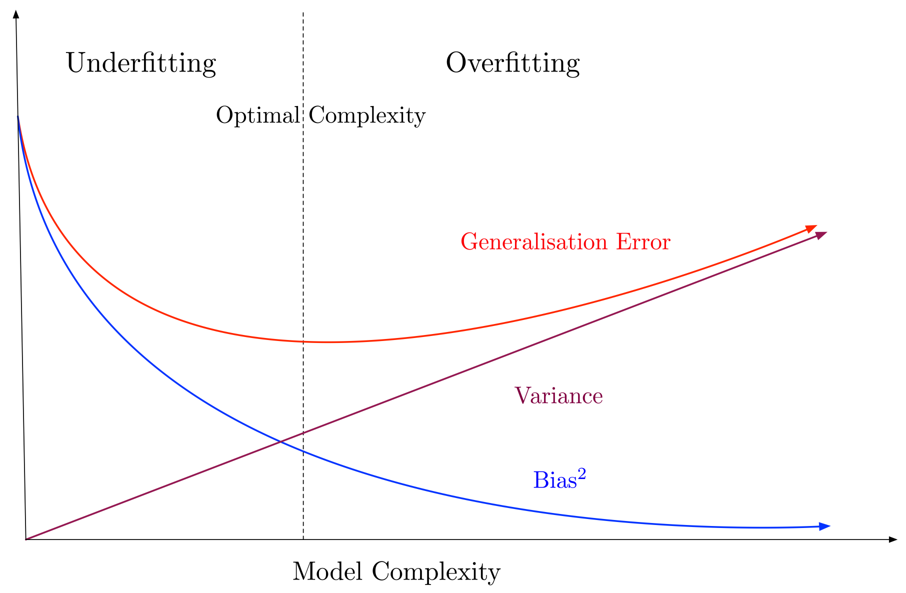


Figure 2.8: Bias-Variance trade off.

There are three contributing factors accounting for the error in a model's predictions. Consider the mean squared error of a prediction using an unseen feature-response pair (x_0, y_0).

The mean squared error is defined as;

$$\mathbb{E}\left[\left((y_0 - \hat{f}(x_0)\right)^2\right].$$

Here the expectation is over the possible choices for training sets, sampled from the data generating distribution. We can rewrite this as;

$$\mathbb{E}\left[\left(y_0 - \hat{f}(x_0)\right)^2\right] = \mathrm{Var}\left(y_0 - \hat{f}(x_0)\right) + \mathbb{E}\left[y_0 - \hat{f}(x_0)\right]^2 . \tag{2.1}$$

Noticing that $y_0 = f(x_0) + \varepsilon_0$, we see that the first term in Equation 2.1 becomes;

$$\mathrm{Var}\left(y_0 - \hat{f}(x_0)\right) = \mathrm{Var}\left(f(x_0) + \varepsilon_0 - \hat{f}(x_0)\right) .$$

Since $f(x_0)$ is a constant, this becomes;

$$\mathrm{Var}\left(y_0 - \hat{f}(x_0)\right) = \mathrm{Var}\left(\varepsilon_0 - \hat{f}(x_0)\right) .$$

As ε_0 is assumed to be independent from the training data X, and hence x_0 and $f(x_0)$,

$$\begin{aligned}\mathrm{Var}\left(y_0 - \hat{f}(x_0)\right) &= \mathrm{Var}\left(\varepsilon_0\right) + \mathrm{Var}\left(\hat{f}(x_0)\right) - 2 \cdot \mathrm{Cov}\left(\varepsilon_0, \hat{f}(x_0)\right) \\ &= \mathrm{Var}\left(\varepsilon_0\right) + \mathrm{Var}\left(\hat{f}(x_0)\right) . \end{aligned} \tag{2.2}$$

The term $\mathbb{E}\left[y_0 - \hat{f}(x_0)\right]^2$ in Equation 2.1 is the squared bias of the prediction:

$$\mathbb{E}\left[y_0 - \hat{f}(x_0)\right]^2 = \mathrm{Bias}\left(\hat{f}(x_0)\right)^2 . \tag{2.3}$$

Combining Equation 2.2 and Equation 2.3 allows us to rewrite Equation 2.1 as;

$$\begin{aligned}\mathbb{E}\left[\left(y_0 - \hat{f}(x_0)\right)^2\right] &= \mathrm{Var}\left(y_0 - \hat{f}(x_0)\right) + \mathbb{E}\left[y_0 - \hat{f}(x_0)\right]^2 \\ &= \mathrm{Var}\left(\varepsilon_0\right) + \mathrm{Var}\left(\hat{f}(x_0)\right) + \mathrm{Bias}\left(\hat{f}(x_0)\right)^2 .\end{aligned}$$

Note that $\mathrm{Var}\left(\varepsilon_0\right)$ is the unavoidable noise in the data, we cannot make improvements to our model to reduce this term. Instead, we try to minimise the error stemming from the two other terms, $\mathrm{Var}\left(\hat{f}(x_0)\right)$, and $\mathrm{Bias}\left(\hat{f}(x_0)\right)^2$.

A significant bias corresponds to a model that has poor performance on the training data; it is underfit and needs to undergo further training. Contrastingly, a model with a small bias term but with a large variance is an overfit model. While it would be ideal to minimise both the bias and the variance of the model, we are unable to do so effectively with limited data. Most forms of regularisation follow the principle of parsimony, wherein the simplest model that can adequately describe the data should be chosen.

Model simplicity is related to the number and range of the possible parameters the model makes use of. Previously, we used the term model **complexity**. Models with many parameters or those that can take parameters in a large space have larger **complexity** than those with fewer parameters, or that take parameters in a smaller space. We can reduce the bias by increasing model complexity; however, this leads to a higher variance.

Regularisation techniques can reduce the generalisation error by making useful trades between bias and variance through model complexity.

A simple way to reduce model complexity is by penalising model coefficients in order to constrain the parameter space. We do so by adding a multiple of a regularisation term to the cost function, usually denoted $\Omega(\boldsymbol{\theta})$. Common choices for these regularisation terms are the L^1 or L^2 norm.

A direct way to improve a model is to give it more data to train on. Usually, we have a fixed training data, and so this isn't always a possibility. However, in classification tasks, we can simulate real data by **augmenting** training data and maintaining the same labelling. That is, we can make small transformations to the training data by adding small amounts of noise that should not affect the labelling, to give us seemingly more data than we had when we started. These transformations are useful, especially in object recognition tasks. Typical image augmentation techniques such as random rotations, translations, and scaling have been all been shown to be effective [Goodfellow et al., 2017, Section 7.4].

2.9.1 Dropout

Developing models that are robust to noise, in general, is an effective regularisation technique, known as a **stochastic regularisation technique**, or SRT [Goodfellow et al., 2017, Section 7.5], [Khan et al., 2018]. Hinton et al. [2012] build upon this idea in a process that is known as known **dropout**, an effective and commonly implemented regularisation strategy. Hinton describes dropout as having a regulatory effect by preventing complex co-adaptations on training data, by multiplying activations of nodes in forward propagation by 0 with some probability p, known as the **dropout probability**. At test time, dropout is not run[1], instead, each activation from each node is reduced by a factor p corresponding to its dropout probability, attempting to match the strength of the activations in training time when nodes were subject to dropout.

Hinton likens dropout to a form of model averaging, in which an **ensemble** of models are trained, who each "vote" on a prediction by having their predictions averaged. Model averaging is a computationally expensive procedure, scaling linearly with the number of models trained, that can reduce the variance of predictions in the averaging. Dropout instead retains the computational efficiency of having a single model, while reaping some of the regulatory effects associated with model averaging.

2.10 Uncertainty in Neural Networks

For specific applications, it is essential that we know the model is making correct predictions, and when it is uncertain about its predictions. For example, a self-driving car needs to be sure that it will not cause a collision. If it is at all uncertain regarding a situation, it should avoid it, otherwise, risk serious injury.

In the previous section, we have shown the mathematical framework for neural networks that make a single prediction $\hat{\mathbf{y}}$ for each input $\mathbf{x}$. This single prediction is insufficient to capture the uncertainty of both the model and the data. For example, in binary classification tasks where we use a sigmoid activation function for the output layer activation function, classifications are often made extremely close to 0 or 1. If we wanted to represent uncertainty in a prediction, it would be beneficial to have the prediction further from these boundaries.

The model previously developed behaves well when making a prediction on a data point that is relatively similar to those it trained on, however, when a new data point is dissimilar from the training data, it would be desirable for the model to realise this and correspondingly make a prediction with low confidence - but this is not the case.

Uncertainty in predictions arises from two sources,

[1]Although Gal and Ghahramani [2015] propose running dropout at test time can be used to form an approximate predictive distribution that allows for empirical estimates of both the predictive mean and predictive variance. Chapter 4 explores this idea further.

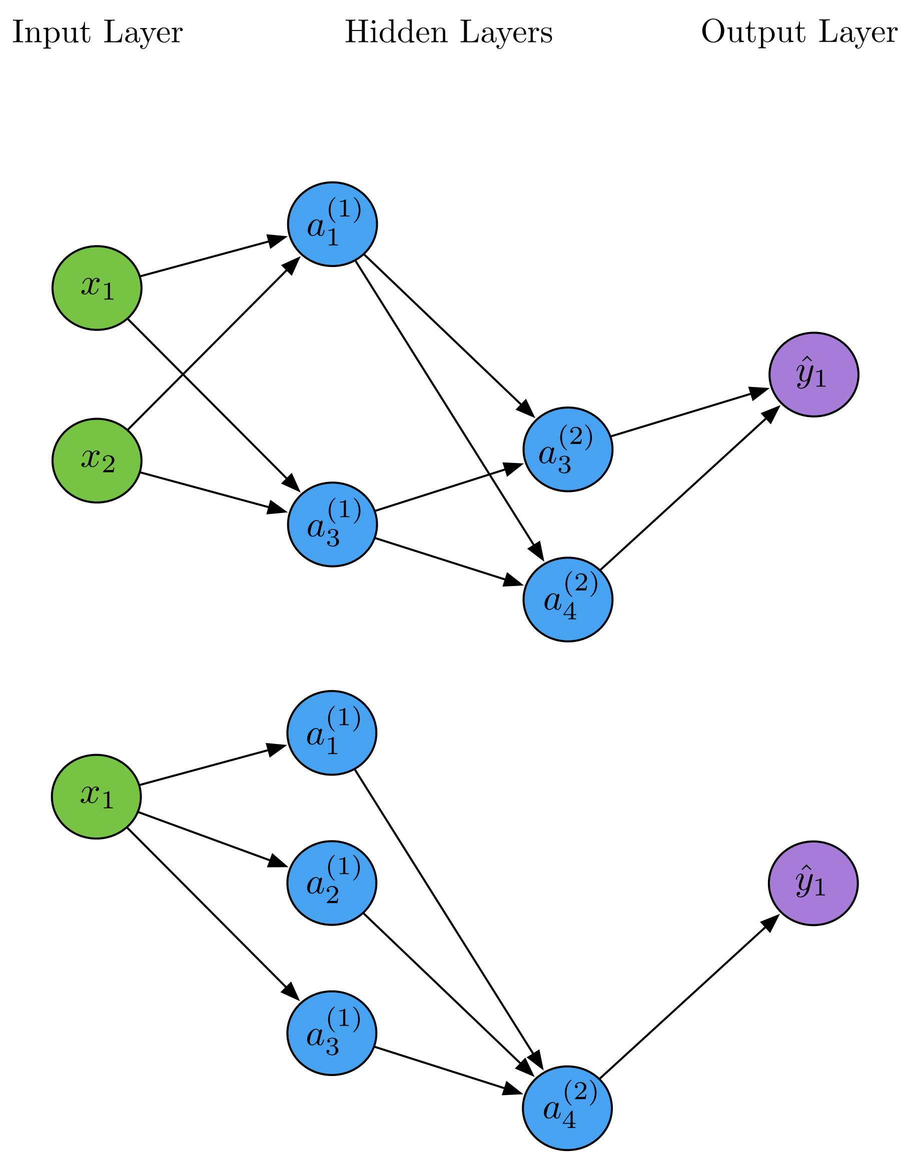


Figure 2.9: An example of a multilayer perceptron network (Seen previously in Figure 2.2) in two separate instances of dropout. In the top illustration, the second node in the first hidden layer, and the first two nodes in the second hidden layer are dropped out. In the bottom illustration one of the input nodes, the second and third nodes in the first hidden layer, and last node in the second hidden layer are dropped out.

- $\mathrm{Var}\left(\varepsilon_0\right)$ - the **aleatoric uncertainty**.
- $\mathrm{Var}\left(\hat{f}(x_0)\right)$ - the **epistemic uncertainty**.

The former, aleatoric uncertainty, corresponds with random noise in the data and is irreducible despite the choice of model. On the other hand, epistemic uncertainty comes from uncertainty in the model. The model parameters chosen are unlikely those that would minimise the loss over the data distribution. Similarly, the number of layers and nodes in each layer induces some variability in the model, adding to this model uncertainty.

Let f be the underlying function that captures the data generating distribution, so that $\mathbf{y} = f(\mathbf{x}) + \varepsilon$, where ε is random noise. Since f is deterministic, $Var(\mathbf{y}) = Var(\varepsilon)$, this is the aleatoric uncertainty.

To capture the uncertainty of predictions made by our model, we consider $Var(\hat{\mathbf{y}})$. Making the assumption that the noise is independent of the choice of model, we see:

$$Var(\hat{\mathbf{y}}) = Var(\hat{f}(\mathbf{x})) + Var(\varepsilon)$$

Here we see $Var(\hat{f}(\mathbf{x}))$ is the epistemic uncertainty corresponding to our choice of model.

One way to capture uncertainty in predictions is by using stochastic regularisation techniques, (SRTs), introduced in section 2.9.1. These SRTs are widely used in the training of deep learning models as a way to mitigate overfitting and involve adding noise to either input or the model. One such technique is dropout, introduced in Section 2.9, where activations of nodes are randomly multiplied by 0 with some probability p.

We can take a model incorporating SRTs to make n different predictions $(\hat{\mathbf{y}}_1, \cdots, \hat{\mathbf{y}}_n)$ from a single example $\mathbf{x}$. These predictions are from an **approximate predictive distribution** [Gal, 2016], allowing empirical estimates for the predictive mean;

$$\mathbb{E}[\mathbf{y}] \approx \frac{1}{n}\sum_{i=1}^{n}\hat{\mathbf{y}}_i,$$

and predictive variance;

$$\mathrm{Var}(\mathbf{y}) \approx \mathrm{Var}(\varepsilon) + \frac{1}{n}\sum_{i=1}^{n}\hat{\mathbf{y}}_i^T\hat{\mathbf{y}}_i - \mathbb{E}(\mathbf{y}^T)\mathbb{E}(\mathbf{y}).$$

Summary: This concludes the background given on deep learning. We began our journey by looking at what machine learning is and how it works. After which, we looked at deep learning, and the structure a neural network. The prediction process known as forward propagation, was detailed. This was a two-step process that first made predictions by successively finding activations of nodes, and secondly measured the accuracy of our predictions using a cost function. After which, we moved on to the optimisation strategy in deep learning known as stochastic gradient descent. This involved updating the parameters of the model, the weights and biases by a small step the direction that most decreased the cost function evaluated on a random subset of the training data. Following this, we demonstrated how the gradient of the cost function with respect to the parameters could be found in backpropagation, by recursively applying the chain rule to unravel the composition of activations. At this point, we had covered the algorithms underpinning deep learning and moved on to looking at activation functions, cost functions, and regularisation in detail.

Moving forward, we fit a deep learning model, and look at quantifying uncertainty in deep learning.

Chapter 3

Diagnosis of Misclassification

3.1 Model Fitting

Image classification is an increasingly important tool used in numerous fields, from medicine to self-driving cars. Medical diagnoses have the potential to be made quickly and accurately with the assistance of effective models, however, diagnoses must be made as accurately as possible. Misdiagnosing someone with a disease they do not have may cause unnecessary stress, or worse, not identifying a life-threatening condition may result in the loss of life. Self-driving cars must excel in object detection, which is comprised of image classification and location tracking. A self-driving car mistaking a bicycle as a car poses a potential danger to the cyclist. We need ways of quantifying model uncertainty to be able to trust the predictions deep learning models make when using them in these sensitive areas. This chapter will look at doing so by diagnosing when a model is likely to be making a misclassification, in order to know when predictions can be trusted.

Before looking into model uncertainty, we first develop a deep learning model. We consider a data-set consisting of labelled pictures of cats and dogs, which will be used to train a model to perform image classification. The data-set was made available by Kaggle [2013], with example images provided in Figure 3.1.

(a) Image labelled as cat. (b) Image labelled as dog.

Figure 3.1: Images found in training set of cats and dogs data-set before preprocessing.

3.1.1 Convolutional Neural Networks

While multilayer perceptron networks can be incredibly effective, they are often surpassed in performance by **convolutional neural networks**. As such, to develop a practical and realistic model, we make use of convolutional neural networks in this section. The full details of how convolutional neural networks function are not necessary for the implementation of uncertainty quantification that we later apply in this thesis, however, a simple overview of how they work is presented. A comprehensive resource can be found in Goodfellow et al. [2017, Chapter 9].

In convolutional neural networks, we replace nodes with what are called **filters**. Filters are small matrices that are used to look at localised regions of an image to determine if the pattern in the filter is present in the image. If the filter matches an area of the image closely, we have a large activation in this area of the image. Applying the filter to every region in the image creates a matrix of activations, showing where the pattern is present. These activations are then fed into a non-linear activation function, such as those detailed in Section 2.7, which now act element-wise.

To condense the information given in this matrix down, a process of **pooling** occurs. This process involves applying type of function to the matrix known as a **pooling function**. Pooling functions summarise local regions of matrices of activations into matrices of smaller dimensions. An example of a commonly used pooling function, **max pooling**, outputs the maximum activations of specific regions of the activation matrix. For example, we may have a 4×4 matrix of activations that we want to summarise into a 2×2 matrix using max pooling. We do so by finding the maximum activation in each quadrant of the 4×4 matrix, and reporting this in a 2×2 matrix. Other pooling functions exist that report alternative summaries, such as mean activation in specific regions.

Each layer of a convolutional network has multiple filters, and so various patterns can be found from the same input, each with their own matrix of activations. These matrices pass through an activation function, and undergo pooling separately, to give the outputs of the layer. These outputs constitute the inputs to the next layer.

The last additional step in a convolutional neural network is to make a prediction. To do so, we **flatten** the final layer's output matrices of pooled activations into vectors by appending columns on top of each other. The output of this can be used in a multilayer perceptron network as defined in Section 2.3, which when used in this way is known as a **densely-connected classifier**, to arrive at an output.

3.1.2 A Simple Approach

We now begin fitting a model to the cats and dogs data-set, making use of a convolutional neural network. Doing so, we make use of Chollet and Allaire [2018] to write the code in the programming language R [R Core Team, 2019], making use of Keras, a functional API for deep learning [Chollet et al., 2015], which runs on top of Tensorflow [Abadi et al., 2015]. All code for this project can be found at the GitHub link in Appendix A.2.

The cats and dogs data-set contains 25,000 images in the training set. Since we are not attempting to achieve state-of-the-art results, we may reduce training time by restricting ourselves to a subset of 2000 training images, with 1000 in each class. While doing this, we also use 1000 validation images distributed evenly between classes to assess model accuracy in training, and further reserve 1000 images for testing after we have fit the model.

Before we start building the neural network, we need consistent formatting for the input images. Figure 3.1 shows images of different resolutions, which is an inconvenient formatting. To amend this, we will preprocess the images to be of fixed size, of 150 pixels, by 150 pixels, with three colour channels for RGB.

We now define our convolutional neural network. Note that there are many alternative ways that this network could be defined, and usually there is some trial and error involved to determine what works best. We define our network with the first hidden layer having 32 filters of size 3×3, a ReLu activation function, and a pooling size of 2×2. The pooling size tells us the size of the neighbourhoods we want to summarise. Following this layer are four more layers, each with 64 filters of size 3×3, ReLu activations, and pooling size of 2×2. The densely-connected classifier consists of 2 additional hidden layers, with 64 and 8 nodes respectively, again making use of ReLu activations. Since we are in the setting of binary classification, the output layer consists of a single sigmoid node.

Moving on, we address how we optimise our network. As mentioned, we are performing binary classification, and accordingly, use binary cross-entropy as a loss function. As we are opting for simplicity, we do not include any penalty terms here. We make use of a batch size of 2 inputs, as defined in Section 2.5.1. We use an optimiser known as RMSProp with a learning rate of 10^{-4}. RMSProp is a popular optimiser, that makes use of an adaptive learning rate, instead of a fixed constant. The details of which are omitted, however, it is introduced in Tieleman and Hinton [2012].

Now that all of the details of the model specification have been addressed, we commence the training process. Figure 3.2 shows a plot of the model performance over 30 iterations (epochs) on the training set (shown in red), and the validation set (shown in blue), measured using the binary cross-entropy loss (top figure), and accuracy measured by the proportion of images that are correctly classified (bottom figure). As accuracy is a measure of the proportion of correct classifications, a good model will have a large accuracy. Conversely, loss is a measure of how dissimilar the predictions and labels are, hence a good model will have a low loss. After the first epoch, the classifier performs slightly better than random guessing (accuracy of 0.5), moving up to an accuracy of just over 0.7 on the validation set after 30 epochs. Notice that the model has begun overfitting after 10 epochs when the training loss/accuracy improves more than the validation loss/accuracy.

Furthermore, the validation loss increases after 20 epochs, a sign of significant overfitting and that training should stop after 20 epochs. Another interesting feature of Figure 3.2 is that while the validation loss increases after 20 epochs, the accuracy stays constant. This can be attributed to the overall class prediction remaining the same, but the model predictions being pushed towards the boundaries

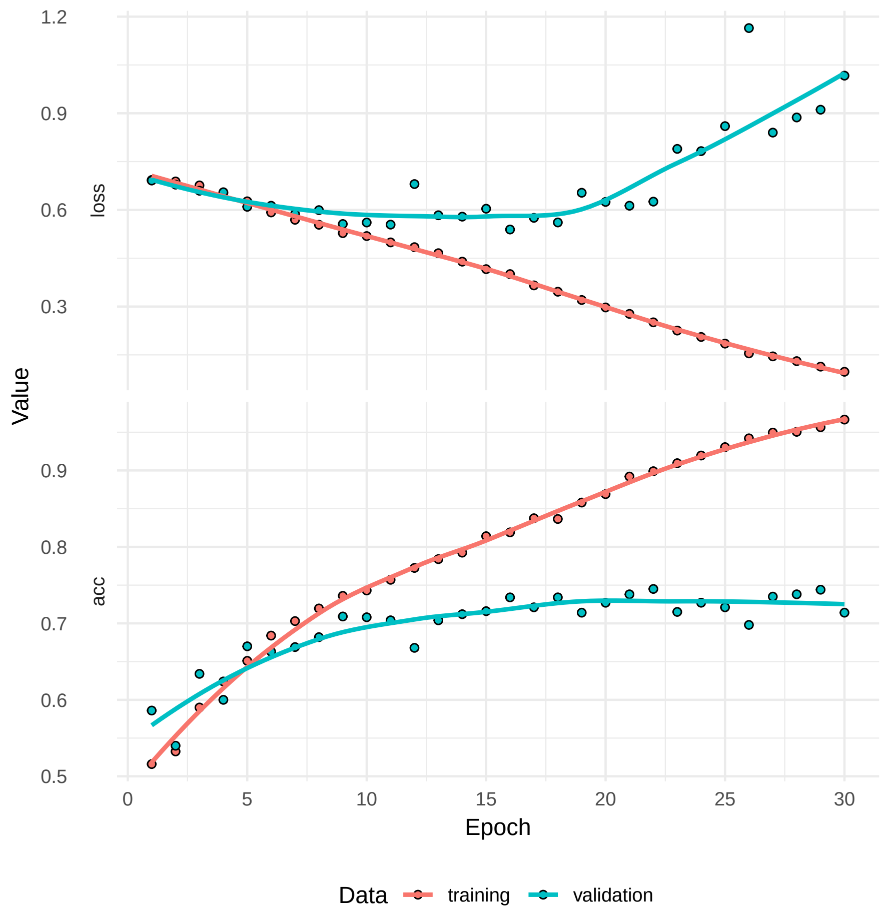


Figure 3.2: Training and validation accuracy and loss on the cats and dogs data-set using a simple convolutional neural network.

0 and 1 of the prediction interval $[0, 1]$.

As we identified, training should have stopped after 20 epochs, we retrain the model to run for 20 epochs. A validation plot is shown in the Appendix Figure A.2. The performance on the test set is indicated in Figure 3.1, correctly classifying 73.6% of the test images.

Table 3.1: Test set performance for simple convolutional neural network.

Loss	Accuracy
0.563	0.736

3.1.3 A More Powerful Approach

While it is certainly possible to train a neural network from scratch, as demonstrated in section 3.1.2, it is more efficient to make use of what is known as a **pre-trained convolutional base** in a process known as **transfer learning**, which exploits an existing neural network. This process involves taking the input layer, up to a specified layer from an existing neural network and using these layers as the first layers of a new neural network with additional layers appended. Since early layers of a neural network learn very general patterns, such as detecting particular edges, which can then recognise more complicated features in an image, the generalised early layers can be reused in another neural network performing a similar task as the patterns they learned should be relevant. Transfer learning is exceptionally efficient when the convolutional base used has been trained on an extensive data set, reducing the need for the new data-set to be as large, as the convolutional base has learned to detect useful patterns. A survey of transfer learning is given by Tan et al. [2018].

We now develop a more robust model by taking a convolutional base from the VGG 16 network [Simonyan and Zisserman, 2014] trained on the data-set ImageNet [Deng et al., 2009].

To make use of this convolutional base, we define our model to have the convolutional base connected to a new densely-connected classifier. In this new network, we make use of one hidden layer in the densely-connected classifier, using 256 nodes, a ReLu activation function, an output layer of a single sigmoid node, a batch size of 20 images, and a learning rate of 2×10^{-5}.

When using a pre-trained convolutional base with a densely-connected classifier, the convolutional base has already undergone extensive training to find optimal parameters for the task it was assigned. Contrastingly, the densely-connected classifier has had no training. It is beneficial to freeze the parameters in the convolutional base, as holding them constant allows the densely-connected classifier to be trained to make use of the convolutional base, without compromising the highly tuned parameters. After the densely-connected classifier has undergone a certain amount of training, it is then useful to unfreeze some deeper layers of the convolutional base and further train the model, in a process known as **fine-tuning**. Using this process, we freeze the entire convolutional base and train the densely-connected classifier.

A validation plot is illustrated in Figure 3.3. Here we see that the neural network makes rapid progress, quickly achieving an accuracy of 85% on the validation set.

We also note that this model is highly generalisable to new data, as there are minimal signs of overfitting.

At this point, we have tuned the densely-connected classifier to make the most of the pre-trained convolutional base and can begin the process of fine-tuning by unfreezing layers after the 5^{th} layer in the convolutional base. Doing so, we reduce the learning rate to 10^{-6}. Figure 3.3 shows that fine-tuning has greatly improved the performance of the model. After 100 epochs, we are correctly classifying approximately 95% of images in the validation set, which is reflected in the test set with an accuracy of 0.943 (Table 3.2). Notice the model immediately started overfitting when fine-tuning, as the accuracy and loss on the training set were consistently better than that on the validation set. After 100 epochs the model almost perfectly classifies the training set with near-zero loss, so that further training is redundant. Regularisation techniques would have proved beneficial to prevent overfitting from occurring here to yield an improved classification rate on the validation set, which is later demonstrated in Section 3.2.

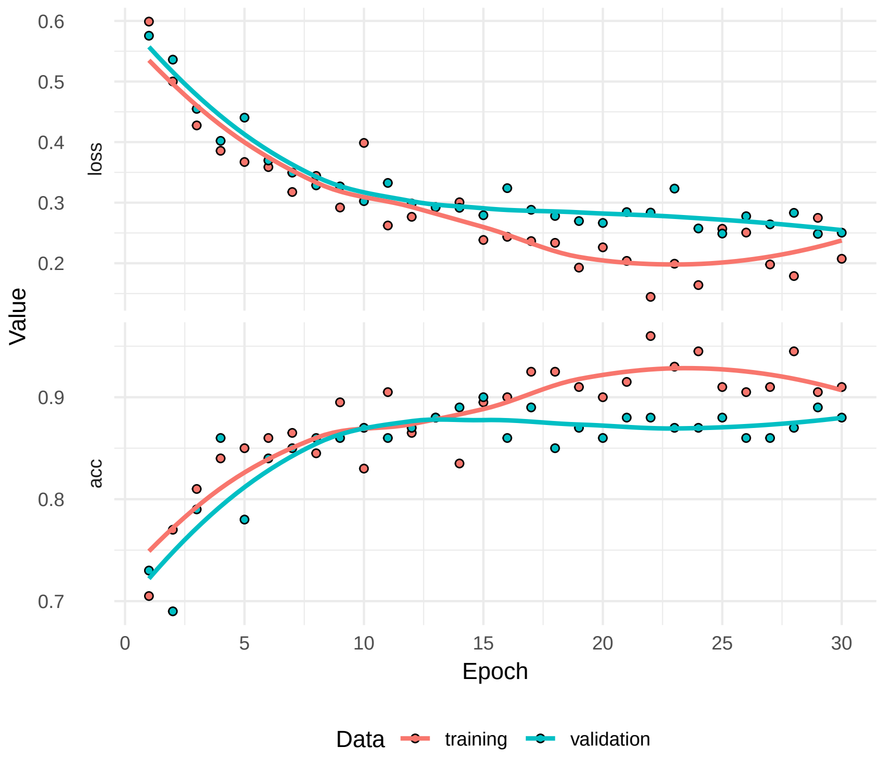
Table 3.2: Test set performance for convolutional neural network with VGG 16 as a convolutional base.

Loss	Accuracy
0.202	0.943

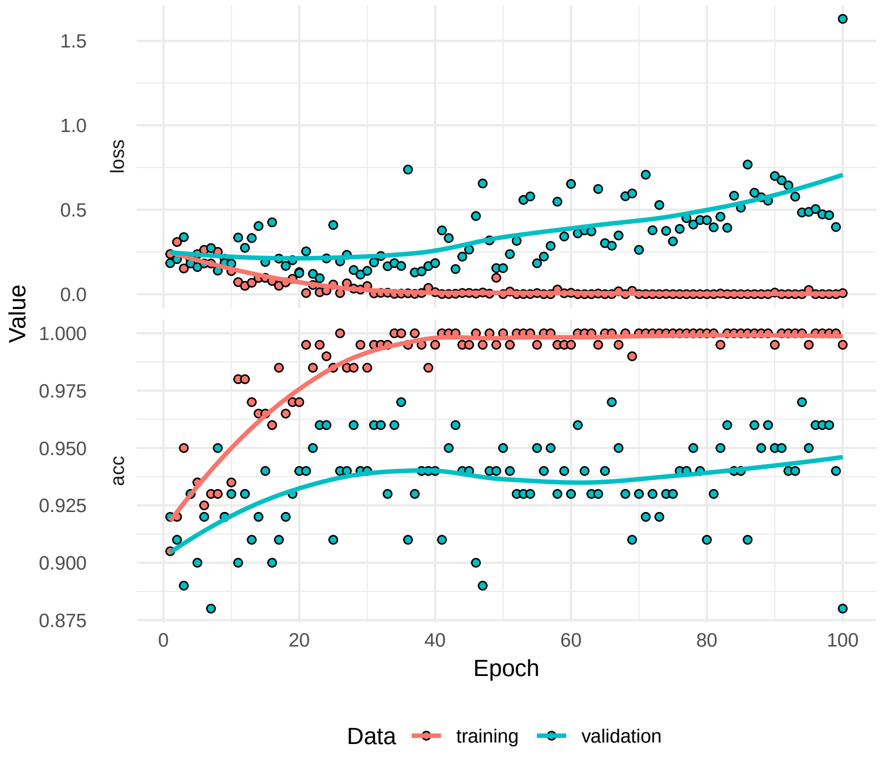
3.2 Quantifying Model Uncertainty

We now look to two strategies for quantifying model uncertainty that are built on a similar principle of using stochastic regularisation techniques to generate a distribution of predictions for an input, such as the approach taken by Gal and Ghahramani [2015], who show that running dropout at test time is an effective way of obtaining a distribution of predictions. An alternative method to applying dropout, which effectively adds noise to the model, would be to add noise to inputs instead. Here we also consider this approach by taking small transformations of an input according to randomised parameters when making predictions. The addition of noise to the input while maintaining the same label is reasonable under the assumption that a small transformation yields an input similar enough to the original that it would be perceived to be of the same class.

The intuition behind why stochastic regularisation techniques help to quantify



(a) Tuning the densely-connected classifier.



(b) Fine-tuning.

Figure 3.3: Training and validation accuracy and loss on the cats and dogs data-set found by tuning a model making use of the VGG 16 convolutional base. The top illustration shows the validation plot for tuning the densely-connected classifier, and the bottom illustration shows the validation plot for fine-tuning from the 5^{th} convolutional layer of the convolutional base.

model uncertainty may be that misclassifications are made due to the model looking at random/insignificant patterns in the input when making a prediction. Small transformations to the image or the use of dropout may disturb these patterns enough to alter the way predictions are being made. In contrast, patterns that the model looks for in images that are correctly classified may be stronger than those insignificant patterns in misclassified images, and hence more robust to small transformations/dropout. Applying these transformations to the same image allows a distribution of predictions to be obtained. The presence of a well-established pattern in the input that is robust to transformations results in a distribution of predictions with low variance, corresponding to certainty. In contrast, inputs that may have patterns that result in misclassifications would have a more substantial variance, corresponding to uncertainty.

3.2.1 Updating the Model Fit

To incorporate stochastic injection techniques into our model for the purpose of uncertainty quantification, we need to update the way we fit our model, as we will be using dropout, and applying augmentations to our images at test time. Applying these transformations does not necessarily require any change in the model. However, it is beneficial to train the network on images that have undergone similar transformations, as this will lead to a model familiar with any artefacts associated with the transformations. As an example, new pixels must be added to a rotated image, so we develop a model robust to the addition of new pixels in this manner. Two new models are developed, with a similar design to that in Section 3.1.3. The exception in their designs is both new models are trained with randomly augmented training data. The augmentations of the training images occur as: random rotations of up to 40 degrees, random width and height shifts of up to 20% of the image, random shearing of up to 0.2 radians, random zooming up to a 20% increase, and random flipping about the y-axis. Also, as we will be using a model that uses dropout at test time, we need to train a neural network that includes dropout in its architecture. We do so by adding dropout to the densely-connected classifier, with a dropout rate of 0.5. Recall the dropout rate for a particular layer is the probability of activations in the layer being multiplied by zero. Here we apply dropout at both training and test time.

In Appendix A.1, validation plots are given for the dropout model (Figure A.3), and for the non-dropout model (Figure A.4). The results of each are similar to their counterpart in Section 3.1.3, however, both exhibit reduced overfitting, which is expected as they are developed using regularisation techniques. The dropout model shows further reduced overfitting compared to the non-dropout model and has a marginally improved accuracy on the validation set. However, the non-dropout model outperforms the dropout model on the unaugmented test set, as observed in Table 3.3. While this is the case, the dropout model is running dropout at test time, and hence has a slight handicap. We expect it would perform similarly, or outperform the non-dropout model if: the same model was developed to not implement dropout at test time, or, we instead made multiple predictions on each image in the test set and used the empirical mean as the model prediction.

Table 3.3: Comparison between performance on the test set for the model not making use of dropout and the model making use of dropout.

Model	Loss	Accuracy
Non-Dropout	0.388	0.959
Dropout	0.425	0.965

3.2.2 Methods for Quantifying Uncertainty

Now that training of the models has been completed, we move on to methods for quantifying uncertainty. To achieve this, we will take the test set of images, and perform stochastic injection techniques detailed in Table 3.4 to obtain a set of predictions for each test image, for each model, where here we describe the model as the neural network equipped with the stochastic injection technique. We can use these predictions to form estimates of the empirical mean and variance of the predictions. To increase computational efficiency, only 50 predictions were made on each of the 1000 test images. Future work may consider making more predictions on each image.

3.2.3 Expectations

It is well-known that convolutional neural networks are inherently translation-invariant [Goodfellow et al., 2017, Chapter 9], meaning that the output of an image does not depend on its translation. This invariance is not the case for other transformations that we use, such as scaling, zooming, or rotating. As such, we should expect that the predictions made using translations should be less varied than those made using other transformations. However, this may not necessarily be reflected in the results for three reasons. All models have random flipping about

Table 3.4: Stochastic injection techniques used for each model, allowing for predictive distributions to be made for the labelling of cat-dog images. Indicated is whether dropout was used (Dropout), the range of degrees of rotation (Rotation), the width shift range as a fraction of total width (Width), the height shift range as a fraction of total height (Height), the shear range as a shear angle in radians (Shear), and the zoom range (Zoom). Transformations were applied using $image\_data\_generator()$ in Keras, and transformations that result in the image having missing pixels made use of the argument $fill\_mode = nearest$ to fill these pixels in. In all cases the argument $flip = TRUE$ was used, reflecting images about the y-axis with probability $p = 0.5$.

Model ID	Dropout	Rotation	Width	Height	Shear	Zoom
1	FALSE	0	0.0	0.0	0.0	0.2
2	FALSE	0	0.0	0.0	0.2	0.0
3	FALSE	0	0.0	0.2	0.0	0.0
4	FALSE	0	0.2	0.0	0.0	0.0
5	FALSE	40	0.0	0.0	0.0	0.0
6	FALSE	40	0.2	0.2	0.2	0.2
7	TRUE	0	0.0	0.0	0.0	0.2
8	TRUE	0	0.0	0.0	0.2	0.0
9	TRUE	0	0.0	0.2	0.0	0.0
10	TRUE	0	0.2	0.0	0.0	0.0
11	TRUE	40	0.0	0.0	0.0	0.0
12	TRUE	40	0.2	0.2	0.2	0.2

the y-axis, a transformation which models are not necessarily invariant to. Certain artefacts associated with translating will impact model predictions. Translating images results in some pixels being discarded, and consequently introduces new ones. Also, while predictions made using convolutional neural networks may not necessarily be invariant to all transformations, data-augmentation at training time improves the robustness of models to random transformations of inputs [Engstrom et al., 2017].

3.3 Results

We aim to quantify model uncertainty of predictions made using stochastic injection techniques applied to the cats and dogs data set using the model introduced in the previous section.

We hope to see that the distribution of model predictions for incorrect classifications is significantly different from that of correct classifications, as this would permit the model to recognise images that it may be misclassifying. Doing so would allow these uncertain predictions to be verified by a human. In this Section, results will be drawn from looking at the variation between predictions arising from stochastic injection techniques. In particular, some key statistics we use are the empirical mean and variance of the multiple predictions made on the test images. An overview of the prediction process is seen in Figure 3.4.

The empirical mean of the predictions provides an estimate of the expected response (Section 2.10). Hard classifications of 0 for Class Cat and 1 for Class Dog were found by using a cutoff value of 0.5. If the empirical mean of predictions for an image is less than 0.5, then the image is predicted to be Class 0 (Cat), and otherwise predicted as Class 1 (Dog). Table 3.5 summarises the number of misclassifications made by each model on the 1000 test images, along with the classification rate of the model as a measure of accuracy. The classification rate is defined as the proportion of images correctly classified. Notice that all models enjoy a high classification rate, ranging from 0.964 to 0.972, meaning that up to 97.2% of images were correctly classified.

Previously we introduced the notion that misclassifications may be made due to insignificant patterns making their way into the model's prediction. We hypothesised that these misclassifications might be more sensitive to transformations, and hence the predictions that are made from misclassified images should have more variation. We assess the reasonableness of this hypothesis through Figure 3.5, in which we look at the density of the absolute error of each prediction using Model 8. Note that all models exhibit similar qualities. The absolute error of prediction $p_{i,j}$ of the i^{th} prediction for the j^{th} image in the test set is defined to be:

$$\text{Absolute Error}(p_{i,j}) = |p_{i,j} - y_j| \,,$$

where y_j is the true label corresponding the j^{th} image in the test set.

Figure 3.5 shows a frequency polygon for the density of absolute error in predictions for Model 8. Correctly classified images have predictions that have low absolute error (shown in red), and misclassified images have predictions with high absolute error (blue). While this is tautological, it is interesting that correctly classified predictions have smaller variation than misclassified predictions. The reduced variation is evident as the peak density of correctly classified prediction is more prominent than those of misclassified predictions. There is more certainty

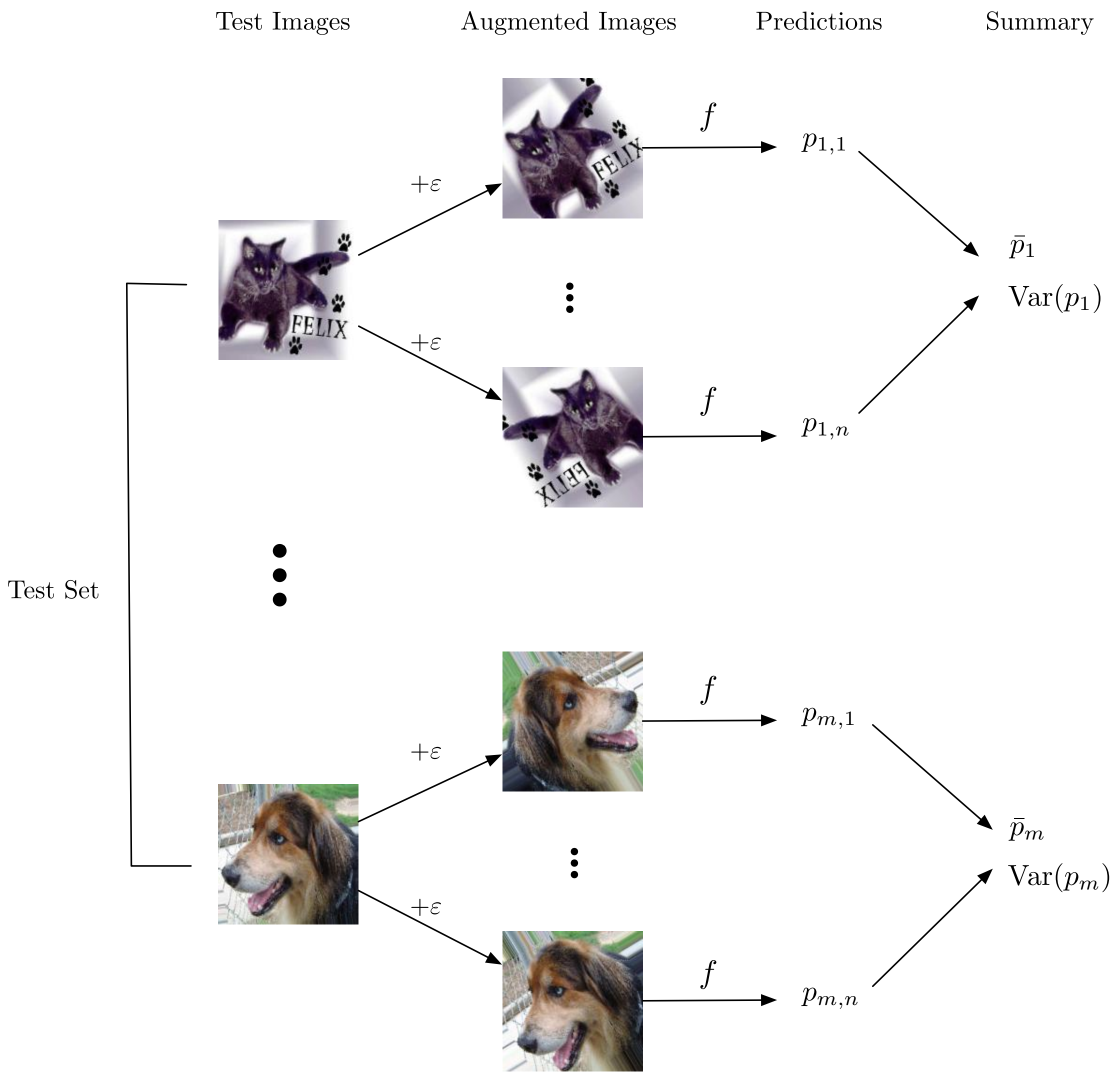


Figure 3.4: Overview of prediction process. Images are taken from the test set and undergo random transformations according to Table 3.4, 50 times for each test image. The random transformations can be thought of as adding some noise ε to the image. Predictions are made to the transformed images by applying the model function f, to give $n = 50$ predictions for each of the $m = 1000$ different test images. Empirical estimates of the mean and variance are found for each collection of images.

exhibited by correctly classified predictions, supporting the hypothesis that predictions of misclassified images may be more sensitive to noise.

We can explore this idea that misclassified images have higher variation within

Table 3.5: Number of misclassifications made for each model (Table 3.4) on the 1000 test images. Images classes were predicted as the empirical mean of the predictions with a cutoff of 0.5. We then define a misclassification to be an image that was predicted to be the incorrect class. Accuracy is reported as the classification rate, the proportion of correct classifications.

Model ID	Misclassifications	Accuracy
1	31	0.969
2	35	0.965
3	32	0.968
4	28	0.972
5	34	0.966
6	35	0.965
7	28	0.972
8	35	0.965
9	33	0.967
10	29	0.971
11	34	0.966
12	36	0.964

their prediction by looking at the variance of the predictions for each 1000 test images for each model against the mean absolute error of the predictions, shown in Figure 3.6.

Figure 3.6 shows the variance of predictions for the 1000 test images. Taking into account that misclassifications have lower representation that correct classifications, we see that misclassifications tend to have a larger variance than correct classifications. The subplot for Model 12 illustrates this clearly. We see almost equal spread of variance in predictions for misclassifications, whereas correct classifications tend to have lower variance. There are still misclassifications that have very low variance in their predictions, indicating that some misclassified images are robust to the stochastic injection techniques applied.

We can summarise findings by comparing the mean of the variances of predictions for correctly classified images to those that are misclassified. That is, for each image, we find the variances of the model predictions for the same image. We then find the mean of those variances for the images that were misclassified and compare it with that of those correctly classified. This is seen in Figure 3.7.

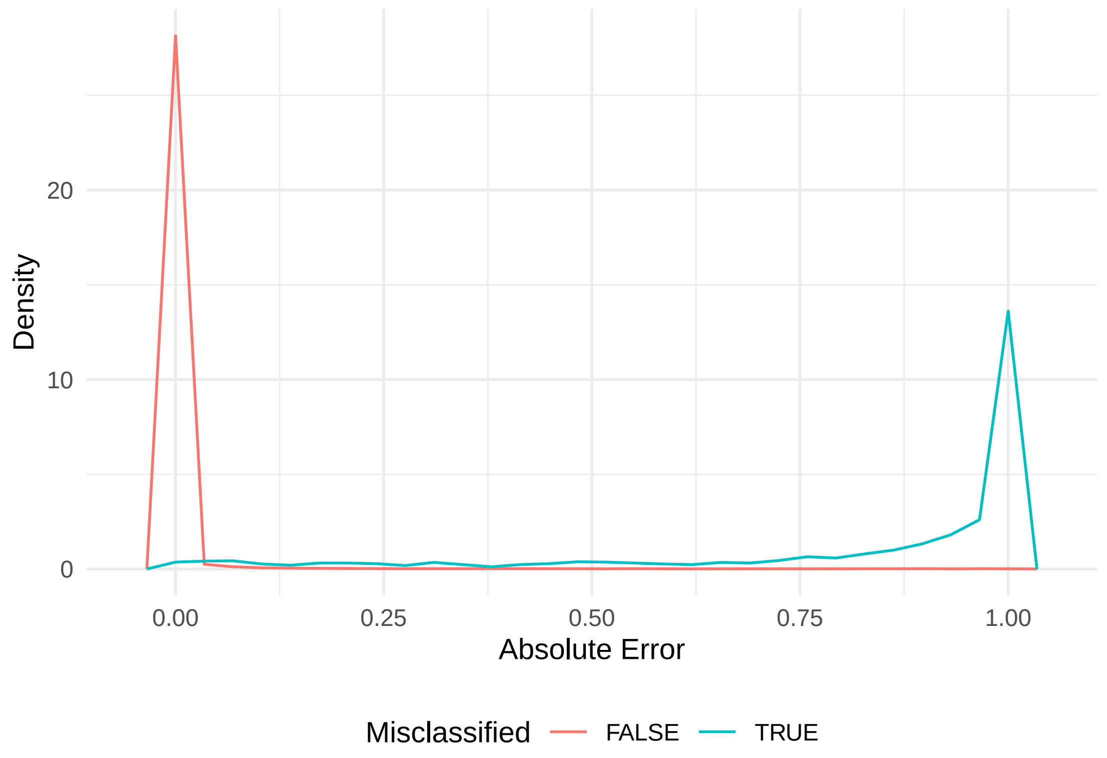


Figure 3.5: Frequency polygon indicating the density of absolute error in predictions for Model 8. Absolute error corresponding to predictions that were correctly classified is shown in red, and shown in blue for misclassifications.

Figures 3.7 summarises key results from Figures 3.6, and 3.5 illustrating that the mean-variance of the predictions for misclassified images is significantly higher than that of correctly classified images. We use this to indicate that a model might be making an inaccurate prediction when the variance of predictions for an example is over some threshold. We show a plot of the mean-variance of correctly classified images against the mean-variance of misclassified images.

Higher variance for misclassified images and lower variance for correctly classified images is desirable, so we seek to optimise these two criteria. Some models perform better in some aspects of this optimisation, and others in different aspects. We cannot simply assess which model is optimal, as we do not have an optimal model for both criteria simultaneously. Instead, we can look at models that are **Pareto-optimal**. We say that a model is Pareto-optimal if there does not exist a model with a lower variance of correct predictions while also having a higher variance of misclassified predictions.

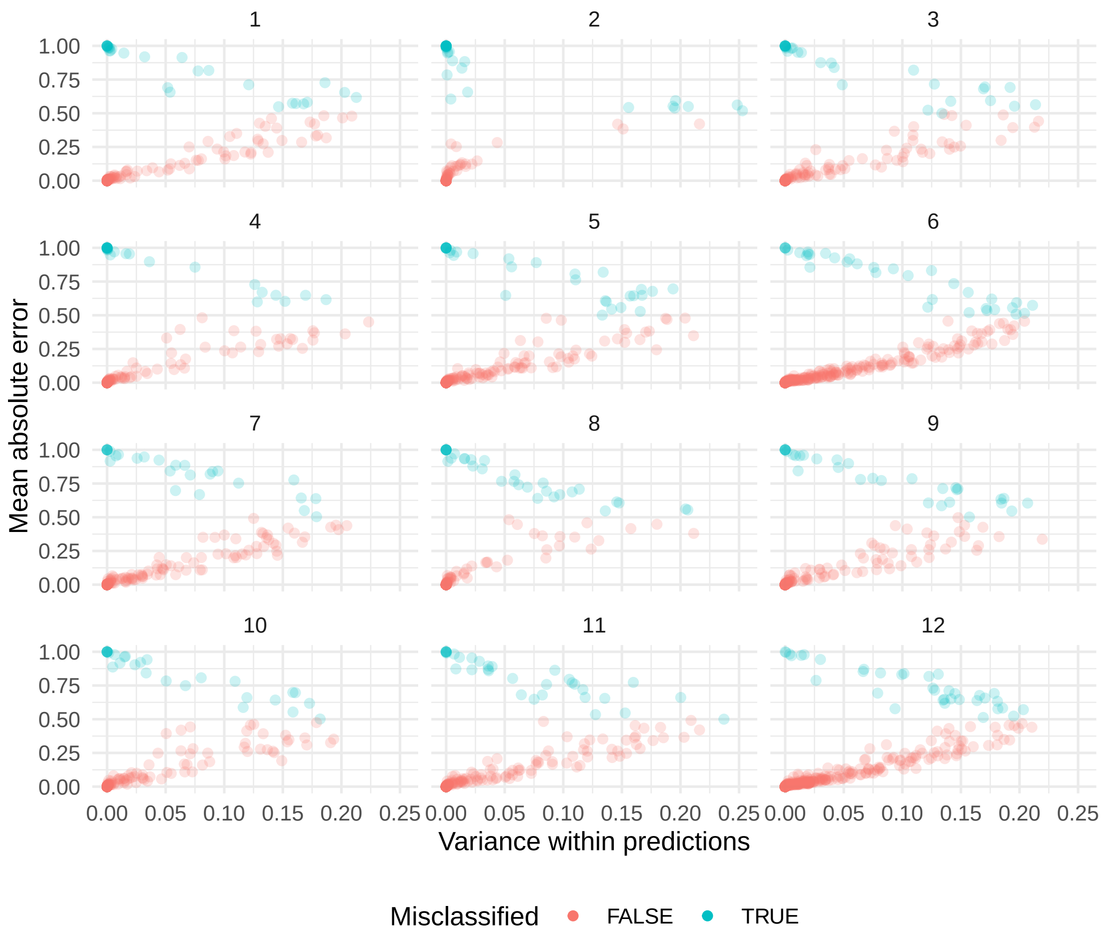


Figure 3.6: Scatter plot of the variance of predictions for each image in test set for each model, against the mean absolute error of those predictions. Predictions with a mean absolute error greater than 0.5 are misclassifications, and shown in blue. Correct classification are shown in red.

Figure 3.7 shows Models 2, 8, 10, 3, 9, 12 to be Pareto-optimal (in order of ascending variance of correct predictions) for the objective of minimising the variance of the correctly classified mean predictions, while maximising the variance of the misclassified mean predictions. We believe that these models may offer the best detectability of misclassified images while not misidentifying correctly classified images as misclassifications. While Models 3, and 10 are identified to be among the Pareto-optimal, they only have a marginally greater variance within misclassified predictions compared to Model 8, and a much greater variance within correct predictions. As such, Models 3 and 10 are likely to perform worse than Model 8.

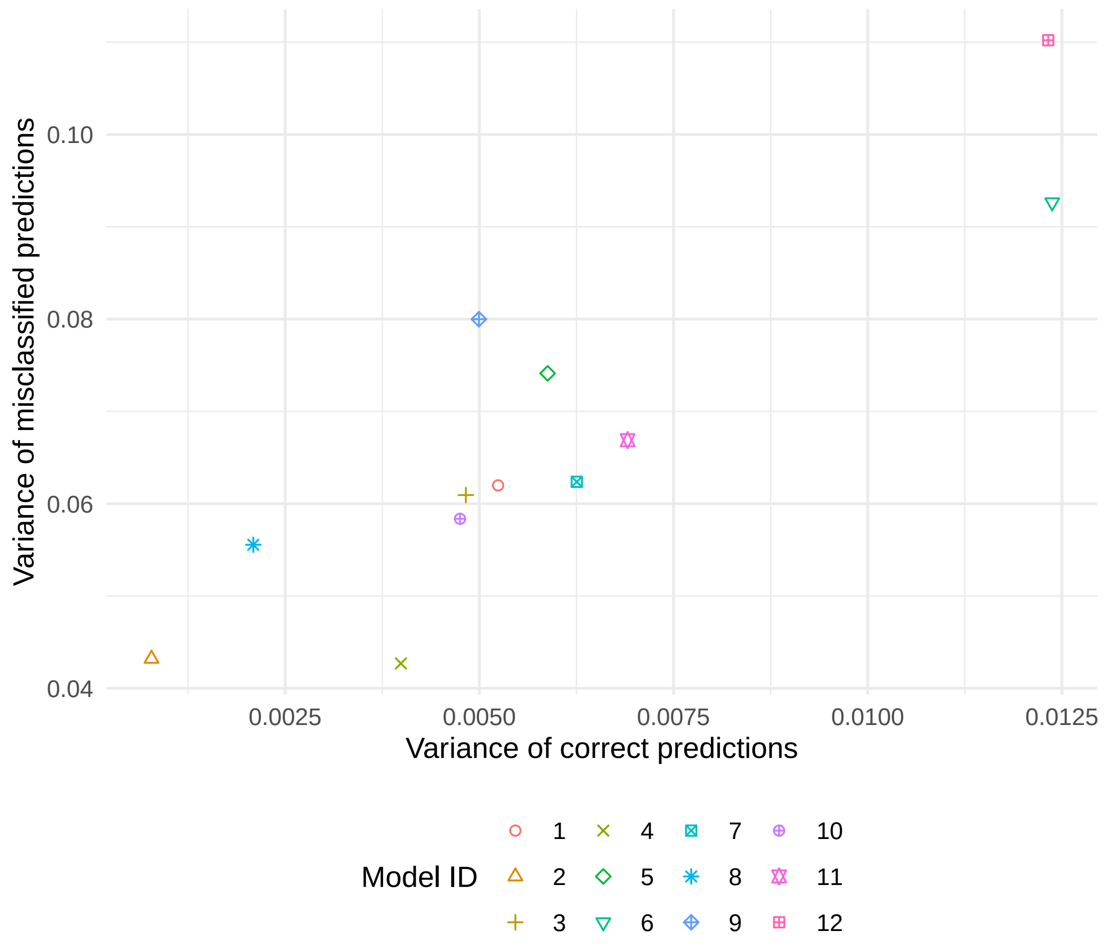


Figure 3.7: Scatterplot indicating the (mean) variance of predictions for a correctly classified image against the (mean) variance of predictions for misclassified images for each model described in Table 3.4. There is a moderately strong positive linear relationship between the variance of correct predictions and the variance of misclassified predictions. Models that are desirable have low variance for correct predictions, and high variance for misclassified predictions.

To diagnose potential misclassification of a particular image, we could look at the distribution of the predictions made. In particular, Figure 3.7 summarises the main result of this research, that misclassifications exhibit larger variance in predictions than correct classifications. We can look at using a *threshold* variance to *flag* an image as a potential misclassification if it has variance greater than this threshold. An example of this using an arbitrarily chosen threshold variance of 0.05 is demonstrated in Figure 3.8.

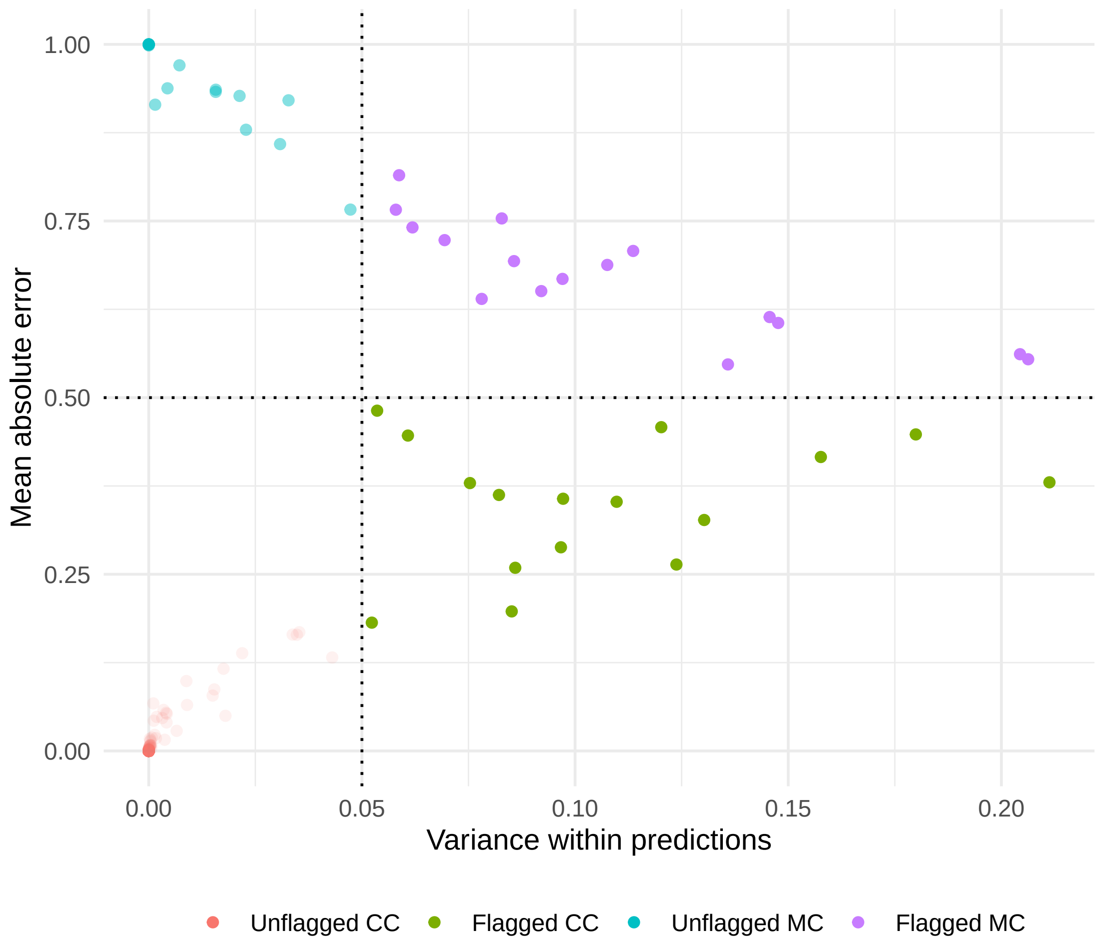


Figure 3.8: Scatter plot of the variance of predictions for each image in the test set for Model 8, against the mean absolute error of those predictions, adapted from Figure 3.6 to now use an arbitrarily chosen threshold variance of 0.05. The vertical dotted line indicates this threshold. Points with variance greater than this threshold are flagged as potential misclassifications. A horizontal dotted line is used to separate correct classifications (CC) and misclassifications (MC). Points are coloured according to which quadrant created by the dotted lines they lie within. For improved visualisation, we control the transparency of points in each quadrant its number of points.

Table 3.6 gives a summary Figure 3.8 by counting the number of unflagged and flagged; misclassifications and correct classifications using Model 8 with a threshold variance of 0.05. Table 3.7 further summarises this result, where we compute the proportion of misclassifications that are flagged. This proportion is calculated as the number of misclassifications flagged (shown in purple), divided by the total number of misclassifications (shown in blue and purple).

The proportion of flagged images that are Misclassifications is similarly calculated as the number of misclassifications flagged (shown in purple), divided by the total number of flagged images (shown in purple and green).

Table 3.6: Summary of Figure 3.8, detailing the number of Unflagged and Flagged Misclassifications (MC) and Correct Classifications (CC) in Model 8 using the threshold variance of 0.05.

	Unflagged	Flagged
MC	19	16
CC	949	16

Table 3.7: Summary of Figure 3.8, detailing the proportion of misclassified (MC) images that are flagged in Model 8 using the threshold variance of 0.05, and the proportion of flagged images that are misclassified.

Prop MC Flagged	Prop Flagged MC
0.46	0.50

The effectiveness of the threshold variance method to identify potential misclassification in each model is illustrated in Figure 3.10, by comparing proportion misclassified images we would flag given a latent threshold variance[1], to the proportion of the images that get flagged that, are misclassifications for a range of thresholds. Firstly, we consider Model 8 in isolation.

By setting a large threshold variance, most of the images that are flagged were misclassifications, since these images have higher variation in their predictions. The colour gradient of the line illustrates this in Figure 3.9, which except for at the start, runs red to blue (high variance to low variance) for a large proportion of flagged images misclassified to a low proportion of flagged images misclassified. It should be noted that as the total number of images considered for a small proportion of misclassifications is small, this result is subject to high variance. This helps to explain why we see the proportion of flagged images that are misclassified as 0

[1]For each model the predictions are ordered by decreasing variance. Then, for each prediction, we find the proportion of points with equal or lesser variance and flag them as potential misclassifications. Corresponding proportions are computed using the methods demonstrated in Figure 3.8.

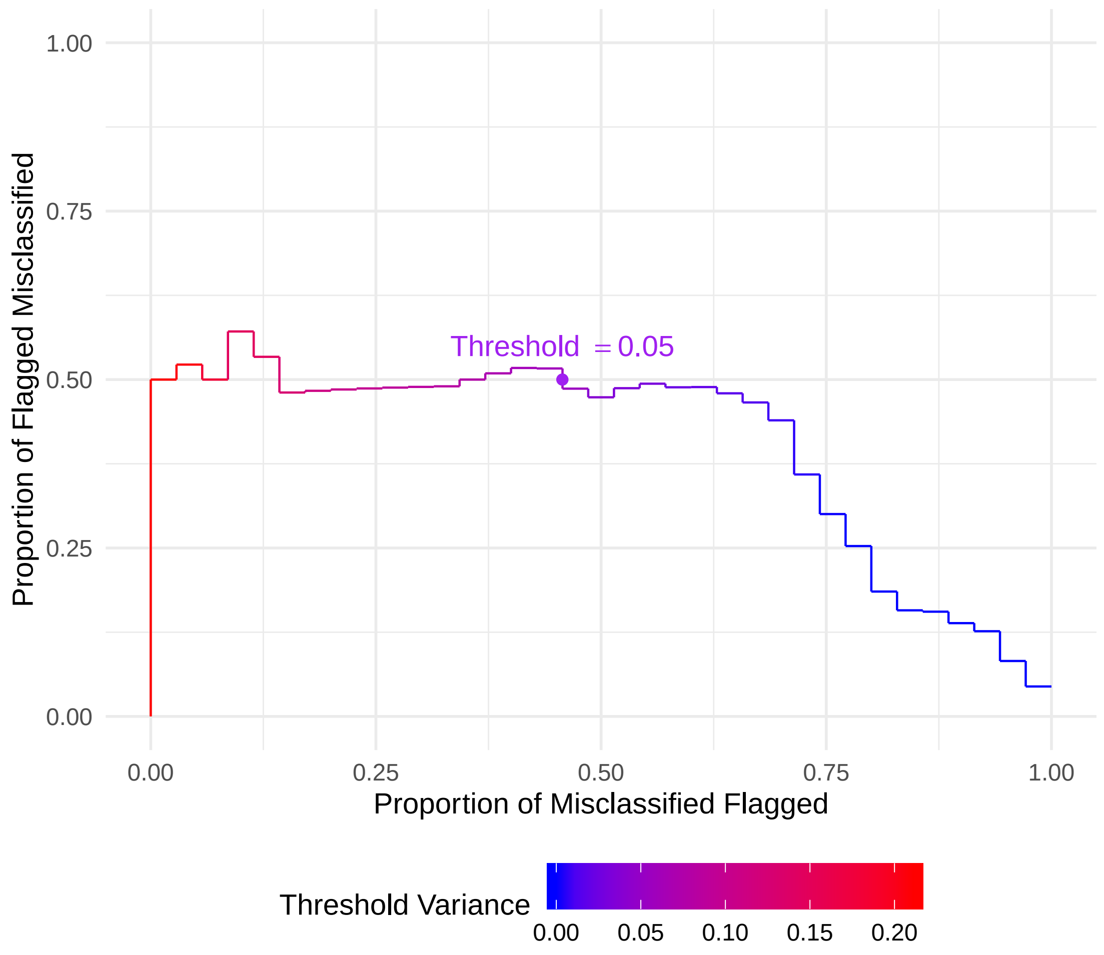


Figure 3.9: Graph indicating the proportion of misclassified images that are able to be flagged (due to variance above a threshold) as potential misclassifications against the proportion of flagged that are misclassified using Model 8. The proportions in Table 3.7 found using a threshold variance of 0.05 are included.

initially, where the most varied prediction was correctly classified, seen in Figure 3.8. The proportion of flagged images that are misclassified rapidly increases after this point and becomes more stable as more images are taken into consideration by setting the threshold variance to be lower.

Figure 3.10 considers the proportion of misclassified images that are flagged against the proportion of flagged images that are misclassified for all models. As the relationship between threshold variance and these proportions is simple, it is omitted in preference of colouring the graphs by their Model ID. The relationship can

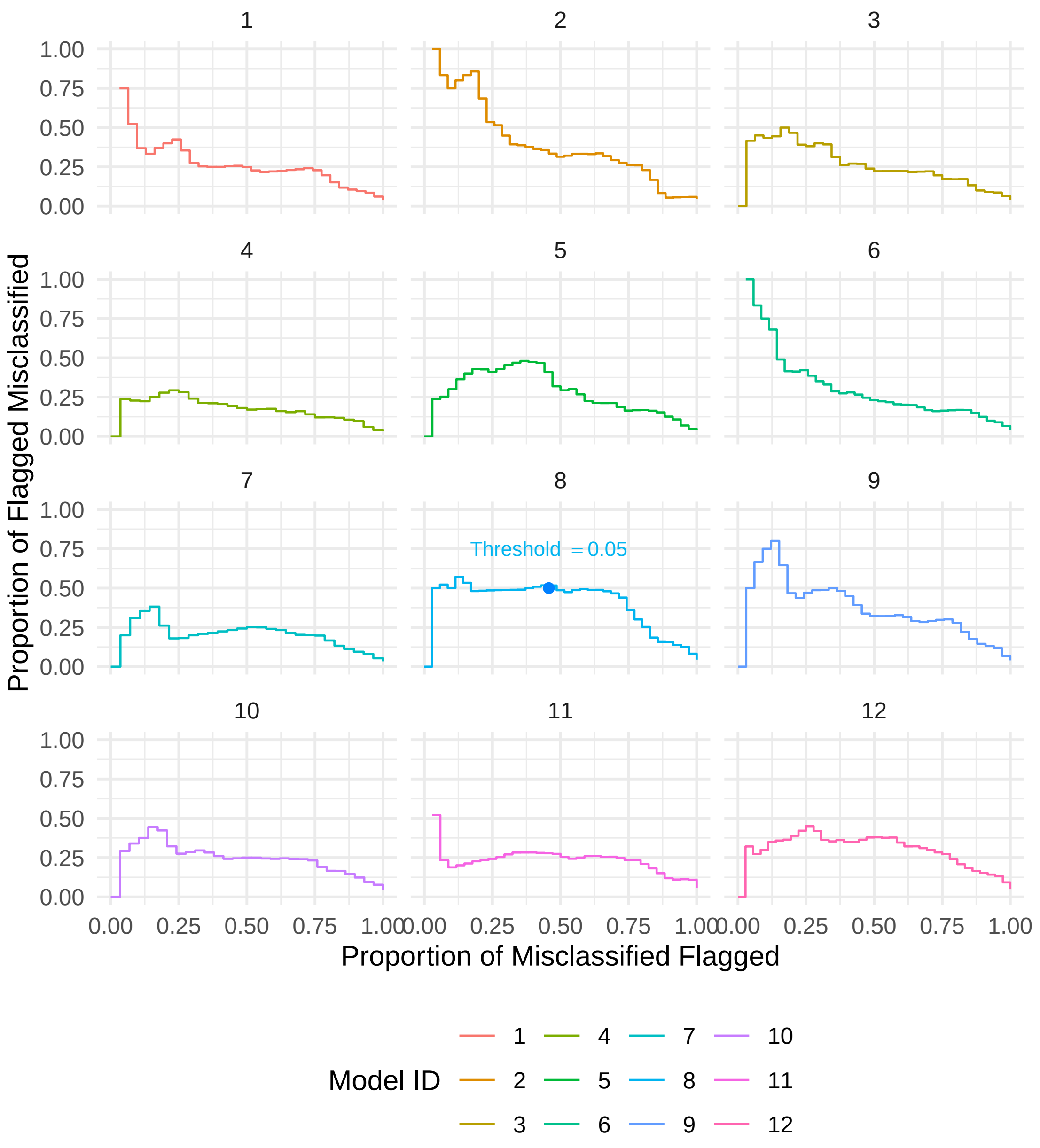


Figure 3.10: Graph indicating the proportion of misclassified images that are able to be flagged (due to variance above a threshold) as potential misclassifications against the proportion of flagged that are misclassified. By setting the threshold variance to be lower, more misclassified images are flagged, at the cost of having more correctly classified images also flagged.

instead be found in the Appendix in Figure A.1.

Models 2, 6, and 9 both perform well if the objective is to only flag a small proportion of misclassifications, as most of the images flagged are indeed misclassifications. However, as mentioned previously, this result is subject to large variance[2]. Moving towards flagging a more substantial proportion of the misclassifications, we see a drop in the proportion of flagged images that were misclassifications for most models. We expect this relationship, as correct classifications should have lower variance on average, and also because there are many more correct classifications. As the threshold variance is reduced further, all images become flagged, and hence the proportion of misclassified images flagged becomes the proportion of misclassified images in the data.

If we wish to flag a large proportion of the predictions as misclassifications, Model 8 has a distinct advantage. We see that the proportion of images that it flags that are misclassified drops off less rapidly than all other models. Model 8 flags a near equal share of misclassified images and correctly classified images until just before it flags 75% of the misclassified images. This is evident in Figure 3.8, where for all variances over 0.025, there is a roughly equal share of misclassifications and correct classifications.

The models favoured for diagnosing misclassifications, as found in Figure 3.10, are somewhat expected. In particular, Models 2 and 8 have great performance. Referring to Table 3.4, both models make use of shearing, which is a relatively deforming transformation. Model 8 also makes use of dropout, in which Gal [2016] demonstrates as an effective stochastic injection technique to assist in quantifying uncertainty.

These results demonstrate that stochastic regularisation techniques can be used to diagnose potential misclassifications by flagging images as potential misclassifications when there is a large variation in the predictions that exceeds some threshold. In addition, these methods are not limited to only adding noise to the model; they can be extended to adding noise to inputs to the model in the form of random transformations.

[2]Future work may consider using a more extensive test set so that small proportions consist of more images to yield more reliable results.

Chapter 4

Conclusion

The purpose of this thesis has been to evaluate how a deep-learning model may be able to quantify uncertainty in its predictions. In doing so, Chapter 2 gives a brief introduction to machine learning, where we attempt to illicit a relationship between features and response variables. We outline the different types of data used in machine learning, such as the training, validation, and test set. After which, we look at how parameters control the way that the model views the relationship between features and response, and how we can tune the model to find the best parameters that minimise a cost function.

Following the introduction to machine learning, Chapter 2 casts its focus on the specific area of machine learning known as deep learning, and the associated models known as neural networks. We begin by exploring the details of a specific neural-network, the multilayer perceptron network, which consists of layers of multiple nodes that feature a type of hierarchical learning structure. Basic patterns give rise to more abstract patterns as information propagates through the layers of the network. The mathematics of the prediction process known as forward propagation is formalised, which involves looking at how nodes apply a non-linear activation function to an affine transformation of their inputs to find their activations, which are sent to nodes in subsequent layers. Tuning the network to give optimal parameters is explored; we look at gradient descent and stochastic gradient descent, two similar optimisation strategies that allow the model to update its parameters by adjusting them in the direction that most reduces the cost function. In order to perform gradient descent, we demonstrate how the gradient of the cost function is found, using a process of backpropagation.

After we have covered the essential components of deep-learning, we look towards regularising models to prevent them from overfitting. Two stochastic regularisation techniques are explored, which involve either adding noise to the inputs of the model, or to the model itself in a process known as dropout, which becomes a significant component of Chapter 3.

Chapter 3 begins by fitting a deep-learning model to a data-set consisting of labelled images of cats and dogs, making use of $2,000$ training images, $1,000$ validation images, and $1,000$ test images. We start by first developing a basic model, and work towards a more powerful model, with which we achieve accuracies of approximately 97% on a held-out test set. Chapter 3 then further explores the use of typical stochastic regularisation techniques in an alternative framework that attempts to explain and quantify uncertainty in predictions using deep-learning, following the work of Gal [2016]. In doing so, we apply multiple different image augmentation techniques to transform images in the test set to obtain multiple predictions of the underlying response, under the assumption that the small transformations applied do not change the class label of the images.

Image augmentations are combined with the approach taken by Gal [2016], giving us distributions of predictions for each test image. We look at these distributions separately to identify their characteristics for both correctly classified images and misclassified images in an attempt to quantify uncertainty in model predictions. The ultimate result of this work is that misclassified images generally exhibit a more considerable variance in their predictions, and as such the variance of predictions can be used to identify when a model may potentially be misclassifying an image. This result has significant applications to sensitive areas where deep learning models are deployed, where certainty in predictions is crucial. Furthermore, the methods we use can be applied to already existing models if they make use of dropout or are robust to image augmentations, which is often the case.

4.1 Limitations

One limitation of this method is that there are correctly classified images that have a large variance of model predictions on a single image. If we were to set a threshold variance to flag an image as a potential misclassification, we would flag both correctly classified images and misclassified ones. Similarly, using this method, we are unable to detect some misclassifications.

Using stochastic injection techniques to quantify uncertainty results in no increase in the training time, as stochastic regularisation techniques are a highly popular form of regularisation. Doing so allows already existing models to be used for uncertainty quantification. There is a drawback to this method, however, as there is an added computational expense in the prediction process. Namely, for each image we wish to predict on, we end up replicating and transforming it to obtain multiple predictions. The computational expense of prediction scales linearly with the number of replications made, which is a significant drawback.

A further limitation of this work is that it does not guarantee a diagnosis of misclassification; however, it indicates when a model *may* be making a misclassification.

4.2 Future Work

Future work may consider looking at different stochastic injection techniques, for example, brightness and contrast adjustments, as well as different ranges for the transformations we used in Table 3.4.

Further refinements could be made by increasing the number of predictions made for each image, at the cost of increased computation complexity. Doing so would give a more reliable distribution for the predictions made and allow for a more accurate comparison between models.

In addition, future work should look at applying these stochastic regularisation techniques on models developed using alternative data-sets. In particular, this work primarily focussed on binary classification. A natural extension to this would be to look at deploying these techniques on models performing multinomial classification.

Appendix A

Appendix

A.1 Figures

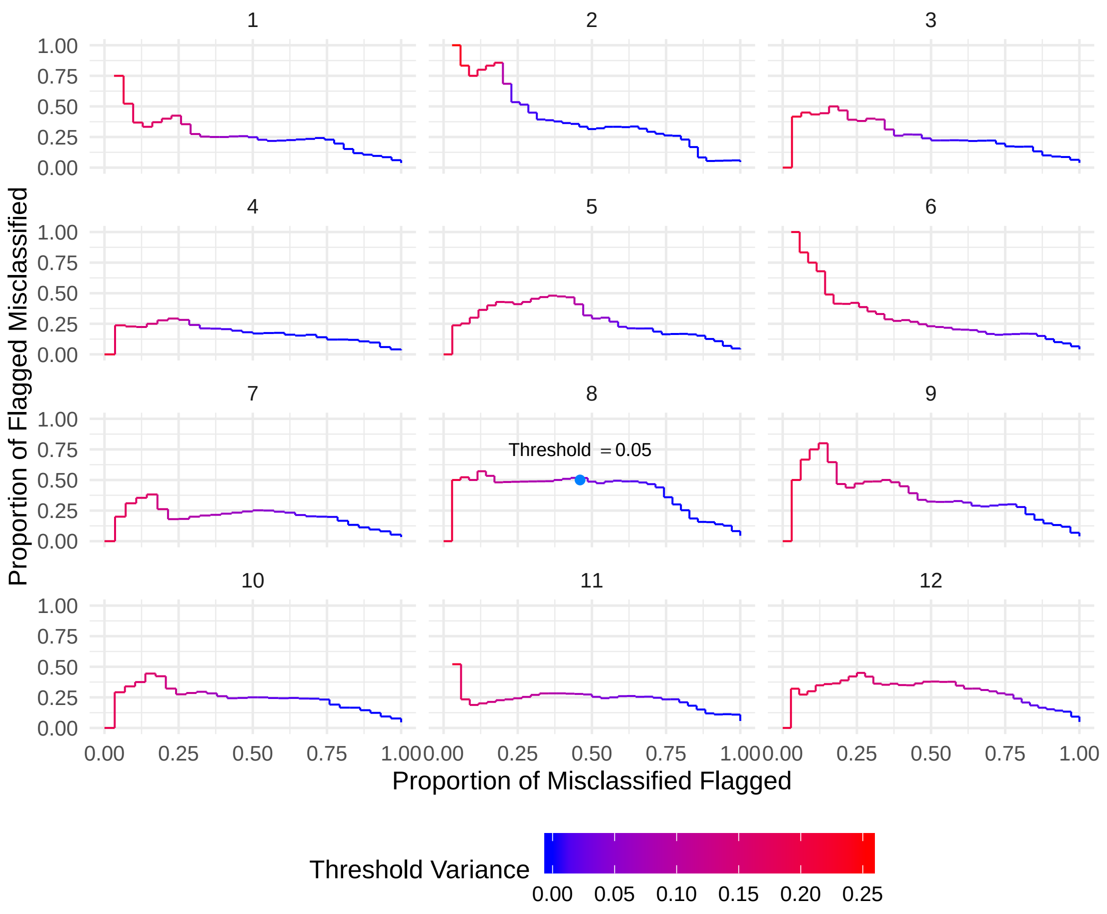


Figure A.1: Graph indicating the proportion of misclassified images that are able to be flagged (due to variance above a threshold) as potential misclassifications against the proportion of flagged that are misclassified. By setting the threshold variance to be lower, more misclassified images are flagged, at the cost of having more correctly classified images also flagged. Included is the latent threshold variance as a colour gradient.

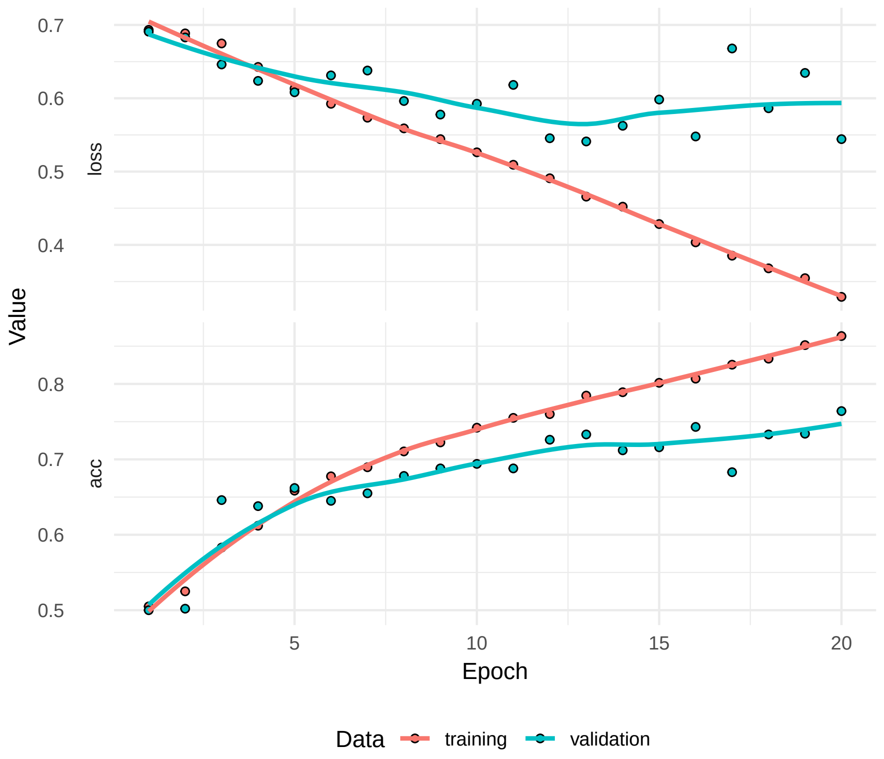
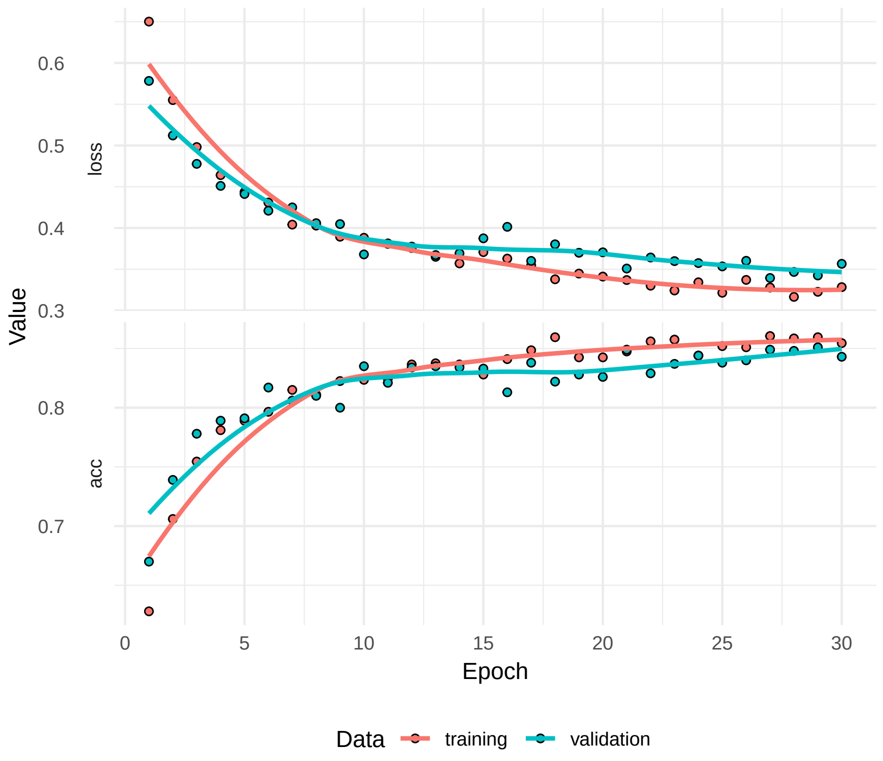
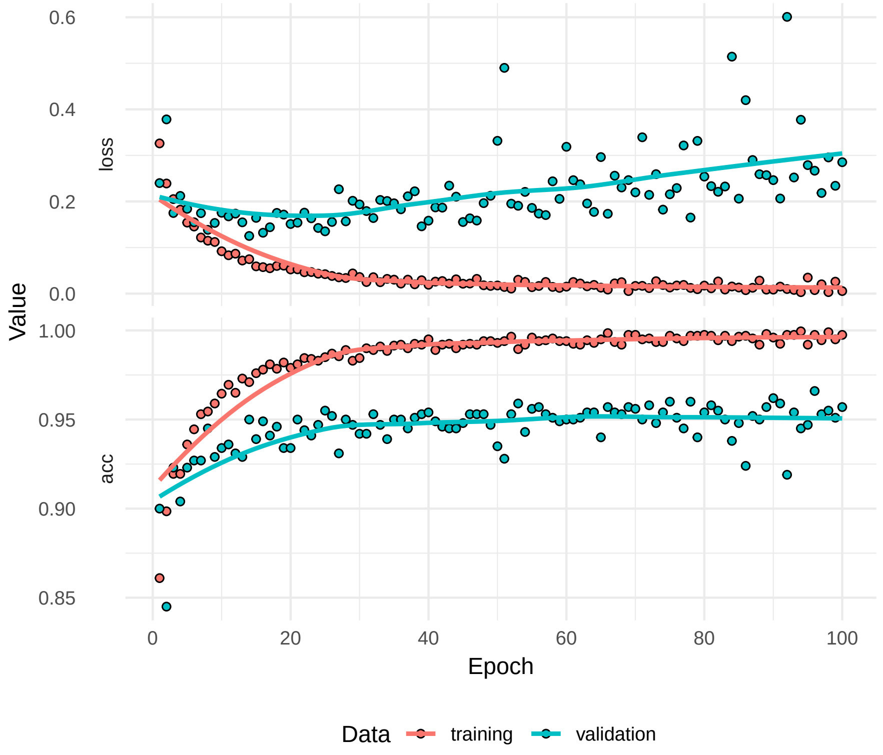


Figure A.2: Plot of accuracy and loss of simple convolutional neural network described in Section 3.1.2, now trained for 20 epochs to aviod overfitting.

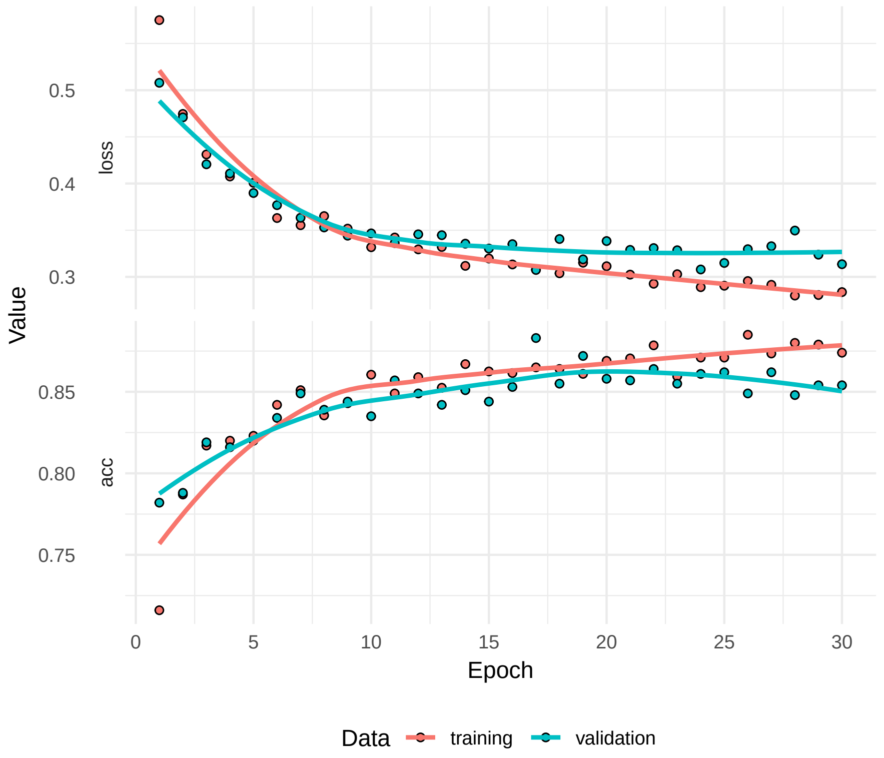


(a) Tuning the densely connected classifier of dropout model.

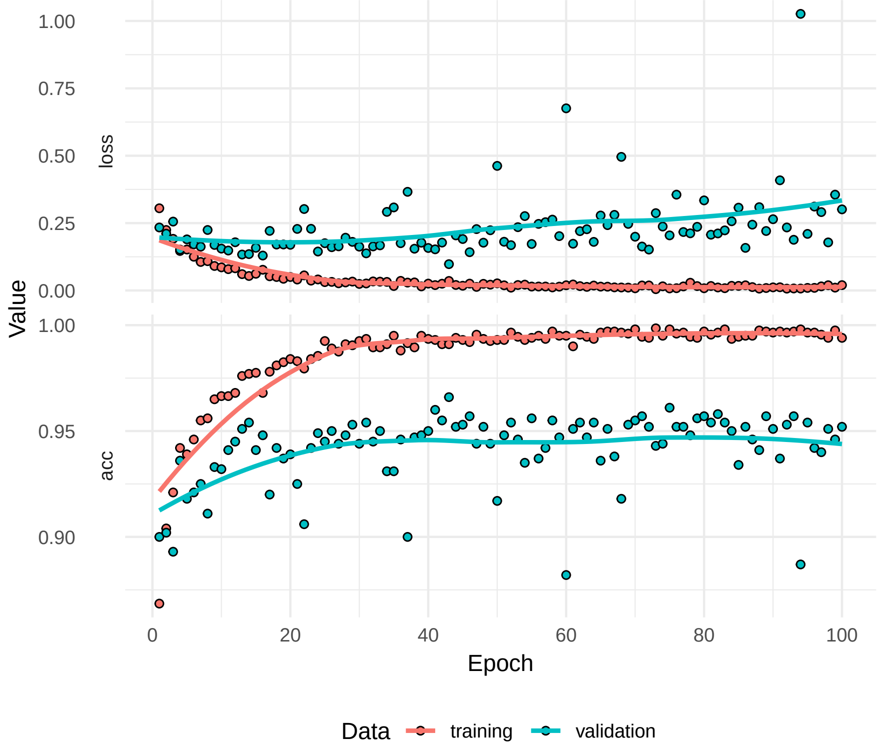


(b) Fine-tuning of the dropout model.

Figure A.3: Training and validation accuracy and loss on the cats and dogs data-set found by tuning a model making use of the VGG 16 convolutional base, dropout, and image augmentation on the training set. The top illustration shows the validation plot for tuning the densely-connected classifier, and the bottom illustration shows the validation plot for fine-tuning from the 5^{th} convolutional layer of the convolutional base.



(a) Tuning the densely connected classifier of non-dropout model



(b) Fine-tuning of the non-dropout model

Figure A.4: Training and validation accuracy and loss on the cats and dogs data-set found by tuning a model making use of the VGG 16 convolutional base, and image augmentation on the training set. The top illustration shows the validation plot for tuning the densely-connected classifier, and the bottom illustration shows the validation plot for fine-tuning from the 5^{th} convolutional layer of the convolutional base.

A.2 Code

All code used in this project can be found at the following GitHub repository:

https://github.com/murrcurt09/ThesisCode

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

François Chollet, Daniel Falbel, JJ Allaire, Yuan Tang, Wouter Van Der Bijl, Martin Studer, and Sigrid Keydana. Keras. `https://keras.io`, 2015.

François Chollet and Joseph J. Allaire. *Deep Learning with R.* Manning, 2018. ISBN 161729554X. URL `https://www.ebook.de/de/product/30981217/joseph_j_allaire_deep_learning_with_r_p1.html`.

G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, dec 1989. doi: 10.1007/bf02551274.

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

Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. Exploring the Landscape of Spatial Robustness. 2017.

Yarin Gal. *Uncertainty in Deep Learning.* PhD thesis, University of Cambridge, 2016.

Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. 2015.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2017. ISBN 0262035618. URL `https://www.ebook.de/de/product/26337726/ian_goodfellow_yoshua_bengio_aaron_courville_deep_learning.html`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving Neural Networks by Preventing Co-adaptation of Feature Detectors. 2012.

Kaggle. Dogs vs. Cats. `https://www.kaggle.com/c/dogs-vs-cats/`, 2013.

Najeeb Khan, Jawad Shah, and Ian Stavness. Bridgeout: Stochastic Bridge Regularization for Deep Neural Networks. 2018.

Zachary C. Lipton. The Mythos of Model Interpretability. 2016.

R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2019. URL `https://www.R-project.org/`.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM Press, 2016. doi: 10.1145/2939672.2939778.

Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *Neural Networks*, 61:85–117, jan 2015. doi: 10.1016/j.neunet.2014.09.003.

Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014.

Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A Survey on Deep Transfer Learning. 2018.

T. Tieleman and G. Hinton. Lecture 6.5 - RMSprop: Divide the Gradient by a Running Average of its Recent Magnitude. 2012.