Uncertainty Calibration for Image Classification Models

Canvern Stevenson Harris
Supervised by Cuong Nguyen

Mathematical Sciences
Durham University

2nd May 2025

Preface

Plagiarism Declaration

This piece of work is a result of my own work and I have complied with the Department's guidance on multiple submission and on the use of AI tools. Material from the work of others not involved in the project has been acknowledged, quotations and paraphrases suitably indicated, and all uses of AI tools have been declared.

Abstract

In the modern world, there is an increasing demand for artificial intelligence and many companies are creating deep and complex models to gain advantages and compete with rivals. Many of these businesses need image classification models to perform advanced image requirement tasks. Whilst, there have been significant improvements in recent years to the accuracies of image classification models, calibration errors have grown severely. This report aims to highlight characteristics affecting the calibration error of image classification models, and it seeks to apply methods to recalibrate model predictions while maintaining high accuracy. In this report, I introduce the basics of machine learning and deep learning, followed by the core topics: image classification, calibration introduction, methods to recalibrate modern image classification model predictions, further calibration influencing factors, and practical applications.

Contents

Chapter 1

Machine learning

1.1 Introduction

Ever since the invention of the first computer, traditional programming has been the standard method to write a program. Traditional programming encodes a precise set of instructions that defines every possible step within the program. Once all instructions are provided and the program is running, inputs such as mouse clicks or keyboard presses are given to the program, and the program responds according to the predefined instructions for that action. While traditional programming should be used when we know the instructions for the program, many programs do not contain clear predefined instructions. For example, the classification of an individual through facial recognition is a difficult program to create since writing clear predefined instructions that distinguish multiple individuals can be a challenging task. This raises the question: how can we create desired programs when the precise set of predefined instructions is unclear? The answer is machine learning.

Machine learning is the development of statistical algorithms where the goal is for computers and machines to learn from data without the need for direct programming. Typically, the data is a collection of inputs or inputs and outputs and is provided into a machine learning algorithm where patterns within the data are discovered. Such patterns capture meaningful information about the data and can be converted into a program for further applications. Machine learning differs from traditional programming since the program is learned through exposure to data rather than predefined instructions. The key paradigms of the two programming methods are illustrated in Figure 1.1.

(a) Traditional programming

(b) Machine learning

Figure 1.1: Comparison between traditional programming and machine learning

Traditional programming is perfect for programs such as calculators, database searching and sorting algorithms because they require a precise set of rules that must be followed to produce the output. Machine learning is necessary for programs such as image classification, determining whether a patient has a disease and weather prediction. These programs need to use patterns identified from previously seen data and apply these patterns to make new predictions for new data. Therefore, machine learning is a necessary tool for programs such as these.

1.2 Supervised learning

1.2.1 Supervised learning introduction

One common approach to machine learning is supervised learning. In supervised learning, we require that for every input, we have a corresponding output. Often the input is represented as a k-dimensional array, where k is an integer greater than or equal to zero. For k = 0, 1, 2, these represent a scalar, vector and matrix respectively, and are all specific types of a k-dimensional array. Sometimes, the input may require other data structures like the rank-k tensor which is a generalisation of the k-dimensional array. The corresponding output is typically stored as a scalar or vector, although many options are available. The input x and its output y are then written as (x, y) and represents one sample.

Example 1.2.1:

$$x_1 = \begin{pmatrix} 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 \end{pmatrix}, y_1 = 1, \quad x_2 = \begin{pmatrix} 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} \end{pmatrix}, y_2 = 4 \quad x_3 = \begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \end{pmatrix}, y_3 = 8$$

Figure 1.2: Examples of inputs and corresponding outputs

The input x, is made up of several values called features. In Figure 1.2 each input has 20 features, each providing information towards its output. Once we have our input and corresponding output, in supervised learning we provide the machine learning algorithm with both of these objects. The machine learning algorithm then studies this input and output and tries to determine the connection between the two. In practice, many samples are provided to the learning algorithm until strong patterns between inputs and associated outputs have been established, after which a program is produced. The program is called a model and is used to predict the outputs of any new provided inputs.

Example 1.2.2: Continuing example 1.2,1, given the following new two dimensional array input, what is the correct output?

$$x = \begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \\ \mathbf{1} & 0 & \mathbf{1} & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \end{pmatrix}, \quad y = ?$$

Figure 1.3: New input with unknown output

Reexamining the inputs and corresponding outputs in example 1.1, we see that if we draw a line between all adjacent ones, we obtain a rough sketch of the number similar to a number represented on a digital clock. Therefore, after observing the given samples, we wish for the model to learn that the input is a simple sketch of the output. We then want the model to accurately apply this thinking to determine the output of the new two-dimensional array input. In this instance, we wish for the model to predict that output y is 6.

The above method demonstrates supervised learning. Provide the model with some samples, learn a model that maps the inputs to the outputs and use the model to make predictions for

new data. In practice, there can be many inputs that all map to the same output. For example, in example 1.1, for x_2, we could shift each number to the left and replace the right-most column with zeros, creating a shifted sketch of the same number.

1.2.2 Training, validation and test data

Training data $\{(x_i, y_i)\}_{i=0}^{n_0-1}$ is a collection of n_0 samples initially provided to the machine learning algorithm where patterns between inputs and corresponding outputs are studied. Once training data has been analysed, a model is produced. Typically, the more training data, the better the produced model because it has more data to learn from.

A hold-out validation set $\{(x_i^{'}, y_i^{'})\}_{i=0}^{n_1-1}$ is a collection of n_1 samples that are held out separate from the training and test data. Often, the validation set is used to fine-tune the model. Validation sets will be required from Chapter 2 onwards.

Test data $\{(x_i^*, y_i^*)\}_{i=0}^{m-1}$ is the data used in the testing phase to test the model's performance. While testing, we only provide the model with the inputs $\{x_i^*\}_{i=0}^{m-1}$, and the model computes predictions $\{\hat{y_i}\}_{i=0}^{m-1}$ for each data input. We can then compare the predicted outputs $\{\hat{y_i}\}_{i=0}^{m-1}$ and the true outputs $\{y_i^*\}_{i=0}^{m-1}$ to see how the model performed.

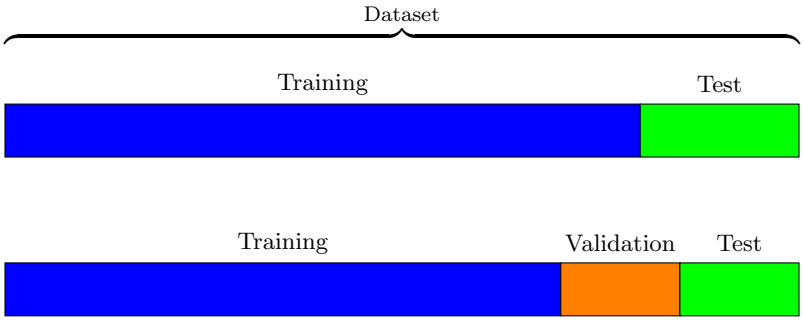


Figure 1.4: Common splits of the dataset

In supervised learning, n samples of data are collected prior to model training by researchers. To train a model we need to split the data into a train, test and potentially a validation set. Common splits of data include 80% for training data and 20% for test data, or 70% for training, 15% for validation, and 15% for testing if a validation set is needed. Assuming n is large enough, this would allow for many training samples to train a good model, yet this would also provide sufficient test data and validation data (if needed) to ensure that the produced model works well in practice. The training, validation and test datasets require that no data overlap otherwise, in the fine-tuning or testing phase, the model may be regurgitating results that it has already seen. This woud prevent us from determining whether the model applies to new unseen data. Also, the n samples must be collected randomly according to some fixed underlying distribution, otherwise, the learning algorithm is bound to struggle to find meaningful patterns.

1.2.3 The mapping function

The goal of machine learning is to learn a function that accurately maps an input $x \in X$ to an output $y \in Y$. For each problem, there is a true underlying mapping function $f : X \to Y$ and is defined by $f(x) = y$ for all samples. To determine the mapping function, a collection of training samples is inputted into a machine learning algorithm. This algorithm studies an input's features and tries to determine the link to its corresponding output. After spotting potential links, the algorithm moves on to other training samples and attempts to establish stronger connections. Sufficient high-quality training data is needed to find strong links in the data.

In practice, anywhere from thousands to millions of different training samples are necessary to learn accurate mapping functions.

The mapping function that is learned during the training phase is an approximation of the true underlying distribution. The learned mapping function can be represented as $\hat{f} : X \to Y$ defined by $\hat{f}(x) = \hat{y}$. During training, the learned mapping function changes as each sample is analysed. Once the training phase is complete, the learned function is finalised and the details are used to produce a model. Once we have learned a function $\hat{f}$, for any new input x we compute $\hat{f}(x) = \hat{y}$ to predict its output. To ensure that $\hat{y}$ is an accurate representation of the true value y, we must learn an approximately correct mapping function for the data.

1.3 Binary classification

1.3.1 Binary classification introduction

Binary classification is a specific type of supervised learning problem where the output is restricted to take only one of two values. Such values can be true or false, yes or no, on or off. To represent these values we let the possible outputs Y be the set $Y = \{0, 1\}$, and each output must take one of these values. In binary classification, the output 0 represents false, whereas 1 represents true. The following is a type of binary classification problem.

Example 1.3.1: At the end of the year students take an official exam to determine their progress throughout the year. In order to prepare for this exam, students take a practice test one month before the official exam. The scores on the practice test range from 0 to 10 with 10 being the high achievable mark. One month later, students take the official exam which is different to the practice test. In the official exam, a student either passes the exam (represented by 1) or they do not (represented by 0). Table 1.1 shows the result for six students from a previous year who went through this process.

	Student					
	1	2	3	4	5	6
Score on practice test	0	3	5	6	8	10
Passed the official exam?	0	0	1	0	1	1

Table 1.1: Binary classification training data

Given a new score, x, on the practice test, how can we predict whether the student passes the official exam?

There are many approaches to solving these kind of problems. A simple approach is to have a threshold input, and when the threshold is exceeded predict 1, otherwise predict 0. One particularly important approach is logistic regression

1.3.2 Logistic regression

One popular method to solving binary classification problems is logistic regression [8]. Logistic regression is a statistical method that fits a sigmoid function to predict the probability of each binary output occurring. The sigmoid function is:

$$\sigma_{sigmoid}(z) = \frac{1}{1 + e^{-z}}. \tag{1.1}$$

This function converts any inputted number, z, to a probability that its output is 1, that is $\sigma_{sigmoid}(z) = P(y = 1|z)$. After using the sigmoid function, we can calculate $P(y = 0|z) =$

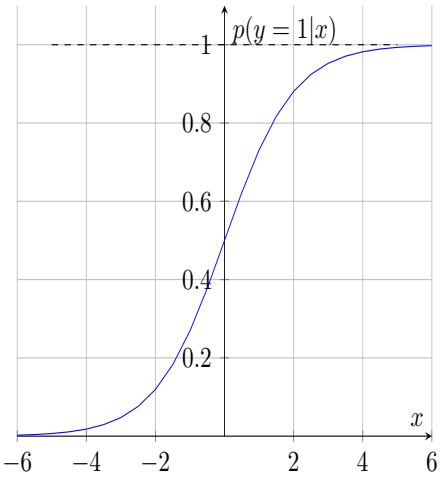


Figure 1.5: Sigmoid plot

$1 - P(y = 1|z)$ to compute the probability that the input's corresponding output takes an output of 0 instead. Figure 1.5 shows the sigmoid function.

Example 1.3.2: Continuing example 1.3.1 to demonstrate logistic regression, suppose we have the training data in Table 1.1. We begin by rewriting the information in Table 1.2 as a supervised learning problem with the provided data as training data. After defining the inputs

	i					
	1	2	3	4	5	6
x_i	0	3	5	6	8	10
y_i	0	0	1	0	1	1

Table 1.2: Supervised learning binary classification training data

and outputs, logistic regression fits a sigmoid function to the data. In particular, the logisitic model is:

$$p(y = 1|x) = \frac{1}{1 + e^{-(B_0 + B_1 x)}}. \tag{1.2}$$

Using the method of maximumd likelihood estimation, we get $B_0 = -4.8766$ and $B_1 = 0.8895$. Therefore the model is $p(y = 1|x) = \frac{1}{1+e^{-(-4.8766+0.8895x)}}$. We can use this to determine the predicted outputs and probabilities of each output for subsequent test data.

The term "regression" in logistic regression refers to the continuous probabilities that is produced by the logistic model. In Figure 1.6 the y-axis represents the model's probability that a student passes the official exam given a score of x in the practice exam, $p(y = 1|x)$. Now, we can use the logistic regression model to compute the probability that a new student passes the exam. For example, now a new student has achieved a score of 7 in the practice test. Using the model we compute $p(y = 1|x = 7) = 0.7941$, and this is shown by the dashed line in Figure 1.6. Also, $p(y = 0|x = 7) = 1 - 0.7941$, hence $p(y = 0|x = 7) = 0.2059$. Calculating these probabilities gives us an indication for how well the student might do, and we could combine this with an intuitive threshold probability of 0.5, to determine whether we classify a student as predicted to pass the exam or not.

The above example was an example of simple logistic regression. Multilogistic regression is used for binary classification when multiple inputs are used for each sample. The multiple inputs are

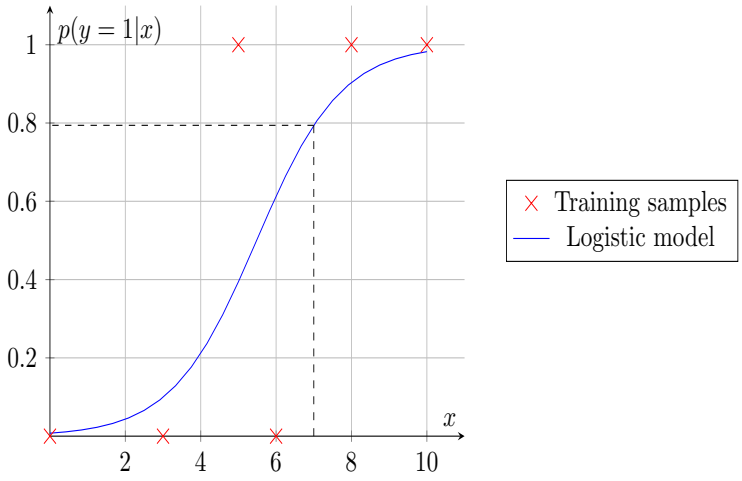


Figure 1.6: Logistic regression for example 1.3.2

stored in a vector and correspond to a binary output. An assumption of logistic regression is that the samples observed are independent. Another assumption of simple logistic regression is that the relationship between the log odds of the output and the inputs are linear.

1.4 Measuring performance

In machine learning classification problems, once training phase is complete and a model has been trained, we test the model on the set of test data $\{x_i^*, y_i^*\}_{i=0}^{m-1}$. The accuracy of the model on the test set is

$$\text{Accuracy} = \frac{1}{m} \sum_{i=0}^{m-1} \mathbb{1}(\hat{y}_i = y_i^*), \tag{1.3}$$

where $\hat{y}_i$ is the predicted value of y_i^* for a test input. The main focus in classification supervised machine learning is to achieve high accuracy in the test set. If the model performs well in the test set, we say that the model generalises well to unseen data. This is the desired property for many companies since they want models that will work in real world applications.

Chapter 2

Deep learning

2.1 Neural networks

2.1.1 The neural network

Deep learning [15] is a subset of machine learning that uses artificial neural networks to extract high-level features from data. A neural network is a model made up of many layers used to find meaningful patterns within the data. Neural networks consist of input, hidden and output layers. The input layer contains the raw input data allowing the data to be provided ready for processing. The hidden layers perform complex computations of the data identifying the patterns. The output layer is where the final predictions of the output are produced. In neural networks, each layer consists of nodes which receive data as inputs, process data or learn patterns, and pass an output to nodes in the next layer. Figure 2.1 illustrates a neural network.

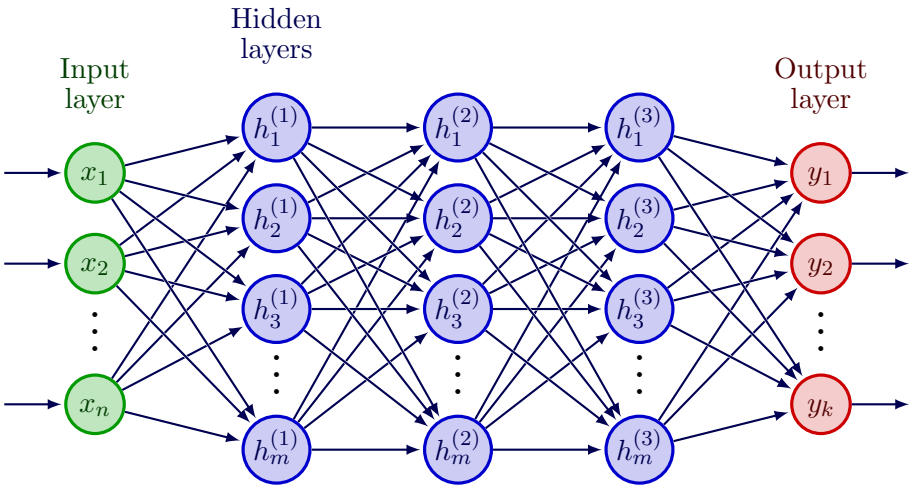


Figure 2.1: Neural network architecture

When an input is provided to the deep learning model, the input data goes through many hidden layers and many transformations within these layers to determine the output. In the hidden layers, a node will receive inputs from all nodes in the previous layer. The node assigns a weight for each received input and applies these to compute the weighted sum of the inputs. A bias is then added to the weighted sum and is called the pre-activation value which is then passed through a non-linear transformation function called the activation function. The result of the activation function is passed to all nodes in the next layer. This method is the forward propagation algorithm and is outlined in Algorithm 1.

Before proceeding with the forward propagation algorithm, in the hidden layers and output

layer, in each node, we must set the initial weights for all nodes in the previous layer as well as a bias for the node itself. For example, in Figure 2.1, in node $h_1^{(2)}$ we set m weights for the nodes in the first hidden layer and a bias for the node itself. Similarly, in node $h_2^{(2)}$, we set m different weights for the nodes in the first hidden layer and again set a bias for the node. We repeat this for all nodes in the hidden and output layers, setting different weights for the previous layer and different biases for the node itself. Once the weights and biases are set, we can use the forward propagation algorithm 1 to convert inputs into predicted outputs. In algorithm 1, $x_j^{(i)}$, $\alpha_j^{(i)}$ and $\sigma_j^{(i)}(.)$ represent the final value of the node, the preactivation value and the activation function respectively for node j in layer i. Also, $w_1^{(j,i)}, \ldots, w_m^{(j,i)}$ represent the assigned weights of every node in layer $i-1$, for node j in layer i. Also, $b^{(j,i)}$ represents the bias for node j in layer i.

Algorithm 1 Forward propagation algorithm

procedure FORWARD_PROPAGATION($[x_1, \ldots, x_n]$)
  Set $x_1^{(0)} = x_1$, $x_2^{(0)} = x_2$, ..., $x_n^{(0)} = x_n$
  for $i = 1, 2, 3, 4$ **do** ▷ Iterate layers
    Set m as the number of nodes in layer $i-1$
    Set k as the number of nodes in layer i
    for $j = 1, \ldots, k$ **do** ▷ Iterate nodes in layer
      Compute $\alpha_j^{(i)} = w_1^{(j,i)} x_1 + \cdots + w_m^{(j,i)} x_m + b^{(j,i)}$ ▷ Weighted sum and bias
      Compute $x_j^{(i)} = \sigma_j^{(i)}(\alpha_j^{(i)})$ ▷ Apply activation function
    end for
  end for
  return $x_1^{(4)}, \ldots, x_k^{(4)}$ ▷ Final outputs
end procedure

The forward propagation algorithm 1 converts inputs into outputs for the neural network shown in Figure 2.1. In forward propagation, the purpose of the hidden layers is to extract high-level features of the data. In this sense, high-level features do not refer to the individual elements of the input but rather they refer to advanced characteristics of the data that would be difficult to spot with basic methods. Activation functions within the hidden layers apply non-linear transformations enabling the discovery of these high-level features. Typically, within each layer, the same activation function is applied across the nodes. Some popular examples of the activation functions include ReLu, $\sigma_{ReLu}(x) = \max\{0, x\}$ [18], tanh, $\sigma_{tanh}(x) = \tanh(x)$ and sigmoid, $\sigma_{sigmoid}(x) = \frac{1}{1+e^{-x}}$ which we saw in 1.1.

To represent the transformation of data between layers we use matrices. At a given layer, each node within that layer has a bias and it assigns a weight for all nodes in the previous layer. Denote $\underline{w_i} = (b_i, w_{i,1}, \ldots, w_{i,n})$, where b_i is the bias and $w_{i,1}, \ldots, w_{i,n}$ are the weights for node i. The inputs to each node in a given layer are the values of the nodes in the previous layer. Denote $\underline{x} = (1, x_1, \ldots, x_n)$, where 1 represents a placeholder term for the bias and $x_1, \ldots, x_n$ are the values of the nodes in the previous layer. Then the pre-activation value for node i in the given layer is given by $\alpha_i = \underline{w_i}^\intercal \underline{x}$. Furthermore, if we assume there are m nodes in the given layer, then we can write $W = (\underline{w_1}, \ldots, \underline{w_m})^\intercal$ and compute the pre-activation values of the whole layer as $\underline{\alpha} = W\underline{x}$. Finally, since the activation function used within a layer is the same for every node in that layer, by writing $\sigma^{(1)}, \ldots, \sigma^{(4)}$ as the activation functions for the layers and $W^{(1)}, \ldots, W^{(4)}$ for the matrices of weights and biases for each layer, then the raw inputs in the example neural network in Figure 2.1, go through the collection of transformations:

$$(\sigma^{(4)} \circ W^{(4)} \circ \sigma^{(3)} \circ W^{(3)} \circ \sigma^{(2)} \circ W^{(2)} \circ \sigma^{(1)} \circ W^{(1)})(\underline{x}) = \underline{y}. \tag{2.1}$$

The weights (and biases) and activation functions in neural networks act as the mapping function between the input and output. The activation functions are chosen specific to the desired neural network architecture. During model training, the weights (and biases) are free to adapt and change as data is learned, allowing the network to learn from its mistakes. The weights encapsulate the weights and the bias and are shortened to weights for simplicity.

The inspiration behind the neural network is the human brain. Each node is a neuron and the idea behind the neural network is to simulate an artificial brain. Neural networks have shown significant improvements in performance compared to traditional machine learning methods. Neural networks serve as a general structure for computers to learn from experience and correct previous errors. The word "deep" refers to the many layers in the hidden layer that transform the data. The depth of the model is the number of layers in the hidden layer of the neural network.

2.1.2 Softmax function

Multi-class classification is an extension of binary classification that distinguishes objects between three or more possible outputs. The set $Y = \{y_{(1)}, \ldots, y_{(k)}\}$ stores the possible outputs where each output represents one class. The true class for a sample (x_i, y_i) is the class defined by output y_i. Every input must belong to one of the classes and the goal of classification problems is to determine which class an input belongs to. Neural networks are strong deep learning algorithms used in classification problems. The output layer of a neural network has k nodes, with one node representing each class. In practice, the output layer does not produce a single predicted class for an input, but rather it predicts the probabilities that an input belongs to each class. In the output layer, probabilities of each class are calculated using the softmax function [2]:

$$\sigma_{SM}(\underline{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}. \tag{2.2}$$

Here, $\sigma_{SM}(.)$ represents the softmax function, $\underline{z}$ represents the vector of pre-activation values of each node in the output layer and $\sigma_{SM}(.)_i$ and z_i represent the ith element of the softmax function and the ith element of the preactivation vector respectively. The preactivation vector in the final layer is also called the logit vector and loosely provides numeric representations of the probabilities of the classes before softmax is applied. The softmax function is a non-linear activation function applied in the output layer to convert the logit vector into a probability distribution for comparisons between the neural network's calculations for the chances of the classes. The class with the highest probability is the predicted class for the input. Some alternatives for the final layer in binary and multi-class classification use $k-1$ nodes and compute the probability of the final class by subtracting the total of the $k-1$ nodes from one. For example, the sigmoid activation function uses one node to calculate the probability of one class, and the probability for the other class can be found by subtracting this probability from one.

The non-linear activation functions make neural networks well suited for problems beyond linear problems, making significant progress on previous machine learning structures. In classification problems with k classes where we wish to predict which class an input falls into, the final layer of the neural network has one node for each class. In this case, the outputted values in the output layer are the probabilities that the input belongs to each class. The class with the highest probability is then the predicted class for the input.

2.1.3 Loss function

Ensuring a deep learning model learns a mapping function between inputs and outputs that is typically correct is critical for producing high-accuracy models. The loss function is a mathematical function that measures how well a model performs. This function quantifies the error

between a model's prediction and its true output, with higher values indicating a larger error in the model's prediction. In multi-class classification, the outputted probabilities for each class are used to determine the contribution to the total loss for each training sample. One popular loss function for classification problems is the cross-entropy loss function [5].

Example: For a training set $\{x_i, y_i\}_{i=0}^{n_0-1}$, let p_i be the predicted probability by the model for the true class for sample i. Then the cross-entropy loss, also called the negative log likelihood (NLL), for the training set is

$$l\left(\{(x_i, y_i)\}_{i=0}^{n_0-1}; p_0, \ldots, p_{n_0-1}\right) = -\sum_{i=0}^{n_0-1} \log(p_i). \tag{2.3}$$

Each p_i is a probability between zero and one, therefore the loss contributed by each term, $-\log(p_i)$, increases as the probability p_i decreases, and hence our goal is to have p_i as close to one as possible. If a model correctly predicts a class with probability equal to one then the prediction is perfect and no loss is contributed to the loss function. On the other hand, if a model predicts that an input belongs to the true class with probability zero then the loss contributed can be infinite. While the softmax function does not produce exact values of zero or one, probabilities can become within small intervals of these numbers. Note that the idea of the loss function is to create large values when the predicted value is incorrect and to give lower values when the model's predicted value is correct. Therefore, the model can only minimize the loss function when most or all of the predictions are correct, leading to a high accuracy on the training data. Minimizing the loss function over the training set is the core process of training neural networks.

Recall that the mapping function is a learned function that maps inputs to outputs. Extending 2.1, in neural networks of depth d, the mapping function is a collection of transformations:

$$(\sigma^{(d+1)} \circ W^{(d+1)} \circ \sigma^{(d)} \circ W^{(d)} \circ \cdots \circ \sigma^{(1)} \circ W^{(1)})(\underline{x}) = \underline{\hat{p}}, \tag{2.4}$$

where $\sigma^{(1)}, \ldots, \sigma^{(d)}$ and $W^{(1)}, \ldots, W^{(d)}$ are the activation functions and the matrices of weights for the hidden layers, $\sigma^{(d+1)}$ and $W^{(d+1)}$ is the activation function and matrix of weights for the output layer and $\underline{\hat{p}} = (p_{(1)}, \ldots, p_{(k)})$ is the predicted probabilities by the neural network for the respective classes $\{y_{(1)}, \ldots, y_{(k)}\}$ for input $\underline{x}$. Defining:

$$\hat{f}(x; \underline{\sigma}, \underline{W}) = (\sigma^{(d+1)} \circ W^{(d+1)} \circ \sigma^{(d)} \circ W^{(d)} \circ \cdots \circ \sigma^{(1)} \circ W^{(1)})(\underline{x}), \tag{2.5}$$

where $\underline{\sigma} = (\sigma^{(1)}, \ldots, \sigma^{(d+1)})$ and $\underline{W} = (W^{(1)}, \ldots, W^{(d+1)}$ allows us to write the mapping function between the neural network's inputs and the outputted probabilities for each class as $\hat{f}(x; \underline{\sigma}, \underline{W}) = \underline{\hat{p}}$. Neither $\underline{\sigma}$ or $\underline{W}$ depend on the sample, therefore for each training sample (x_i, y_i) we have

$$\hat{f}(x_i; \underline{\sigma}, \underline{W}) = \underline{\hat{p}_i}. \tag{2.6}$$

Since we have defined p_i to be the predicted probability by the model for the true class for sample i, and $\underline{\hat{p}_i}$ is the predicted probabilities by the model for all classes for sample i, p_i is an element of $\underline{\hat{p}_i}$. Let t_i be the true class of sample i such that $\underline{\hat{p}_i}_{(t_i)} = p_i$ or equivalently

$$\hat{f}(x_i; \underline{\sigma}, \underline{W})_{(t_i)} = p_i. \tag{2.7}$$

Then we can rewrite the cross-entropy loss in 2.3 as a function of the weights and activation functions as:

$$l(\underline{x}, \underline{t}; \underline{W}, \underline{\sigma}) = -\sum_{i=0}^{n_0-1} \log\left(\hat{f}(x_i; \underline{\sigma}, \underline{W})_{(t_i)}\right), \tag{2.8}$$

where $\underline{t} = (t_0, \ldots, t_{n_0-1})$ represents the true classes of the samples of the training data.

Expressing the loss function as a function in the form of 2.8 is important because it demonstrates that the total loss is dependent on the input, the activation functions and the weights. Therefore, improving the loss function requires addressing these three items. The inputs are predetermined by the available samples in the training data although different splits of data and different versions of data can affect the results. The activation functions are chosen specific to the desired neural network architecture and different functions can have different effects. While inputs and activations play a role in the model's performance, the main method to minimise the loss function is by adapting and learning optimal weights of the neural network. Optimal weights allow for high understandings of complex relationships in the data and producing better results. If the neural network contains poor weights, then no understandings of the data can be made and the whole neural network performs poorly. Furthermore by seeing the mapping function as in 2.7, then from 2.8, we can see that minimizing the cross entropy loss can be equivalently achieved by producing a more accurate mapping function that produces values of the predicted probability for the true class, p_i, as close to one as possible.

2.1.4 Stochastic gradient descent and Adam

Assuming we have a training set, a neural network where the number of nodes, layers and activation functions are specified, and a loss function, we are ready to find an accurate mapping function between the inputs and outputs. To find an accurate mapping function, we need to minimize the loss function. Since we have chosen the data that we wish to learn information about, and we have chosen suitable activation functions that we think will help us find an accurate mapping function, we now need to fine-tune and experiment with changing the weights of the previous layer in each node in the hidden and output layers of the neural network. One example is to set all of the weights to be random and then calculate the value of the loss function over all the training samples. After calculating the total loss for this set of weights, choose one node in the hidden or output layer and in this node subtly change the weights for the previous layers. After subtly changing the weights in a specific direction, calculate the total loss for the training set again using the new weights. Since the calculations of the loss function depend on the weights of the neural network, the value of the loss function for the new weights should be different. If the total loss has decreased then, you have successfully improved the mapping function between the inputs and outputs, and by predicting an output by selecting the class with the highest probability, we expect accuracy across the training set to increase. Alternatively, if the total loss for the new weights has increased, then the model has performed worse and the weights should be either returned to the original position or moved in the opposite direction. This process is repeated for all nodes. The collection of weights that achieve the lowest possible total loss on the training data are called the optimal weights for the neural network.

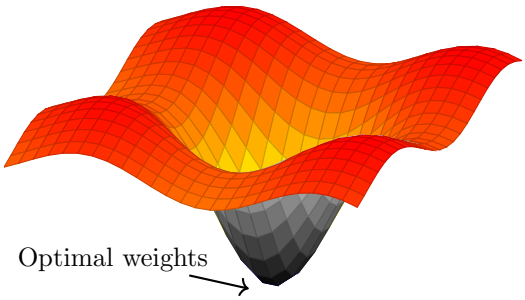


Figure 2.2: Optimizing a neural network

One commonly used optimization method that attempts to find the optimal weights is called

stochastic gradient descent (SGD). Begin by selecting a small random batch of n training samples, $\{x_i, y_i\}_{i=0}^{n}$, where n is a positive integer less than the number of training samples. Next choose the loss function for a neural network, predefined activation functions and the random batch of small data samples as $l\left(\underline{W}; \underline{\sigma}, \{x_i, t_i\}_{i=0}^{n}\right)$, where t_i is the true class of sample i from the batch. The idea of stochastic gradient descent is, for the first small random batch of training data, calculate the derivative of the loss function with respect to weights : $\frac{\partial}{\partial \underline{W}} l\left(\underline{W}; \underline{\sigma}, \{x_i, t_i\}_{i=0}^{n}\right)$ and then depending on the sign, shift the weights $\underline{W}$ such that the loss function for the small batch of data moves down the slope towards its minimum value. Once the weights have been shifted, collect a new random batch of the same size excluding data from any previous batches and again calculate the derivative of the loss function with respect to the weights for the new batch and shift the weights. Repeat this process, until all data in the training data set has been used. Once all data has been used, we say that one epoch has been completed. A new epoch can be started by taking a new initial random batch from the training data and repeating the whole process again. By performing such shifts of the weights, all nodes in all layers are trained simultaneously enabling the discovery of optimal weights in an efficient manner. In neural networks consisting of many layers and nodes, optimising the weights can be difficult and computationally expensive, however, stochastic gradient descent can speed up the process. Stochastic gradient descent aims to find the best collection of weights that minimizes the loss function's error between the predictions and true outputs. We also define η_t as the learning rate at epoch t. The learning rate determines how much we shift towards the optimal values of the weights. Smaller values of η_t leads to more precise measurements in the optimal weights, however, these smaller values may require more epochs to converge since the shifts will be smaller. Large values of η_t may fail to converge to optimal weights smoothly, and may overjump the optimal weights. Often the learning rate is gradually decreased to ensure smooth convergence of the weights to their optimum values.

Figure 2.2 depicts an example of the loss function for a neural network with two input nodes connected to one output node. The weights of this network then are the weights for the two inputs nodes in the output node. The z-axis represents the loss function and the x and y axis represent the weights of the first and second input node for the output node respectively. As the weights are explored the total loss for a fixed training set changes. Throughout exploration of the weights there are many local minimums, where the neural network may incorrectly think it has found the optimal weights. The true optimal weights is the global minimum, representing the weights that achieve the minimum total loss for the data. Avoiding being stuck in local minimums that is not the global minimum is critical to ensuring an accurate mapping function is found. Methods such as momentum in stochastic gradient descent apply inertia to ensure networks escape local minimums. While modeling the total loss as a function of the weights for many nodes requires sketches in many dimension, the ideas of finding the optimal weights remain the same.

The backward propagation algorithm 2 describes the method used in optimizing a neural network. To begin backward propagation, the inital weights are typically randomized. Stochastic gradient descent uses backward propagation and is a core process to optimize and find suitable mapping functions in deep learning.

Adam [9] is a specific type of stochastic gradient descent that changes the learning rate based on the magnitude of the gradients. Adam short for adaptive moment estimation typically converges faster than standard stochastic gradient descent because of the adaptive learning rates. In Chapter 3 and 4 of this report we will use both optimisers to see how they compare. Although they are different optimisers, the process of shifting down the gradient to find suitable weights is the same in each.

Algorithm 2 Backward propagation algorithm

1: **procedure** BACKWARD_PROPAGATION($[x_1, \ldots, x_n], l, \underline{W}, \underline{\sigma}$)
2: Produce a forward propagation for an input
3: Compute the value of the loss between the predicted probability and the true value
4: Starting at the output layer and ending at the input layers, compute the gradient of the loss function with respect to the weight of each node
5: Update the weights of each nodes by shifting the weight in the direction opposite to the gradients calculated in step 4
6: **return** $\underline{W}$
7: **end procedure**

2.2 Overfitting and underfitting

2.2.1 Overfitting

Recall that the model needs to learn a function $\hat{f}(x_i; \underline{\sigma}, \underline{W})$ that converts inputs into outputs accurately. It does so using the training data $\{(\underline{x_i}, y_i)\}_{i=0}^{n_0-1}$ and the aim is to minimise a specified loss function on this training data. After training the model using an optimization algorithm such as stochastic gradient descent, we may achieve near-perfect accuracy on the training data and the loss function may be extremely low. However, the key and whole purpose of the machine and deep learning is to achieve high accuracy on unknown outcomes, therefore, the model must performs well on the test set. If the model finds a mapping function that accurately converts training inputs into their corresponding outputs, but, the same mapping function fails to predict the true outputs for the test inputs we say that the model has overfit the training data.
Identifying overfitting in real applications is simple. When the loss of the training set is descreasing, yet the loss of the test set is increasing, then this is a key indication of overfitting. For example, if the training loss reaches a value of 0.0005 but the loss for the test set is increasing then it is likely that overfitting has occurred. A common cause of overfitting is due to an inadequate amount of training data. Inadequate amounts of data for the model lead to minimal meaningful patterns discovered. Another common reason for overfitting is when the model memorises the training data rather than understanding its structural patterns. This can occur when the model is too complex and there are too many layers and nodes in the neural network.

On the other hand, models may underfit the training data. This may occur when the neural network used to find the mapping function is too basic to extract complex relationships within the data. Other reasons for discrepancies in training and test losses and accuracies include: the data sets being preprocessed differently, the training data or test data containing imbalanced classes, or the optimisation algorithm becoming stuck in a local minimum. This typically leads to poor training and test accuracy and can be overcome by using a different configurations of models or by using different initial random weights in the nodes of the neural network. Taking steps to ensure that the models achieve low loss and high accuracy on the test set is critical for ensuring a model is reliable for predictions of new inputs.

2.2.2 Regularisation

One method to prevent overfitting is regularisation sometimes called weight decay. Regularisation adds a penalizing term to the loss function that involves the magnitude of the weights.

Ridge regression [7] also known as Tikhonov regularization (named after Andrey Tikhonov) is a popular method to reduce overfitting. It works by adding a term to the loss function that is

related to the weights of the learning algorithm in the neural network. The ridge regularization term $\lambda \sum_{j=1}^{q} {w_j}_2^2$ is added to the loss function:

$$l(\{\underline{p_i}\}_{i=0}^{n_0-1})_{new} = l(\{\underline{p_i}\}_{i=0}^{n_0-1} + \lambda \sum_{j=1}^{q} {w_j}^2,$$

where we write the loss function depending on the output probabilites as $l(\{\underline{p_i}\}_{i=0}^{n_0-1})$. This ridge regression term is also called the l_2 regularization because it imposes a term the square of the coefficients of weights to be minimized alongside the loss function. The term λ can be chosen depending how much regularization we wish to impose. With this new loss function, the goal is to again minimize $l(\{\underline{p_i}\}_{i=0}^{n_0-1})_{new}$.

Lasso regression is similar to ridge regression, however, the lasso regularization term $\lambda \sum_{j=1}^{q} |w_j|$ is used:

$$l(\{\underline{p_i}\}_{i=0}^{n_0-1})_{new} = l(\{\underline{p_i}\}_{i=0}^{n_0-1} + \lambda \sum_{j=1}^{q} |w_j|.$$

This is called l_1 regularization because the loss function is minimized with an additional cost function that is the l_1 norm of the weight vector. In practice regularization can be a common tool to prevent overfitting, and this is just one of many techniques that can be used to improve the generalisation of a model.

2.2.3 Cross validation

In machine and deep learning, we need to train and test the model. Often trained models are not ready for testing and need to be fine-tuned potentially reducing overfitting, before being fully effective. Cross-validation is a technique that fine-tunes the model whilst being efficient with data.

One validation method is cross-validation. Cross-validation is a technique that is used to evaluate and fine-tune a model. It is also used when there is minimal data. Cross-validation determines whether the model overfits or underfits the data, and determines if the model generalizes well. The idea is:

1. Take all the training data and split it into train and validation data.
2. Train the model on the training data and then validate the model on the validation data, to obtain the accuracy and properties of the model on the validation set.
3. Repeat steps 1 and 2 with different train and validation datasets many times.
4. Average the results of the validation sets' accuracies and properties to understand the model's performance.

K-fold cross-validation is a simple technique that can have significantly impressive results. A variation of this works by creating a base model using all of the training dataset. Then create a new model using $k-1$ folds for training, leaving one fold for validation and use the validation set to measure its performance. Now, use a different set of $k-1$ folds, train a new model, and evaluate this new model with the remaining fold. Repeating this k times using each fold as a validation set once, can give insights into how the base model will perform without explicitly using the base model. Cross-validation is a simple technique that in practice can fine-tune the model ensuring the model performs well in real applications.

Chapter 3

Image classification

3.1 Image classification

3.1.1 Image classification introduction

Image classification [16]is a type of problem in deep learning that challenges the computer's ability to assign an image into one of at least two different categories. To train an image classification model, an image training dataset, a neural network with a specific structure and a loss function designed for classification are required. During training, optimization algorithms find an appropriate mapping function and well-positioned weights for the neural network. Once the model has been trained, the model is evaluated on test images. If the model achieves high accuracy, then the model can be used or adapted to real-life applications such as autonomous driving, identifying diseases based on medical X-rays or classifying faulty products in manufacturing. Furthermore, the ideas in image classification are required for AI to navigate environments accurately. These tasks enable AI to have computer vision.

Image classification or, closely related, image recognition plays a foundational role in computer vision and forms the basis of computer vision problems. One widely applicable use of image recognition is facial recognition which is regularly used for verification of an individual in banks. Another benefit of image recognition is in education. For many people with reading problems or physical disabilities affecting their education, the model can provide methods such as text-to-speech or image-to-speech to assist patients in their learning. Furthermore, the concepts behind image classification can be expanded to further deep learning algorithms, and the upcoming ideas in image classification are closely related to many different techniques in deep learning.

3.1.2 Preprocessing

Image classification models use neural networks to classify images into one of at least two categories. Before any weight optimization processes occur, images must be broken down into a collection of individual elements for inputting into the input layer of the neural network. Pixels are the building blocks of images and every image can be broken down into its pixels. In grayscale images, each pixel is represented by an integer in $[0, 255]$ specifying the pixel's light intensity. The full image can then be defined as a matrix where the rows represent the pixel values along the width and the columns represent the pixel values along the height. Grayscale images require different shades of black, grey and white to outline different features within the image.

In images containing colour, each pixel is split into three smaller subpixels, where one subpixel specifies the intensity of red, another for green and a final one for blue. The specific order of red, green, then blue of the subpixels is most common and this is shortened to RGB. Each subpixel is

an integer in $[0, 255]$ defining the intensity of the subpixel's colour. Together, the three subpixels create a pixel displayed as one colour. Positioning many pixels, made up of smaller subpixels, close together can make images that appear to have a range of colours.

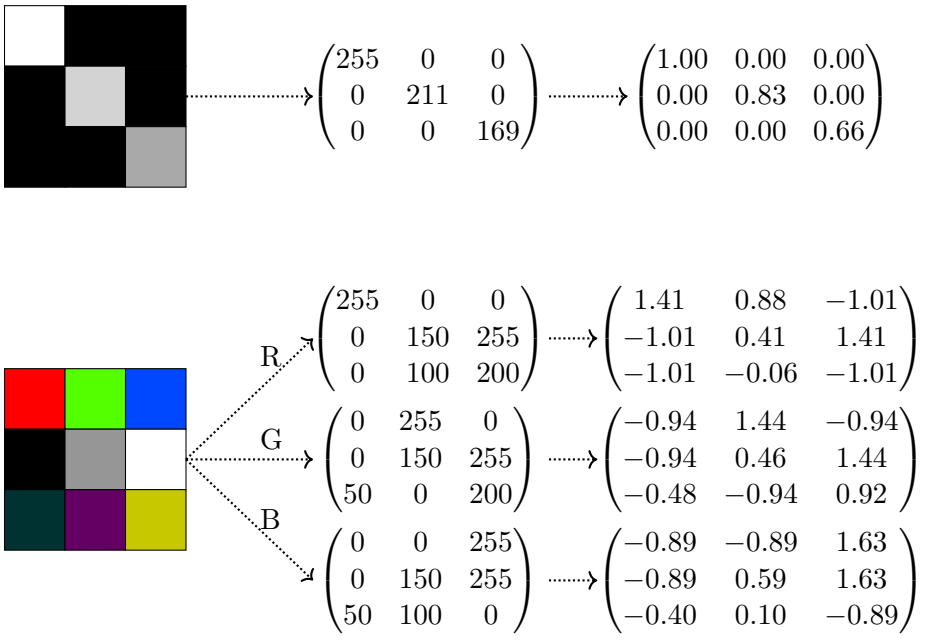


Figure 3.1: Example of image pixel data and data preprocessing

The higher the number of pixels in the width and height of the image, the higher the quality of the image. The total number of pixels in an image is the number of pixels in the width multiplied by the number of pixels in the height. While grayscale images can be represented as a single matrix, coloured images require a third dimension. This allows a coloured image to be represented as a three-dimensional array, where one dimension is the rows of the pixels, another is the columns of the pixels, and the final one is the three channels of colour, where each element is an integer between 0 and 255.

By considering a matrix as a special type of three-dimensional array, the three-dimensional array of either a greyscale or RGB image is then the input that will be input into a neural network. Before putting the image into the input layer, preprocessing of the data can occur. One popular preprocessing method is normalisation. Normalisation is the process of ensuring all inputs fall within a specified range. One common range is to transform the inputs into a value between 0 and 1. This can be achieved by dividing all inputs by 255 before inputting into the model.

Another method of preprocessing an image is standardisation. Standardisation involves transforming the values in the three-dimensional array such that the mean is zero and the standard deviation is one. One method to achieve this is to subtract the mean of all values in the array and then divide it by the standard deviation of all values of the array. Another method is to subtract the channel's mean and divide by the channel's standard deviation separately for each of the respective three coloured channels. Sometimes the mean and standard deviation in each channel are significantly different and standardising by each channel has greater effects for the model.

Centering is another preprocessing method that simply subtracts the mean of the inputs. This can be done using the mean of all the data, or by subtracting the mean of each channel for each

respective channel. The benefit of centering is that it removes biases of different features by centering all data at zero. Centering is a technique that has been applied to the data for many successful image classification models [6] [12] [21].

One benefit of preprocessing the data is the reduction in the risk of the gradients of the loss function either being zero and gaining no progress, or skyrocketing to infinite. Dealing with large, all positive values in the range $[0, 255]$ can create large gradients that lead to difficult optimization of the weights. By ensuring the mean of the data is zero, or the data is in a smaller range, the chances of the gradient of the loss function being abnormally high decreases, leading to a consistent learning rate and efficient convergence of the weights to their optimal positions. Once a preprocessing method has been chosen it must be applied to every training and test data sample to ensure consistent results across different images. Figure 3.1 illustrates the pixel data for a greyscale and RGB image, followed by normalisation by all values and standardisation by the values of each channel for the respective images. By preprocessing the data to ensure that all values are in a fair range, we ensure that no one feature dominates during the training phase. Furthermore, it ensures that the convergences of the weights towards the optimal values are steady and that no significant new problems arise due to an unfair spread of the data. In practice, several preprocessing techniques may tested, and the one that leads to a higher accuracy on the test set is the one that should be used for the specific problem.

3.2 The convolutional neural network

3.2.1 Convolutional neural networks and feature maps

The convolutional neural network (CNN) [13] is a specific type of neural network designed for image classification. Rather than treating each pixel value in an image separately, it combines pixels and their surrounding values to determine the structural patterns visible in the image. Convolutional neural networks contain an input layer, an output layer and many hidden layers in between. The hidden layers extract key features in the image and detect key properties of the image that will help the model identify any objects located in the picture that has been provided. The convolutional neural network has formed the basis of many image classification models and its design has played one of the most important roles in allowing models to see images. The models that are used later in this report all make use of a convolutional neural network, each having slightly different structures. The convolutional neural network is the base model for some of the most successful AI applications that involve seeing images [6] [21]. Figure 3.2 shows the structure of one of the first and most influential convolutional neural networks, the LeNet-5 [14]. The LeNet-5 is considered a basic example of a simple convolutional network.

One key difference of a convolutional neural network is the inputs and outputs of each layer. Typically most neural networks have inputs and outputs of each layer as a collection of individual values that are input into the model as one input vector, however, images are two-dimensional objects that can be represented as a three-dimensional array and by inputting only one pixel at a time key features described by many surrounding pixels may be lost. Convolutional neural networks change this by allowing the inputs and outputs to be collections of two-dimensional arrays, storing the whole collection as a three-dimensional array. The inputs and outputs of the layers of convolutional networks are called feature maps. A feature map is a two-dimensional array with a specific width and height where its values represent the patterns found for the inputted image. By inputting and outputting collections of feature maps the different structures of the originally inputted image can be passed through the whole convolutional neural network in different three-dimensional forms.

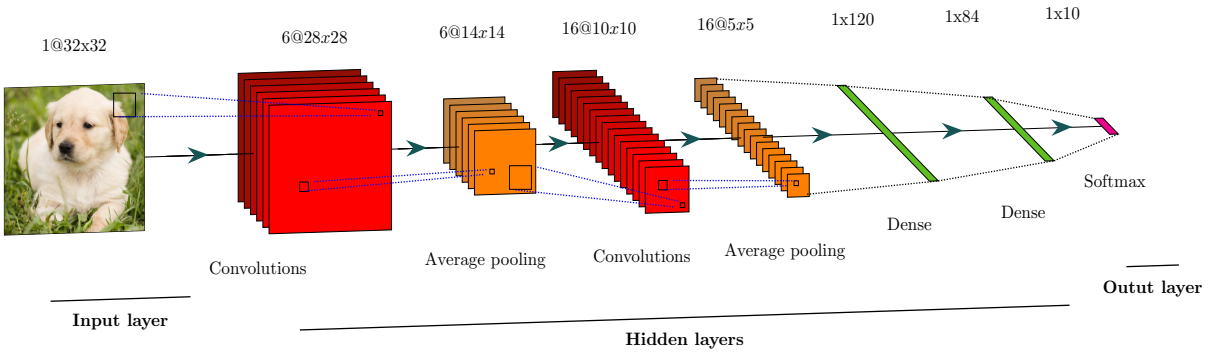


Figure 3.2: LeNet-5 convolutional neural network

3.2.2 The convolutional layer

Another key difference of the convolutional neural network are the layers within the hidden layer. While each layer has its inputs and outputs as feature maps, the way the values within the feature maps are found are different to traditional neural networks. One specific layer in convolutional networks is the convolutional layer. This is a layer in the hidden layer that transform feature maps into new feature maps using a kernel. A kernel is an $n \times n$ square with a corresponding weight for each of the smaller unit subsquares. Kernels are used in the convolutional layer within a convolutional network to learn the structural properties of the inputted image. It does this by placing the kernel in the corner of an image or an input feature map, and performing element-wise multiplication between the weights of each 1×1 sub-square of the kernel and the corresponding pixel value for the sub-square of the $n \times n$ subset of the image or input feature map, followed by summing the obtained results. This is equivalent to applying the dot product. We store the resulting dot product in the equivalent corner of a new output feature map. This output feature map should now contain one of its corners populated as the dot product calculated in the previous step. Next, before the output feature map has the rest of its values populated, we need to define the stride of the kernel. The stride of the kernel is the number of 1×1 squares that the kernel is shifted by before calculating the next dot products. Once the horizontal stride (and vertical stride) are specified, we should shift the kernel horizontally by the same number of units as the horizontal stride and calculate the new dot product of the kernel weights and the corresponding values of the inputted feature map that match the updated positions of the kernel weights. The output feature map should then be populated with this dot product by writing it in the appropriate position. This process should be repeated until the opposite corner is reached and then shift down by the vertical stride and repeat the process until the output feature map has all of its values populated. Figure 3.3 illustrates the process of populating the output feature map using a kernel.

The process of using the kernel is extremely beneficial for images. One benefit is by repeatedly using and shifting a kernel we extract a specific feature presented in different locations of an image or feature map and shrink this feature into a smaller output feature map. The specific feature that the kernel is looking for is determined by the weights of the kernel and different weights represent different features. For example, the kernel in Figure 3.3 has all of its ones in the center of the kernel and positions in the input feature map that lead to high dot products between this kernel and the input feature map indicate positions where that speicfic feature is present. This specific kernel for example represents a vertical line in the input feature map and in practice may provide hints towards an outline of an object in the original image. Another

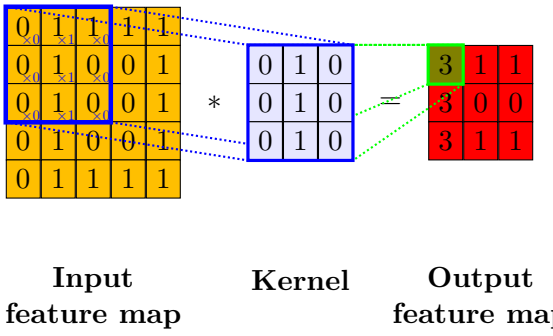


Figure 3.3: Example of kernel in convolutional layer using strides of size one (without padding) in convolutional neural network

benefit of the kernel related to its stride. By having the width and height of the kernel larger than the respective horizontal and vertical strides, many pixels in the image or filter map are repeatedly looked at, each time with a different collection of surrounding pixels. These slightly different shifts of the pixels, have the weights of the kernel in different positions leading to different strengths of patterns detected in the different positions. Having a small stride, also ensures the values are suitably combined, ensuring no important data is missed when creating the output feature map. One downside of the kernel, is that outer pixels fall inside the kernel few times, with corner pixels falling into the kernel only once [10]. Methods such as padding treat this by extending images to include an outer layer of zero-valued pixels, allowing for edges of images to appear in the kernel more times. Padding can be beneficial when the key part of the photo occurs at its edge, increasing the understanding of all parts of the photo.

With all these benefits of kernels for images it may come as no surprise that multiple kernels are used in the convolutional layer of a convolutional neural network. Since one kernel can detect one feature, multiple kernels can detect multiple features. Applying multiple kernels enables many different structures and patterns to be thoroughly checked in the image. Furthermore, in a specific convolutional layer, in each kernel, there are different weights for each of the different input feature maps. In other words, one kernel applies different weights for each of the different input feature maps. While each kernel has different weights for each of the different input feature maps, the kernel weights do not change when sliding the kernel across each respective feature map. In essecence, since in each kernel there are different weights for the inputs this is the exact same as the nodes of the standard neural network where at a given layer, each node within that layer assigns a weight (and bias) for all nodes in the previous layer. As stated previously weights represent both weights and bias and these weights of the kernels are updated during model training in a similar method to that of standard neural networks. Finally, following the convolutional layer after all output feature maps have been made, an activation function is applied to the individual values, allowing non-linear relationship between the data to be learned. Figure 3.5 zooms in on Figure 3.2 and shows how one kernel of multiple 5×5 kernels in a convolutional layer are applied in the convolutional layer. The first six 14×14 orange feature maps are converted into sixteen 10×10 red feature maps through these sixteen different kernels using a stride of one, without padding, in the convolutional layer.

3.2.3 Pooling layer

Another key property of the convolutional neural network is the pooling layer. The pooling layer is contained in the hidden layer of the convolutional network and its main aim is to summarise data in a compact form. The pooling layer makes use of input and output feature maps and the layer uses a kernel with a specified stride to convert the input feature maps to output feature

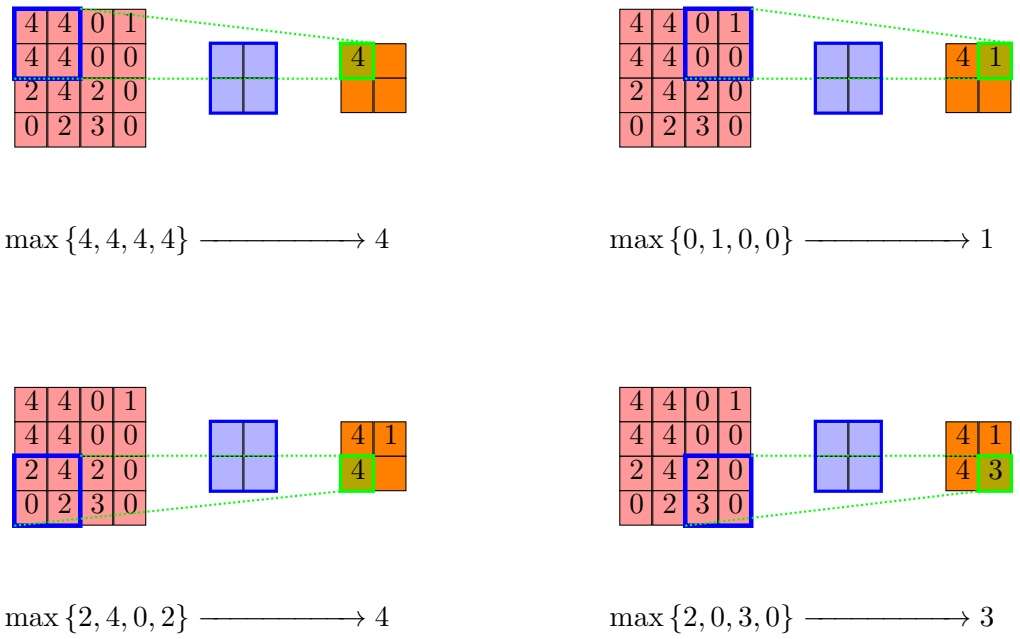


Figure 3.4: Max pooling with a stride of two

maps. The key difference in the pooling layer is that the kernel has no weights and instead the values inside the kernel are pooled together. Two popular methods [**gholama**] of pooling the numbers inside the kernel are max pooling and average pooling. Max pooling takes the maximum value within each kernel, writing the result in an output feature map. The other choice, average pooling, calculates the average of the feature map's values inside the kernel and puts it into an output feature map. Figure 3.4 demonstrates max pooling, as well as the process of sliding the kernel by its stride for populating the values of the output feature map.

Another feature of the pooling layer is the stride of the kernel. One of the most popular strides of the kernel in a pooling layer is a horizontal and vertical stride of two. This helps reduce the height and width of the output feature maps summarising the data in a reduced form. In particular, a stride length of two halves the height and width of the data essentially rewriting the important parts of the data using a quarter of the original inputs.

When using the kernel in the pooling layer across many feature maps, the pooling method of the kernel is kept the same and it produces just one output feature map for each of its input feature maps. For example, the kernel is scanned across one input feature map, producing one corresponding output feature map of the pooled values. The kernel repeats this for all the input feature maps, producing the same number of output feature maps, all with reduced width and height. No activation functions are used after pooling. The right half of Figure 3.5 shows a pooling layer being applied to the resulting feature maps of a prior convolutional layer.

3.2.4 Overview

Throughout the convolutional neural network feature maps are converted to feature maps analysing structural properties of the data and summarising the results. In a typical network, collections of convolutional and pooling layers are applied one after another. Together these collections of layers transform feature maps into smaller more concise feature maps gaining an improved understanding of the data. Many popular current models stack many convolutional

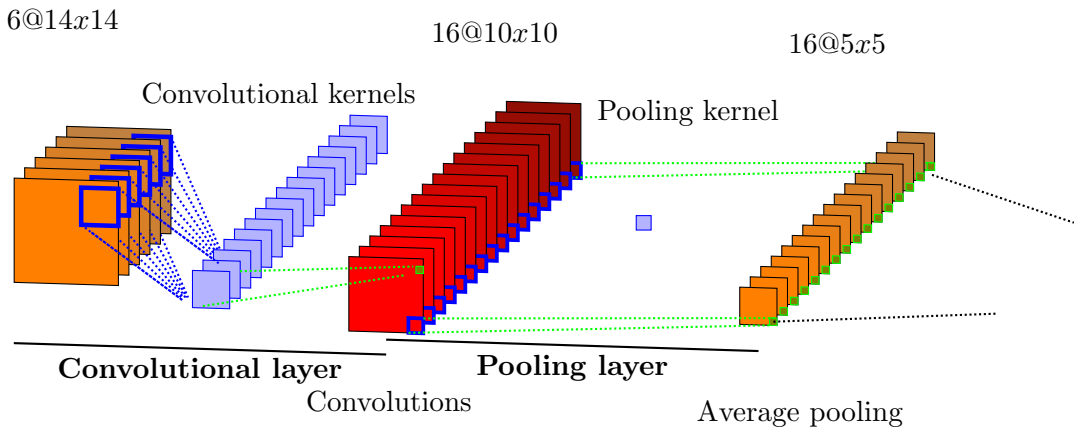


Figure 3.5: A convolutional layer followed by a pooling layer

layers before applying a pooling layer. This is due to the structural properties that the convolutional layers can explore, as well as, the activation functions applied within these layers. Pooling layers are also needed to summarise data.

The input into the convolutional neural network is the three-dimensional array of pixel values for the image. A Greyscale image is a two-dimensional array of pixels and an RGB image is three two-dimensional arrays of pixels. In the input layer of a convolutional network, a greyscale and RGB image can be thought of as one input feature map, and three input feature maps respectively. Although the original image is not a feature map, (since feature maps are only produced by the model), the image's arrays and their pixel values behave as them during the input stage of the model. Therefore the convolutional network can transform the pixels of an image into meaningful structures and summarise these structures throughout the process of the network.

Towards the end of the convolutional neural network, the collection of the feature maps needs to be converted into a one-dimensional array, preparing the model to predict the class of the image. One approach to achieve this uses a collection of convolutional kernels or a pooling kernel the same size as the most recently learned feature maps, creating many scalars that can be combined into an array. This is typically done when the size of the feature maps is sufficiently small enough so that the dot products or pooling method produced by the kernel encapsulates the majority of the feature maps' structural properties. The alternative approach includes flattening the three-dimensional representation of the most recently learned feature maps into a single one-dimensional array. Once we have a one-dimensional array we have reached the fully connected layers. The fully connected layers are a collection of dense layers prior to the final softmax layer that begin to convert the one-dimensional array towards the final predicted class probabilities for the image. The dense layers all have activation functions in them and the dense layers are fully connected layers with the same structure to that of Figure 2.1. In essence, in the fully connected layers, now that all meaningful interactions between adjacent pixels have been reduced to a single vector, the fully connected layers make use of fully connected nodes between layers, combining the features of the image. The final layer of the convolutional network, connected to the last fully connected layer, is the output layer where the softmax function is used and the probabilities for the different potential classes are output. Towards the end of the convolutional network in Figure 3.2, the sixteen 5×5 feature maps, after the second pooling layer, are flattened

to reduce the feature maps into a one-dimensional array consisting of four hundred elements. This is then followed by one fully connected layer, connecting all the discovered patterns in a flattened form, reducing the size of the array to eighty-four new elements. Finally, an output layer is connected to the previous fully connected layer, making use of the softmax function to produce probabilistic predictions for, in this case, ten different potential classes of the image.

To conclude, convolutional neural networks are simply neural networks that have been geared towards images. The standard properties of neural networks such as training a model on a dataset using optimization algorithms such as stochastic gradient descent to find the optimal kernel weights within the convolutional layers and dense weights in the fully connected, that minimise a specified loss fucntion, apply. The key difference in the convolutional neural network is the layers within it that allow surrounding pixels to be studied together, essentially allowing the model to "see" the images.

3.2.5 Popular models

Before we begin evaluating models for image classification, we need to choose a specific architecture for the model. Different architectures of convolutional networks have different combinations of convolutional and pooling layers as well as different activation functions throughout the network. These different architectures learn different features within an image. Typically, most successful architectures use an initial convolutional layer connected to the input layer to initially extract features from the image. Most of these architectures then follow using more layers often using many more convolutional layers than pooling layers.

- **LeNet-5:** As demonstrated in Figure 3.2, the LeNet-5 [14] is a convolutional neural network defined by the layers shown in the figure. The depth of the model is five, and it uses hyperbolic tangent activation functions (tanh) after the convolutional and fully connected layers. This model finialised in 1998 by Yann LeCun is a simple convolutional network that can achieve good results in basic image classification tasks.
- **VGG-16:** The VGG-16 [21], shown in Figure 3.12, designed in 2014 by K. Simonyan and A. Zisserman has many distinct properties. Properties specific to the VGG-16 include: many 3x3 kernels with stride one using padding in each of the thirteen convolutional layers enabling high levels of understanding between small groups of nearby pixels, five max pooling layers using 2×2 kernels of stride two, three fully connected layers, and a ReLu activation function after the convolutional and fully connected layers. The depth of 16 for the model refers to the thirteen convolutional and three fully connected layers within the network.
- **ResNet-50:** The ResNet-50 [6], created in 2015 at Microsoft Research, has its own distinct properties. ResNet-50 extends the idea of applying layers consecutively by allowing output feature maps of layers to be included in the input of another future layer, skipping layers. By including additional feature maps from specific layers before the previous layer, the network becomes stabilized reducing the risk of gradients of the loss function either vanishing and becoming extremely small, or exploding and becoming abnormally high. These additional connections are called skip connections and are a key property of the ResNet-50. The ResNet-50 is split into sixteen residual blocks making use of the skip connections in each block. With these additional skip connections, the ResNet-50 is another convolutional network that uses forty-eight convolutional layers, one max pooling, and one average pooling layer. Throughout the network the ReLu activation function is applied.

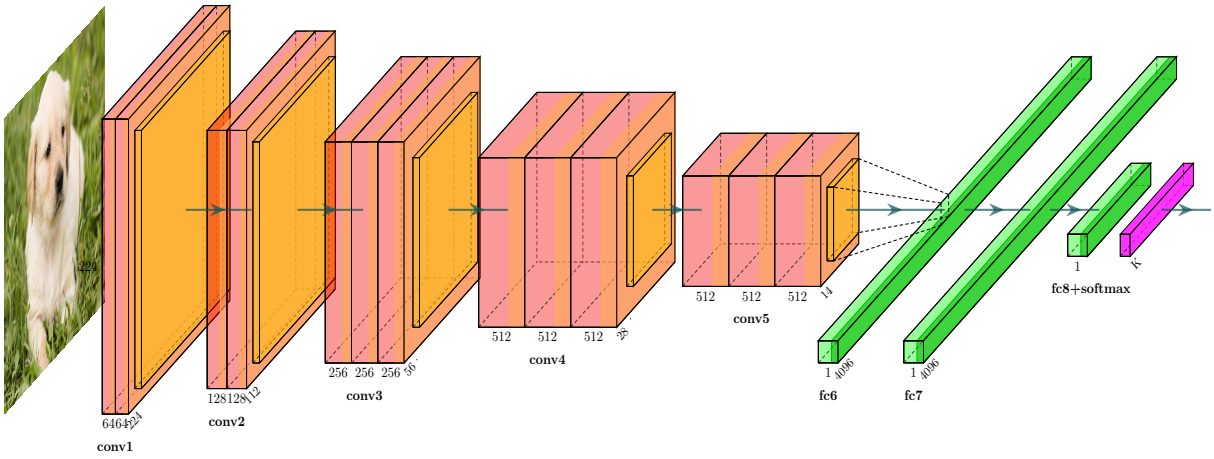


Figure 3.12: VGG-16 model

The ImageNet large-scale visual recognition challenge [19] was an annual competition that began in 2010, challenging competitors to make high-achieving models in a [3] huge dataset with over one million different images split across one thousand different classes. In this challenge, using the top five predictions for each image, VGG-16 achieved a top-five accuracy of 92.7% ranking second place in 2014 and the ResNet-50 achieved a top-five accuracy of 93% ranking first in 2015. In general, the ResNet-50 is considered slightly better than the VGG-16 due to its effective skip connections enabling easier optimization and its typically higher accuracy. Nevertheless, both models have successfully applied convolutional networks for image classification models.

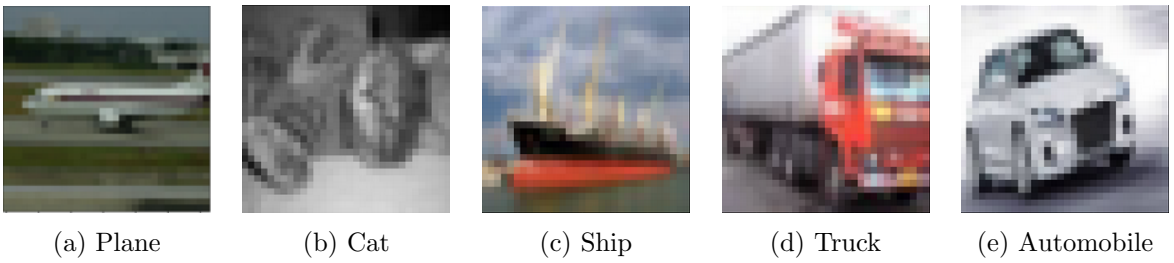
Finally, we note that both models were originally of size 224×224 and had outputs of 1000 classes. However, both models as well as LeNet-5 can be adapted to different input shapes and the final layer can be changed to predict different number of classes,

3.3 Image classification deep learning example

So far, we have gone from traditional programming, needing to predefine the entire program, to introductory machine learning, finding simple patterns in the data with basic techniques, to neural networks using deep structures that learn many non-linear relationships, to convolutional neural networks, specifically adapted towards allowing the model to see. In this section, we put the three popular models LeNet-5, VGG-16 and ResNet-50 in a practical example, including all the steps necessary to train an image classification model. After the model is trained we will test it to see how it performs.

3.3.1 The dataset

The CIFAR-10 [11] dataset is a popular initial dataset primarily used for image classification. This dataset consists of 60,000 different 32x32 images where each image is a picture of an object belonging to one of 10 different classes. The different classes of images in the data sets include automobile, dog, truck and seven other categories. Additionally, the dataset is separated into a train and test set containing 50,000 and 10,000 images respectively and both sets contain an equal number of pictures for each of the ten classes. The dataset is a good dataset for image classification. Figure 3.13 shows a random image from each of the ten classes of the CIFAR-10 dataset.



(a) Plane (b) Cat (c) Ship (d) Truck (e) Automobile

(f) Horse (g) Bird (h) Frog (i) Dog (j) Deer

Figure 3.13: Random sample of CIFAR-10 images

3.3.2 Model training

Now that we have a dataset, we need to train a model. We will train the three convolutional neural network models LeNet-5, VGG-16 and ResNet-50. The models will be trained in the popular machine-learning free-to-use website Kaggle. In Kaggle we will use Keras and TensorFlow [17] to complete and train our model. The following steps are how we train the model.

1. **Load the CIFAR-10 dataset:** We will load the dataset and then split it into 40,000 train images, 10,000 validation images and 10,000 test images. We then apply a preprocessing step to all sets of data. Also, convert output vectors into one hot encoding which is an equivalent form that helps models distinguish outputs.

2. **Define the model:** We define the model according to its architectures. VGG-16 and ResNet-50 will be adapted to take inputs of size 32×32 and the softmax layer will be changed to 10. Also compile the model using the cross entropy loss function 2.3 and an optimiser.

3. **Train the model:** Starting from random weights the model uses the optimiser to find the optimal weights that minimises the cross entropy loss for the training data. Train the optimizer for a minimum of 20 epochs with ideally 100 epochs.

4. **Accuracy** Once the model is trained, new images can be input into the model and the model will complete a forward pass using the learned weights to compute the probability that the image belongs to each class, Compute the accuracy 1.3 on the test set by using the models predicted output as the class with the highest probability.

The above method is the general process for training deep learning models. The goal of training a model is to have high accuracy on the test set. Note that the model is trained on the training set therefore we expect a high accuracy on the training set. We will briefly monitor the accuracy and loss on the training set to check for any potential overfitting of the data.

3.3.3 Results 1

For a fair comparison, we train three models, all using normalisation and stochastic gradient descent of batch size 128. The accuracy of the training and test data are shown in Tables 3.1 and 3.2.

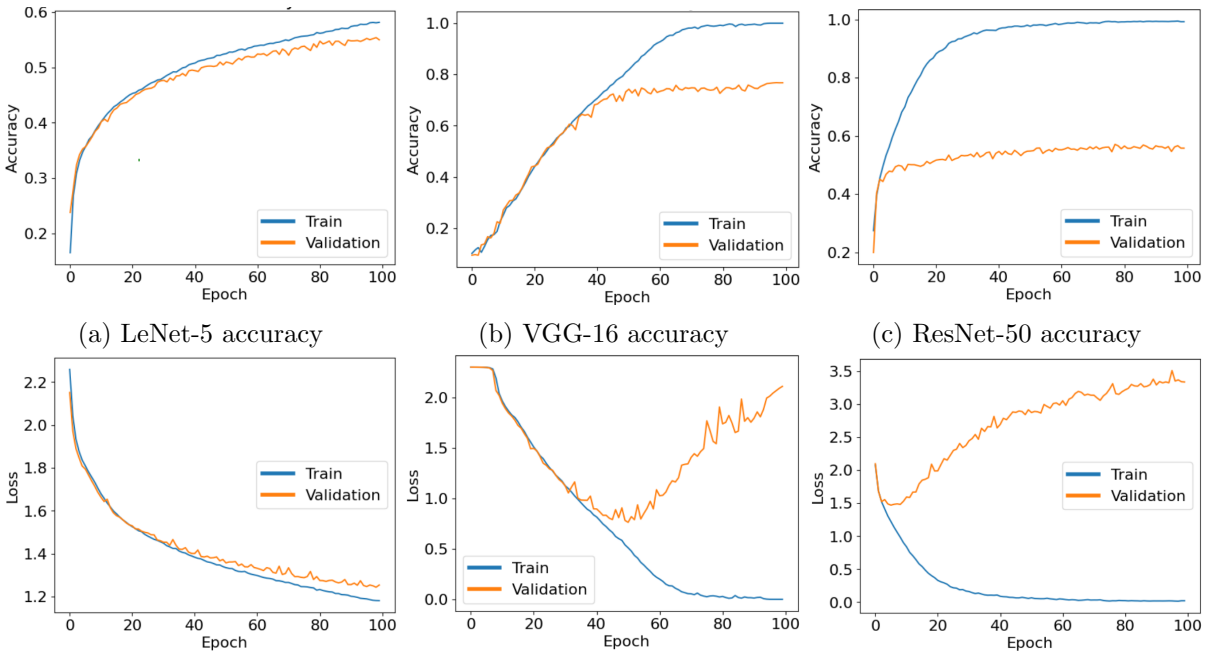
	Number of epochs		
	20	50	100
LeNet-5	44.61	52.34	58.09
VGG-16	41.53	80.77	100.0
ResNet-50	87.60	98.18	99.25

Table 3.1: Training data accuracies percentage

	Number of epochs		
	20	50	100
LeNet-5	44.11	50.47	55.43
VGG-16	43.99	73.23	76.19
ResNet-50	51.18	54.76	55.68

Table 3.2: Test data accuracies in percentage

It appears that VGG-16 is the best model in this case and ResNet-50 struggle severly. To gain an insight into why the model struggled to achieve high accuracy we need to further inspect the models. In Figure 3.14 the top three figures show the accuracy of the models for the training and validation data as the epoch increases and the bottom half shows the loss of the models as the epochs increase.



(a) LeNet-5 accuracy (b) VGG-16 accuracy (c) ResNet-50 accuracy

(d) LeNet-5 loss (e) VGG-16 loss (f) ResNet-50 loss

Figure 3.14: Three models accuracy vs epoch and loss vs accuracy plots

Observing Figure 3.14f shows that after few epoch, the loss function for the training data began to increase whereas the validation data began to increase. This is an example of overfitting. Since the model has many layers and the data was originally designed for the large dataset ImageNet, we see that the model has overfit the data. On the other hand VGG-16 performed well, although some overfitting began to occur at around 50 epoch.

3.3.4 Results 2

We again train the three models, but this time we use a different configuration. Lenet-5 will use Adam optimizer and normalisation, VGG will use subtracting mean per coloured channel and ResNet-50 will use Adam optimiser and standardisation by all values. Tables 3.3 and 3.4 show the train and test accuracies with these new configurations.

With minor changes we see some very different results. In particular ResNet-50 performs much better with the new optimiser and preprocessing. This shows that playing around with different

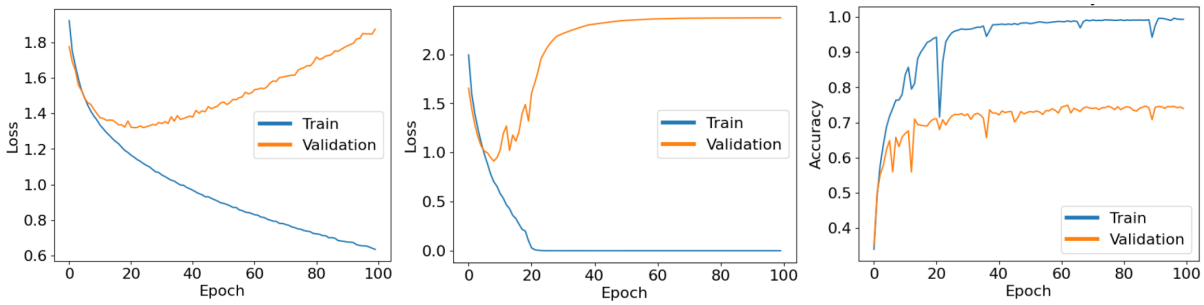
	Number of epoches		
	20	50	100
LeNet-5	58.60	69.14	78.57
VGG-16	95.04	100.0	100.0
ResNet-50	94.85	0.98.53	99.46

Table 3.3: Train accurcies (%)

	Number of epoches		
	20	50	100
LeNet-5	52.10	52.42	48.73
VGG-16	73.10	75.14	74.51
ResNet-50	70.92	72.83	72.81

Table 3.4: Test accuracies (%)

techniques can affect the results and in general a few techniques should be tried before achieving the best model. Figure 3.15 show the loss over the number of epochs ran.

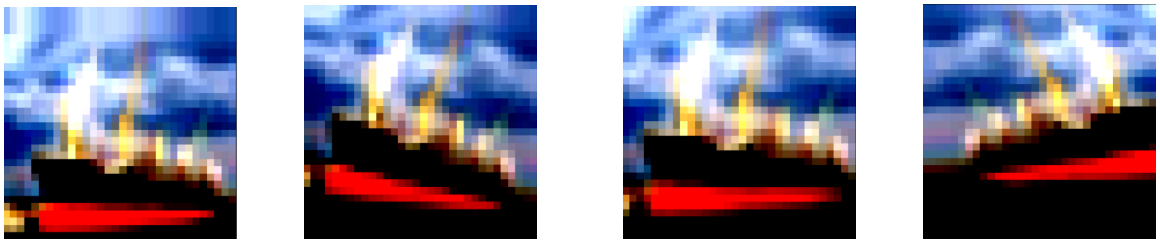


(a) LeNet-5 loss (b) VGG-16 loss (c) ResNet-50 loss

Figure 3.15: New optimiser and preprocessing: loss vs epoch plots

Again we see overfitting, particually in VGG-16 that within the first twenty epoch overfitting began to occur. One method that we could use to limit overfitting is a regularization technique on the cross entropy loss. Another method, particular to images, that we could use is data augmentation.

3.3.5 Data augmentation and improved results



(a) x and y shift (b) rotation and channel intensity shift (c) zoom in and shear angle (d) All and flip

Figure 3.16: Data augmentation examples

Data augmentation [20] is an extremely beneficial concept in training deep learning models for image classification. By using data augmentation, different variations of the same image are produced enabling the model to adapt to significantly more images. For each image there are many different applied transformations that can turn the image into something the model has not seen. In TensorFlow it makes use of efficient algorithms, creating the data augmented images during model training without needing to store many images using valuable space in memory. For the CIFAR-10 dataset we will use seven different augmentation techniques as shown in Figure 3.16. These are: x and y shift up to 12% of its width or height, rotation by up to 15 degrees, zoom in by up to 10%, randomly flip the image horizontally, shift each coloured channel by up to 10% and shear angle up to 10 degrees. Shearing the angle essentially changes the angle

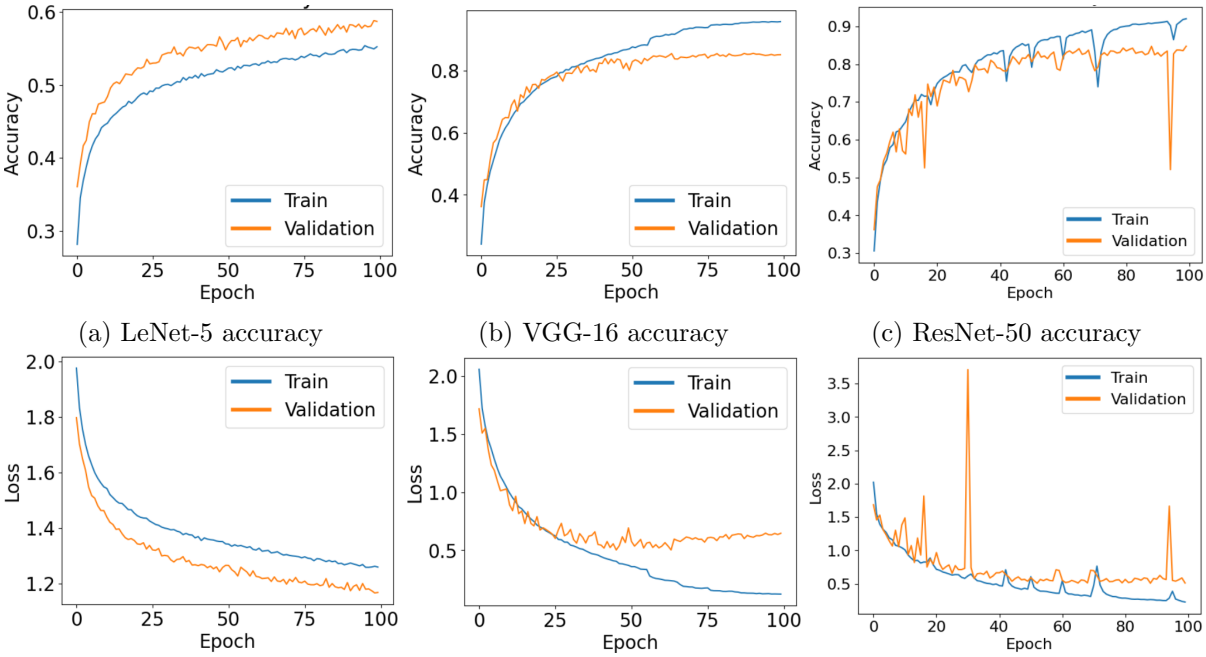
of the image, allowing the model to adjust to multiple angles. In Tables 3.5 and 3.6 we see that the results have changed again. In particular all models have performed better on the test set. Additionally, in Figure 3.17 we see that the loss vs epoch and accuracy vs epoch plots are more smooth indicating overfitting has reduced.

	Number of epoches		
	20	50	100
LeNet-5	48.23	52.33	55.46
VGG-16	75.33	87.19	95.91
ResNet-50	73.03	85.74	96.10

Table 3.5: Train augmentation accuracy (%)

	Number of epoches		
	20	50	100
LeNet-5	51.97	54.82	58.26
VGG-16	74.33	80.14	84.76
ResNet-50	70.71	80.06	84.71

Table 3.6: Test augmentation accuracy (%)



(a) LeNet-5 accuracy (b) VGG-16 accuracy (c) ResNet-50 accuracy

(d) LeNet-5 loss (e) VGG-16 loss (f) ResNet-50 loss

Figure 3.17: Data Augmentation: accuracy/loss vs epoch plots

3.3.6 Transfer learning

To further improve the accuracy of the more complex VGG-16 and ResNet models we can perform transfer learning. Transfer learning is a method where a different model is trained on a dataset that encompasses a wide range of potential images similar to the dataset that you wish to train, and the optimal weights were found. Then, rather than starting model training from randomised intial weights, training can start from the pretrained weights of the same model on a different dataset. To successfully perform transfer learning, the same preprocessing techniques for the pretrained model must be used for the new dataset. This ensures the models weights and learned filters are roughly good starting points for detecting key features within images. If we then wish to train the same model on a new dataset, where the data in the new dataset is similar to the data in the dataset that was used for the pretrained model then the optimal weights for the pretrained model can be used as the initial weights for the new model. Since the data is similar, it is then expected that training should converge closer to the global minimum for the new dataset and accuracy for our dataset should improve if the pretrained model was

trained well. We now test the VGG-16 and ResNet-50 models using the pretrained weights for the ImageNet [3] dataset. Now, retraining the models using the model's respective pretrained weights for ImageNet without data augmentation we obtain the results in Tables 3.7 and 3.8. LeNet-5 is not included because the model's architecture for ImageNet is too simple and using a LeNet-5 architecture would struggle.

	Number of epoches		
	20	50	100
VGG-16	0.98	1.00	1.00
ResNet-50	0.98	0.99	1.00

Table 3.7: Transfer learning training accuracies (%)

	Number of epoches		
	20	50	100
VGG-16	0.84	0.86	0.86
ResNet-50	0.74	0.75	0.79

Table 3.8: Transfer learning testing (%)

One important finding is that the accuracy increased to over 90 percent for both models within five epochs. Furthermore the accuracy after twenty epoch for the test set did not alter much. Transfer learning is a valuable tool in when there is a pretrained model available, however transfer learning may not always be possible.

Chapter 4

Uncertainty calibration on deep neural networks

4.1 Calibration

4.1.1 Calibration Introduction

Deep neural networks have made significant progress on standard machine learning methods. In particular, the accuracy of these deep learning models has increased drastically. However, one important property of the neural network is the reliability of the probabilities produced for its predictions, and in recent years, these probabilities have been slowly drifting further away from the true probability that the model is correct [1] [4], [24].

For example, in multiclass classification, given an input and potential classes, a trained model will produce an associated probability for each of these classes, and its prediction for the input is chosen by predicting the class with the highest probability. Rather than solely relying on the model's prediction to make a decision, the corresponding probability of the prediction should be examined to understand the model's associated confidence for that prediction. For example, in medical diagnosis, a prediction of no disease is far less informative than a prediction of no disease with a probability of 0.7. While both have the same prediction, the probability clarifies the model's thought about the chances of the predicted output occurring, potentially influencing the actions that we might take. However, in recent years these produced probabilities have become far less real representatives of the true probability that the model is correct. For example, if the model predicts that a specific patient has no disease with a probability of 0.7 for one hundred different patients, then we expect that 70 percent of these patients do not have the disease. However, if the true number of patients without the disease is 50 or 90, then the produced probability is incorrect and not calibrated with the true probability of occurring. Our goal in this chapter is to realign the produced probabilities with the true chances of the prediction occurring, as well as explore several factors contributing to this misguided judgment.

4.1.2 Confidence and calibration

In multi-class classification problems, after training a model finding suitable weights, an input goes through the neural network and the final probability for each of the k classes is calculated using a softmax function. Earlier, we defined the softmax function. The softmax function is applied to the logit vector $\underline{z}$, where $z^{(j)}$ represents the preactivation value for class j, is:

$$\sigma_{SM}(\underline{z}) = \left(\frac{e^{z^{(0)}}}{\sum_{j=1}^{k} e^{z^{(j)}}}, \frac{e^{z^{(1)}}}{\sum_{j=1}^{k} e^{z^{(j)}}}, \ldots, \frac{e^{z^{(k-1)}}}{\sum_{j=1}^{k} e^{z^{(j)}}} \right).$$

The model's output vector is the collection of probabilities $\underline{\hat{p}}$. Recall in 2.6 the equation is

$$\hat{f}(x_i; \underline{\sigma}, \underline{W}) = \left(p_i^{(0)}, p_i^{(1)}, \ldots, p_i^{(k-1)}\right),$$

where we now write $\underline{\hat{p}_i}$ as a collection of its individual probabilties for classes $0, 1, \ldots, k-1$ for sample i. Now the softmax activation function produces the final outputs, therefore $\underline{\hat{p}} = \sigma_{SM}(\underline{z})$ and in particular for sample i,

$$\underline{\hat{p}_i} = \sigma_{SM}\big(\underline{z_i}\big), \tag{4.1}$$

where $\underline{\hat{p}_i}$ are the models outputted probabilites for sample i and z_i is the preactivation value of the output layer for sample i. This is the produced probability vector for one input sample. Now we define $\hat{p}_i$ to be the maximum value of $\underline{\hat{p}_i}$, that is:

$$\hat{p}_i = \max\{\underline{\hat{p}_i}\}. \tag{4.2}$$

This an important property of the model prediction and we define $\hat{p}_i$ to be the confidence of the model on its predicted output. The confidence describes the model's probability that its predicted value is correct. For example, in the medical diagnosis example, predicting a patient has no disease with associated probability 0.7, then the confidence of the prediction is 0.7. This is an essential property of neural networks as it can give the model user not only a predicted outcome, but an associated probability of that outcome occurring. Higher confidences will provide more reassurance to the user that its prediction is correct, affecting a persons decision based on the prediction. Furthermore, we define the predicted class $\hat{y}$ as the class which corresponds to the maximum probability in the probability vector. That is:

$$\hat{y}_i = \arg\max\{\underline{\hat{p}_i}\} \tag{4.3}$$

This links the ideas so far. An input from one sample is inputted into the trained neural network and a probability vector is produced where the position of the element in the vector represents the model's probability for that class. The prediction is then the class with the highest probability, therefore, the model predicts that input x_i from sample i has output of $\hat{y}_i$ with associated confidence $\hat{p}_i$. Furthermore, we note that the true output for x_i is y_i and therefore the model is correct if and only if $\hat{y}_i = y_i$.

It is important to distinguish between $\hat{p}_i$ and p_i. Here, we have defined $\hat{p}_i$ as the maximum probability of $\underline{\hat{p}_i}$. On the other hand p_i is the probability in $\underline{\hat{p}_i}$ that corresponds to the true class of sample (x_i, y_i). Therefore if $\hat{y}_i = y_i$ then $\hat{p}_i = p_i$ and the model predicted the correct class with confidence $\hat{p}_i$. On the other hand if $\hat{y}_i \neq y_i$ then $\hat{p}_i$ does not represent the probability p_i and the model predicted the incorrect class with confidence $\hat{p}_i$.

Example 4.1.1: Consider a neural network trained at classifying whether a patient has the common cold, the flu or neither. The neural network takes the patient information as input and produces probabilties for each output. Figure 4.1 shows a sketch of the inputs, neural network and output probabilites for the classes. In this example, the model predicts that patient that patient $i = 1$ has the common cold, flu or neither with probabilites $\underline{\hat{p}_1} = (p_1^{(0)}, p_1^{(1)}, p_1^{(2)})$ and in particular $\underline{\hat{p}_1} = (0.25, 0.4, 0.3)$. The maximum value is $p_i^{(1)} = 0.4$ therefore $\hat{p}_i = 0.4$ and its prediction is class one, the flu. This is equivalent to saying the model's prediction is the patient has the flu with a confidence of 0.4. If this patient really did have the flu, then this prediction would correct. Alternatively if this patient did have the flu, then this prediction would be incorrect. The goal is to have an accurate confidence of the predicted output that represents the true probability of the predicted output occurring.

Calibration is the process of ensuring that a model confidence is the true probability that the predicted event occurs. The calibration of the model's confidences is essential for making optimal decisions in when given a predicted label.

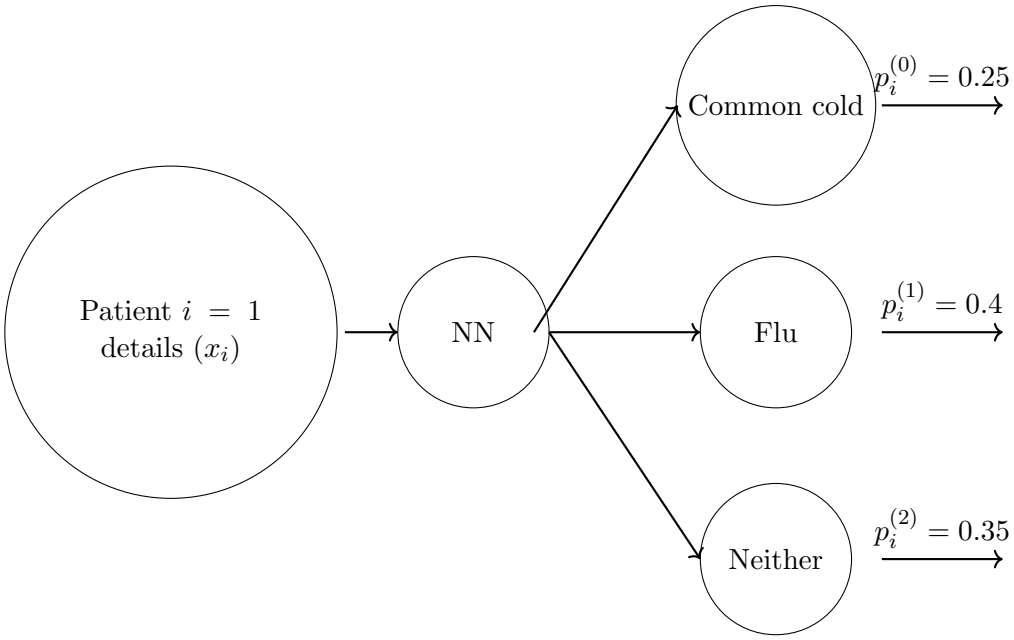


Figure 4.1: Example Neural network confidence

Example 4.1.2: Let us extend example 4.1.1 to 1000 different patients, each time predicting the probability of a patient's condition. Furthermore, for this new example, the model predicts that of the 1000 patients, 100 patients have the flu all with probability 0.8. Now with this information, we expect that 80 of these 100 patients ended up having the flu and if the model is perfectly calibrated in its confidences, then will be the case. However, if in fact 60 or even 100 of these patients had the flu, then the model is miscalibrated because the confidence is not an accurate representation of the true probability that the prediction is correct.

In the above examples we see that the calibration of a model in its confidences plays an important role. In practice the probability of these predictions give key insights into the model's assessment of inputs and when combined with optimal decisions in decision theory can improve the response that is made to certain predictions. Furthermore, calibrating these models is necessary, since with the growing rate of these artificial intelligence neural network models, we are needing to rely on them more often to extract meaningful information from insights that we have not yet discovered.

4.1.3 Metrics

There are several methods to measure the calibration of a model trained for classification. This calibration should be measured on the test set. Given test inputs $x_0^*, \ldots, x_{n-1}^*$, let $(\hat{y_0}, \ldots, \hat{y_{n-1}})$ be the model's predictions and let $\hat{p_0}, \ldots, \hat{p_{n-1}}$ be the model's confidences for these predictions. To measure the calibration of the model we begin by splitting the model's confidences into M equal width intervals, $(0, \frac{1}{M}]$, $(\frac{1}{M}, \frac{2}{M}]$, ... $(\frac{M-1}{M}, 1]$, and denoting each interval as B_m for $m = 1, \ldots M$ respectively. Then the average confidence in each interval is then defined as:

$$\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{\hat{p_i} \in Bm} \hat{p_i}, \tag{4.4}$$

and, the average accuracy in each interval is:

$$\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{\hat{p_i} \in Bm} \mathbb{1}(\hat{y_i} - y_i^*). \tag{4.5}$$

In 4.4, conf(B_m) is the average confidence of the confidences in the interval, and since the each confidence is the interval we have that conf(B_m) is a number in B_m, that is conf(B_m) $\in Bm$. In 4.5 for each confidence $\hat{p}_i$ in the specified interval, we take its corresponding prediction and compare it with the true prediction. Therefore, acc(B_m) is the accuracy of the predictions which have a confidence that fall into the specified interval B_m. These definitions give the following popular methods to measure a model's calibration [4]:

- Expected calibration error (ECE):

$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{n} \left|\text{acc}(B_m) - \text{conf}(B_m)\right|. \tag{4.6}$$

- Maximum calibration error (MCE):

$$\text{MCE} = \max_{m=1,\ldots,M} \left|\text{acc}(B_m) - \text{conf}(B_m)\right|. \tag{4.7}$$

- Confidence-accuracy plots.

The expected calibration error gives a single-value summary of the average miscalibration of the data. The maximum calibration error is useful in high-risk applications such as self-driving cars and gives the maximum calibration error from all of the intervals. The confidence vs accuracy plot is a plot measuring confidence on the x axis against accuracy on the y axis. Ideally, we wish for the plot to lie on the line $y = x$, i.e conf(B_m) = acc(B_m) for all $m = 1, \ldots, M$, for perfect calibration (see Figure 4.4). In confidence vs accuracy plots we plot the average confidence in that bin interval and the average accuracy for the confidence interval. These methods form a collection of methods to determine the calibration of a trained model. Perfect calibration is when the confidence of each interval and the accuracy of each interval are equivalent. In practice, almost all models are not perfectly calibrated and special techniques need to be used to fix them.

4.1.4 Evaluation of models on CIFAR-10 dataset

In section 3.4, we saw that VGG-16 obtained the best accuracies on the test set, ResNet-50 obtained the second best accuracies, and LeNet-5 obtained the lowest accuracies. Note that LeNet-5, VGG-16 and ResNet-50 have a depth of 5, 16, and 50 respectively. In general ResNet-50 is slightly considered better than VGG-16, therefore we may have expected this to perform best, however, for different datasets, different results arise. We now consider, how well are these models calibrated. Using the models with: the same configuration using stochastic gradient descent and normalisation, the data augmentented models and the transfer learning models, we obtain the expected and maximum calibration errors in Table 4.1.

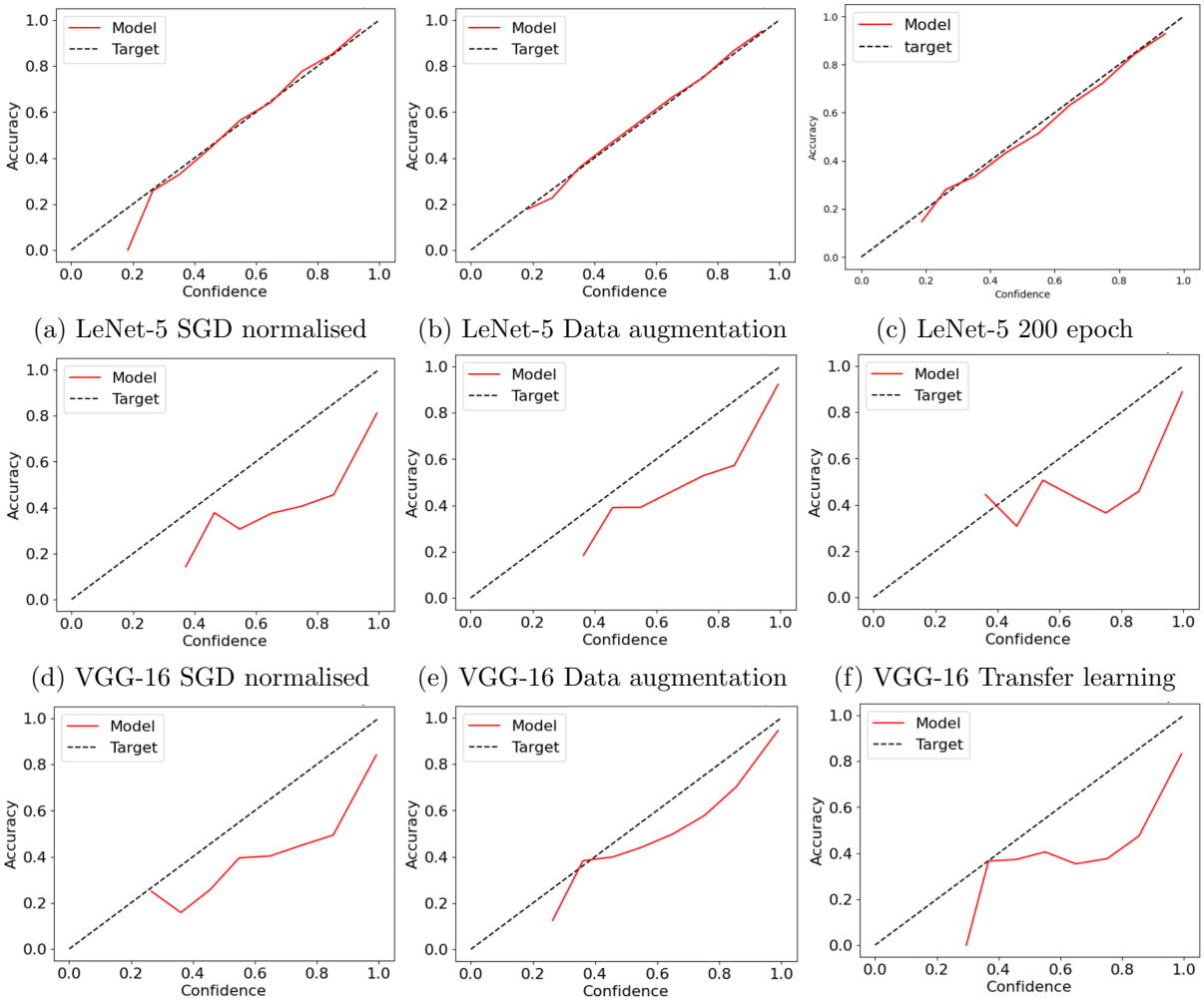
These are the expected calibration errors and maximum calibration errors and are single value estimates of the average and maximum miscalibration respectively. Note that the calibration error cannot exceed one since the confidence and accuracy cannot exceed one. Therefore, we display the calibration errors as percentages. The results show that there VGG-16 and ResNet-50 have large calibration errors especially without data augmentation. On the other hand the LeNet-5 does impressive and maintains a low expected calibration errors. To further investigate these results, we plot the confidence vs accuracy plots in Figure 4.2.

From Figure 4.2 we can see that the LeNet-5 model is well calibrated. This is because the model's confidence is approximately equal to the model's accuracy for all the bin intervals. Therefore, if we have a prediction in the LeNet-5 model and the model's confidence for this prediction

	Number of epoches		
	SGD and normalised	Data augmentation	Transfer learning
LeNet-5	1.44%	1.15%	N/A
	18.37	3.69%	N/A
VGG-16	19.89%	9.36%	18.50%
	39.72%	27.90%	38.92%
ResNet-50	19.20%	6.64%	17.92%
	32.64%	17.39%	37.96%

Key
ECE
MCE

Table 4.1: Expected and maximum calibration errors on CIFAR-10



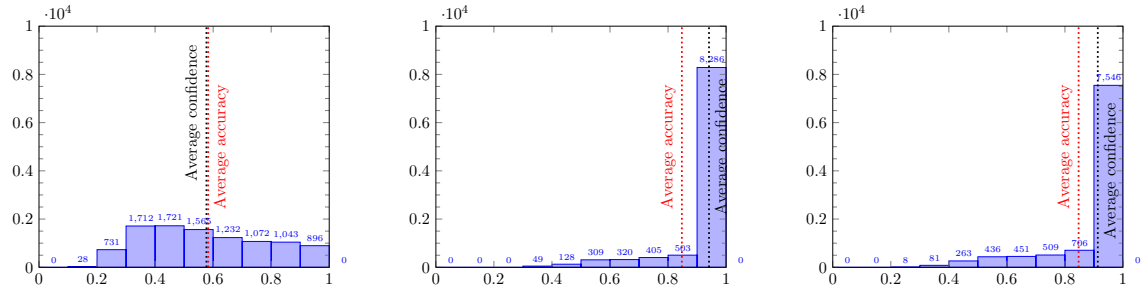
(a) LeNet-5 SGD normalised (b) LeNet-5 Data augmentation (c) LeNet-5 200 epoch

(d) VGG-16 SGD normalised (e) VGG-16 Data augmentation (f) VGG-16 Transfer learning

(g) ResNet-50 SGD normalised (h) ResNet-50 Data augmentation (i) ResNet-50 Transfer learning

Figure 4.2: CIFAR-10 confidence vs accuracy plots using 10 equal width bins

is 0.96, then approximately 96 times out of 100 this prediction will be correct. The VGG-16 model had a high accuracies, however, we can see that the model is not well calibrated, as it does not have its confidence vs accuracy plot on the line confidence = accuracy. We see that the model is typically overconfident as the red line for the model is often to the right of the line $y = x$. This is overconfident because the confidence is higher than the accuracy. For example in the VGG-16, transfer learning plot (Figure 4.2f) we see that when the model confidence is 0.8, the actual accuracy is 0.4. Therefore, if the model makes a prediction with confidence 0.8 that the prediction is correct, in reality the model is only correct approximately 40 times out of



(a) LeNet-5 (b) VGG-16 (c) ResNet-50

Figure 4.3: Proportion of predictions' confidences for data augmentation models

100. Therefore the model is not calibrated. Similarly, the ResNet-50 models are overconfident and the calibration needs improving. We also see from Figure 4.3 that the VGG model and ResNet models are very confident for most of its predictions whereas the LeNet-5 model is not overconfidence.

Our goal throughout the next few sections is to explore methods that will improve the calibration of these models. We will focus on the theory and the test the results on the data augmentation models because in practice, these are the most common models that will be used.

4.2 Temperature scaling

4.2.1 Theory

One method that almost always improves the model's calibration is temperature scaling. Temperature scaling [4]is a post-processing method that updates the confidences of the model predictions by scaling the logits by a temperature parameter before softmax is applied. It finds the optimal temperature by minimising the cross entropy loss on a held out validation set scaled by the parameter. Once this temperature has been found, the confidences of the predictions are updated using the optimal temperature. Recall from 2.3 we have the cross entropy loss as:

$$l\left(\{(x_i', y_i')\}_{i=0}^{n_1-1}; p_0', \ldots, p'_{1n_0-1}\right) = -\sum_{i=0}^{n_1-1} \log\left(p_i'\right), \tag{4.8}$$

where we now rewrite the cross entropy loss using the validation set $(x_i', y_i')\}_{i=0}^{n_0-1}$ and $p_0', \ldots, p'_{n_0-1}$ are the corresponding probabilities produced by the model for the true class of validation sample i. Now let $\underline{z_i'}$ be the logit vector for validation sample i and let z_i' be the corresponding element in the logit vector before softmax is applied such that:

$$p_i' = \sigma_{SM}\left(\underline{z_i'}\right)^{(t_i)}, \tag{4.9}$$

where t_i is the true class of sample i. Then the optimal temperature is found by minimising the cross entropy loss on the validation set where the logits are scaled by the parameter the temperature. Therefore, substituting 4.9 into 4.8 and finding the value of T that minimises the cross entropy loss on the validation data is found according to the following formula:

$$T_{\text{optimal}} = \arg\min_T \left(-\sum_{i=0}^{n_1-1} \log\left(\sigma_{SM}(\frac{\underline{z_i'}}{T})^{(t_i)}\right)\right). \tag{4.10}$$

As seen in 4.10, calculating the optimal temperature requires calculating the cross entropy loss similar to the way it calculated during model training, however for temperature scaling: we

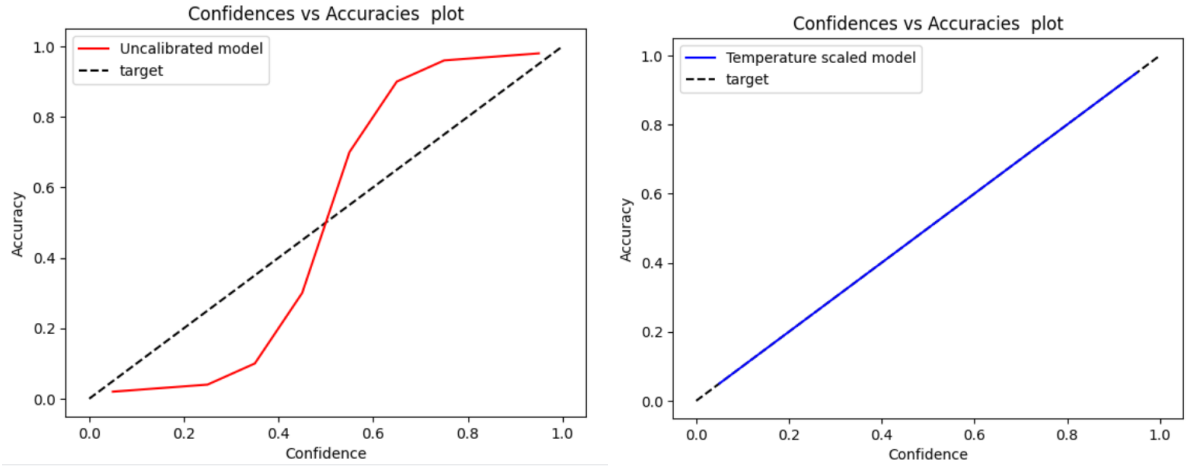
scale the logit by a temperature before applying the softmax formula, we find the optimal value of T that minimises the cross entropy loss, and we perform these calculations on the held out validation set instead of the training set. Finally, once the optimal temperature is calculated, instead of using 4.1 and 4.2 any calculated model confidence for any prediction is updated to a new confidence by:

$$\hat{p_i} = \max\{\sigma_{SM}(\frac{z_i}{T})\}. \tag{4.11}$$

The idea behind temperature scaling is that during model training since the aim of the model was to minimise the loss function on the training data, the model begins to overfit the training data and become overconfident in its predictions. Therefore, temperature scaling uses a separate unbiased validation set to reanalyse the total loss over the validation set and fine-tune the confidences according to the minimum validation loss.

Temperature scaling does not change the accuracy of the model; it only updates the confidences, typically improving calibration

Notice that in Figure 4.4a, the non calibrated model, the model is over confident for low accuracy values and under confident for high accuracy values. This is an undesirable property of models that we wish to recalibrate by bringing the confidences back to the target calibration line as in Figure 4.4b. The line in Figure 4.4b represents $\text{Conf}(B_m) = \text{Acc}(B_m)$ for all bins $m = 1, \ldots, M$.



(a) Non calibrated S shaped model

(b) Perfectly calibrated model

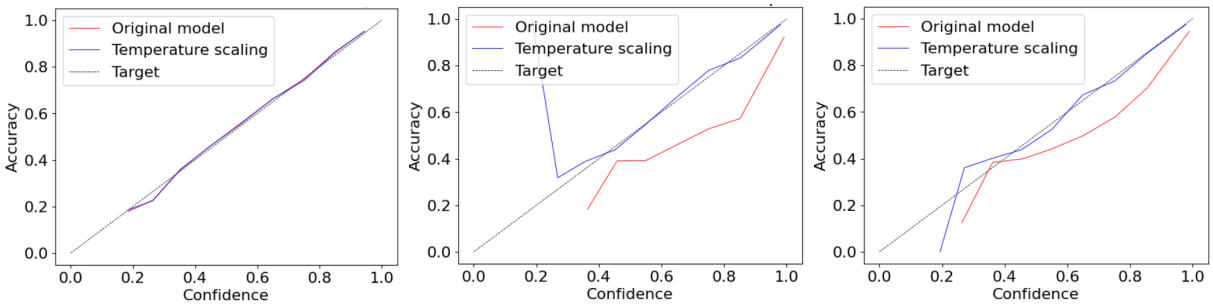
Figure 4.4: Temperature scaling visualization

4.2.2 Results

Using the same data augmentation image classification deep learning models for the CIFAR-10 data set and training on the train dataset (40,000 images), applying temperature scaling on the validation set (10,000 images), we obtained the calibration confidence vs accuracy plots on the test set (10,000 images) for the 100 epoch models in Figure 4.5. The table also displays the temperature scaled expected calibration errors for each model.

Model	TS ECE
LeNet-5	1.13%
VGG-16	1.07%
ResNet-50	0.98%

The confidence vs accuracy plot for these models are below. The red lines are equivalent to



(a) Lenet-5 (b) VGG-16 (c) ResNet-50

Figure 4.5: Confidence vs accuracy plot for three models for CIFAR-10 dataset

Figures 4.2b, 4.2e, and 4.2h for each model and the blue lines are the post processed temperature scaled improved calibration models. The accuracies for the models in Figure 4.5 on the test set were 58.26%, 84.76%, and 84.71%, respectively. Therefore, the VGG-16 model best fits this data. The ECEs (using 10 equal-width bins) for the models on the test set were 1.15%, 9.36%, and 6.64%, respectively, which means that the LeNet-5 model is best calibrated for this dataset. From the confidence-accuracy plots and the ECE scorings, we see that the LeNet-5 model is best calibrated before temperature scaling. We see that before temperature scaling, both the VGG-16 and ResNet-50 models have their confidences greater than their respective accuracies. Hence, both models are overconfident in their predictions for each data sample. The calibrated models brings the models' confidence back to a less overconfident state and hence reduces the miscalibration error in each model. Furthermore, these results hint at a general theme that improvements in deep learning model performance has lead to larger calibration errors. This is an example of temperature scaling, a post-processing technique that restores a model's calibration.

4.2.3 Cross-validation calibration

Now we understand that it is important for a model to have high accuracy and to be well calibrated. And we saw that we split 60,000 images into 40,000 training images, 10,000 validation images and 10,000 testing images. Since more data in the training set leads to higher accuracy in the model we ask: can we have a calibrated model without the need for a validation set? If we could do this, then more data (50,000 images instead of 40,000) could be used to train the model hence improve the accuracy. Maybe we could use the test set as a validation set? No! This cannot be done as the test set must always remain as a test set to ensure the final model performs well on new and unseen data. If we were to use the test set as a validation set, then the model would be trained on the test set, and hence the model would be expected to do well when tested on the test set. However, we can have a calibrated model without the need to waste data on a holdout validation set by using cross-validation.

There are 60,000 images in the dataset and we wish to have 50,000 train, 10,000 test - but then we cannot calibrate the test confidences. A solution is to create a base $Model_0$ has 50,000 train images (and 10,000 test images). Then split the 50,000 train into subsets $A_1, \ldots, A_K$, and let

- $Model_1$: (50,000 - A_1) train, (A_1) validation and obtain $T_{1optimal}$.
- $Model_2$: (50,000 - A_2) train, (A_2) validation and obtain $T_{2optimal}$.
- ...
- $Model_k$: (50,000 - A_K) train, (A_K) validation and obtain $T_{Koptimal}$.

Here we are obtaining the optimal T by applying temperature scaling to each sub model. Once all $T_{1optimal} \ldots, T_{1optimal}$ have been calculated we compute the optimal temperature for the base model using:

$$T_{optimal} = \frac{1}{k} \sum_{k=1}^{K} T_{koptimal} \tag{4.12}$$

Finally after this, we apply temperature scaling to $Model_0$ using $T_{optimal}$ to obtain a model trained on more training samples that similary has had temperature scaling applied.

This method is very strategic. It allows for the model to be trained on more data and hence the model should have a higher accuracy. Also the cross validation allows to obtain a value of $T_{optimal}$ that should improve post calibration of the model without the need for an explicit validation set. Some potential downsides are that finding the optimal T for the base model can be computationally expensive and if there are drastically different values of $T_{ioptimal}$ for $i = 1, , k$, then the model produced optimal temperature for the base model may not be reliable.

4.2.4 Results

We now use k-fold cross-validation with k=5. We use the VGG-16 trained on 20 epoch using transfer learning. We use a lower epoch due to computational power. We also expected calibration error with 10 equal width interval bins and we keep the test set separate until the final model is made.

Model	Description	Acc test data	Prior ECE	$T_{koptimal}$	TS ECE
M_1	5th 10,000 validation	0.84759998	0.109027788	2.24861	0.016914
M_2	4th 10,000 validation	0.84200	0.10417710	2.05932	0.0184781
M_3	3rd 10,000 validation	0.850000	0.106728874	2.21509	0.024898
M_4	2nd 10,000 validation	0.8201000	0.1265961	2.27681	0.02508
M_5	1st 10,000 validation	0.8403000	0.1133089	2.23384	0.026402
M_0	50,000 train, 0 validation	0.85190	0.108044	2.2067	0.02691

From the above data, we see that indeed the base model trained with all the training data has the highest accuracy on the test data. Furthermore, since the CV models had similiar T optimals, using the average T optimal ($T_{0optimal} = \frac{1}{5}\sum_{k=1}^{5} T_{koptimal}$) for the base model improved the calibration on the test set. This can be seen as the ECE before temperature scaling on the base model was 0.1080, and temperature scaling reduced this to 0.0269. Also see Figure 4.6 which shows the confidence accuracy plot for the original and temperature scaled base model M_0.

4.3 Isotonic Regression

4.3.1 Theory

Another post-processing method that typically improves the calibration of the model is isotonic regression. Isotonic regression [25] is a technique used in binary classification that fits a non-decreasing (or non-increasing) piecewise function to the probabilities for one class and the true outputs. The new function that is fit is called the score function and is used to transform a confidence into an updated confidence. The problem that isotonic regression solves is:

$$\min_{\substack{\theta_1,\ldots,\theta_M \\ a_1,\ldots,a_{M+1}}} \sum_{i=1}^{M} \sum_{i=1}^{n} (\mathbb{1}(a_m \leq \hat{p_i} < a_{m+1})(\theta_m - y_i)^2. \tag{4.13}$$

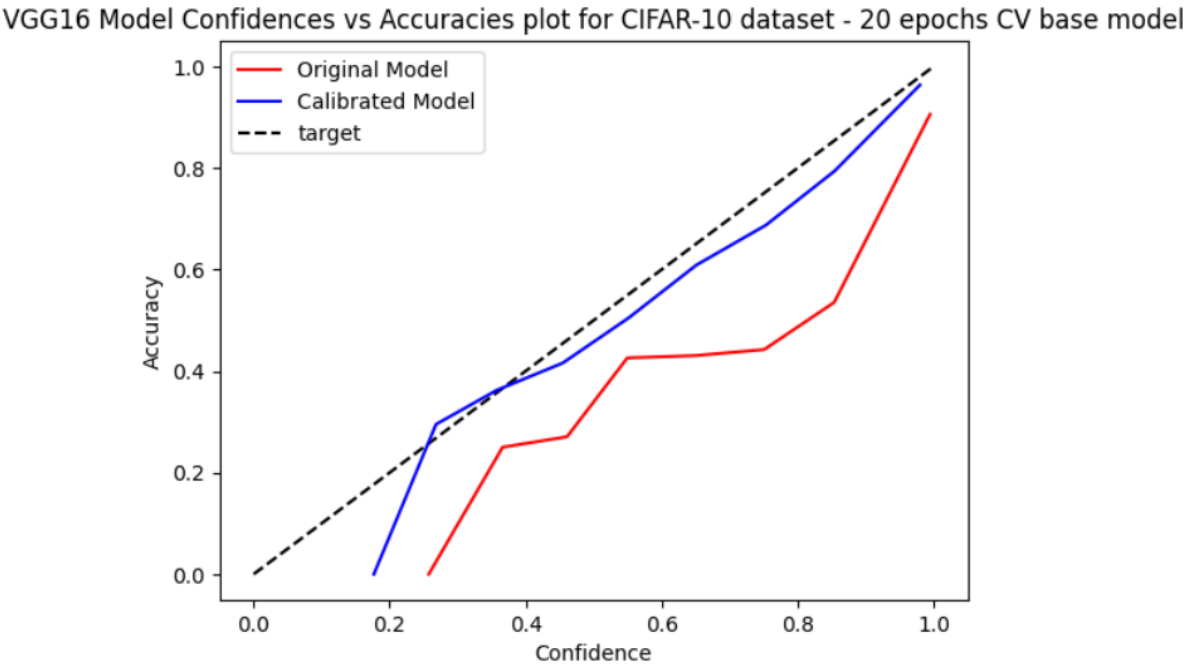


Figure 4.6: VGG16 20epoch CV base model with CV TS plot

where there are M intervals for the isotonic function, $a_1, \ldots, a_m$ are the boundaries of these intervals, and $\theta_1, \ldots, \theta_M$ are the values of the piecewise isotonic function that optimally fits the data. When combined together, these values, make a function $\hat{f}$ and the new calibrated probability can be calculated as computing $\hat{f}(\hat{p})$.

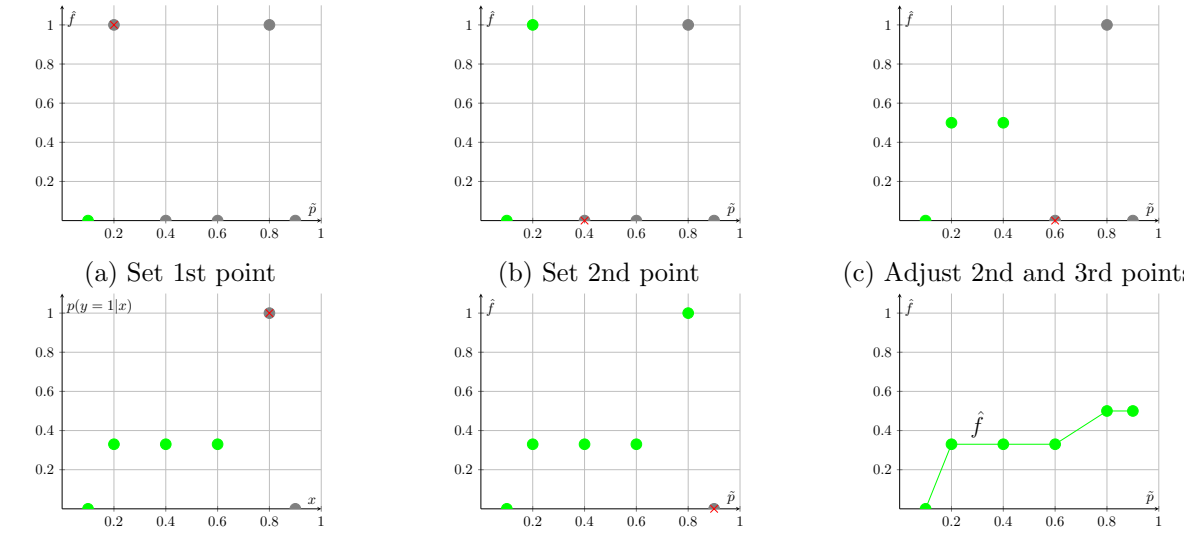
In its natural form, isotonic regression improves the calibration of a binary classification model. One benefit is that it does not rely on any specific distribution, and can be any monotonic function. Then, for possible binary outputs (zero or one) it assumes a monotic relationship between the probabilities for class one and the true outputs. Then isotonic regression calculates the updated probabilities for a validation set for the positive class 1 and fits the isotonic function between these probabilities and the true outputs to calculate the score function.

Example 4.2.1: Assume we have a model where for each input the model makes a predicted value of either 0 or 1. Let $y'_0, \ldots, y'_{n_1-1}$ be the true outputs for validation samples $i = 0, \ldots, n_1-1$ and let $\tilde{p'_0}, \ldots, \tilde{p'_{n_1-1}}$ be the model probabilities that validation sample i belong to class 1. To clarify this definition $\tilde{p'_i} = P(y'_i = 1|x'_i)$ and intuitively larger values of $\tilde{p'_i}$ means more likely that sample i belongs to class 1. This is the assumption that isotonic regression makes assuming the probabilities for the positive class and the true outputs have a non-decreasing relationship. Now we have used a neural network to make probabilities for the positive class and we also have the true outputs defined in Table 4.2. Then, the isotonic regression algorithm for finding

	i					
	1	2	3	4	5	6
$\tilde{p_i}$	0.1	0.2	0.4	0.6	0.8	0.9
y'_i	0	1	0	0	1	0

Table 4.2: Binary classification training data

the optimal function is shown in Figure 4.7. In this algorithm, we plot the points in a graph, with probabilities of class one on the x-axis and true output either 0 or 1 on the y-axis. Then



(a) Set 1st point

(b) Set 2nd point

(c) Adjust 2nd and 3rd points

(d) Adjust 2nd, 3rd, 4th points

(e) Set 5th point

(f) Adjust 5th, 6th points for final function

Figure 4.7: Isotonic regression example

for each point, we use the pool adjacent violators algorithm [**zadrozny2002trais**] used to find the function that minimises the square error. In the figure, moving in the positive x direction, we begin by seeing the first and second points for an increasing function. However, the third point falls below the second point. To ensure the function stays monotonic, the second and third points meet halfway and the fourth point is considered. Similarly, this point is below, therefore the whole function is re-lowered. In isotonic regression, this process repeats ensuring the function is always a monotonic function. With this process, isotonic regression is a post-processing technique with the specific goal of calibrating the probabilities of the positive class, which can in turn be used to improve the confidence of the model predictions.

The post-processing method can also be extended to multi-class classification [4]. Assume we want to use isotonic regression to calibrate a model's prediction for multi-class classification containing k classes. Furthermore, assume we have a trained model, validation set $\{(x'_i, y'_i)\}_{i=0}^{n_1-1}$, and test set. Begin by choosing a class k, and label all validation samples as 0 if it does not belong to class k and 1 if it does belong to class k. Then calculate all model probabilities for the validation set by inputting all validation samples into the model to receive the output probability vectors $\underline{p_i}$ for each validation sample. Choose $\tilde{p_i} = \underline{p_i}^{(k)}$ for each validation sample. Then write the collections of data samples as $(\tilde{p_i}, y_i)$, where y_i is one if and only if i belongs to class k. Then use 4.13 to find the isotonic function between the probabilities of class k and whether or not the sample belongs to class k. Now update the probabilities for the test set using $f(\underline{p_i}^{(k)})$. Repeat the above steps for each class. After repeating each class, a non-normalised probability vector will be available. Normalise this probability to obtain the updated isotonic probabilities for each class.

4.3.2 Results

model	original accuracy	isotonic accuracy	original ECE	isotonic ECE
LeNet-5	58.26%	58.78%	1.15%	0.95%
VGG-16	84.76%	85.23%	9.36%	1.67%
ResNet-50	84.71%	84.57%	6.64%	1.08%

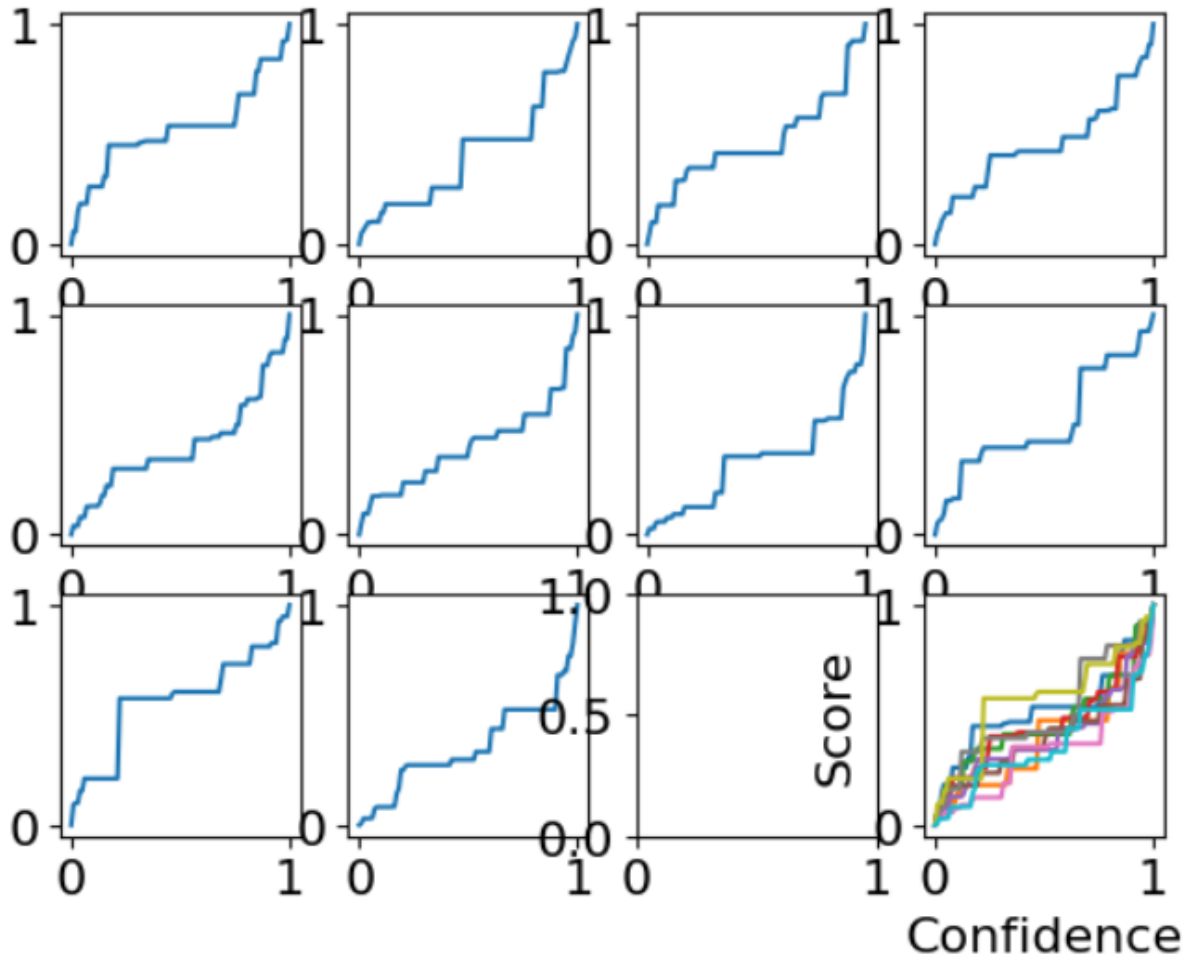


Figure 4.8: ResNet-50 data augmentation isotonic regression k vs rest conf-score plots

In the above table, we use the data augmentation models that we have previously saw. Notice that the accuracy improved using isotonic accuracy. Recall that there are 10,000 images in the test sets, therefore the original accuracy means that the model initially correctly predicted 8476 predictions for VGG-16, and after isotonic regression calibration it predicted 8523 predictions correctly. Exploring the differences we find that isotonic correctly changed 126 predictions and incorrectly changed 79 predictions. For example in one picture, the initial model gave a confidence of 0.35 to class 4 (starting from 0) and confidence of 0.63 for class 8. After Isotonic regression it gave a confidence of 0.32 for class 4 and confidence of 0.27 for class 8. In Figure 4.9, the actual class was class 4 (deer) so in this case, isotonic regression fixed this image. In the above table, we also see that the accuracy for two of the models improved. Therefore, isotonic regression will improve the calibration, however, it may positively or negatively change the accuracy.

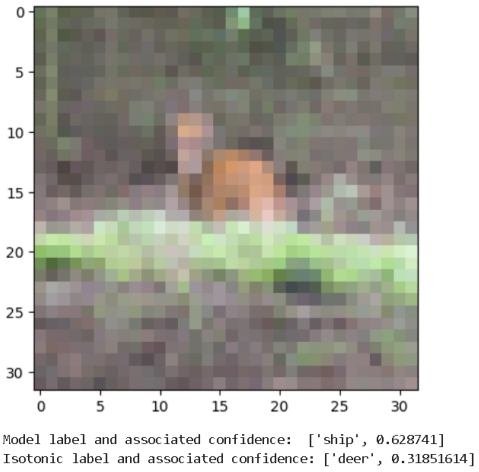
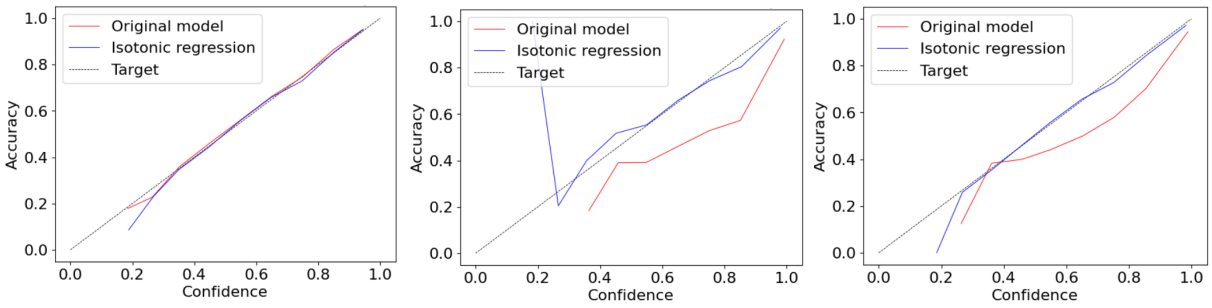


Figure 4.9: Example: isotonic regression deer vs truck

The confidence accuracy plot for this ResNet-50 data augmented epoch model as well as the LeNet-5 and VGG-16 model are shown in Figure 4.10, and it can be seen that isotonic regression improves the calibration.



(a) LeNet-5 (b) VGG-16 (c) ResNet-50

Figure 4.10: Isotonic 100epoch Data augmentation conf-acc plots

We see that isotonic regression improves the models' calibration. Also, since isotonic regression uses a one vs rest approach, it can improve calibration when the model has overconfidence for some classes and underconfidence for other classes. For example, in Figure 4.8 if the isotonic class k vs rest confidence-score plots vary drastically between different classes then isotonic regression will be able to account for these differences appropriately unlike temperature scaling that treats all classes together. For this reason, isotonic regression also has its key strengths when there are a high number of classes.

4.4 Venn-Abers

4.4.1 Theory

Another post-processing method that improves the calibration of a trained model is a more modern technique called Venn Abers predictors [23]. In binary classification, Venn Abers is a technique that improves the confidence of a model's prediction by fitting two different isotonic regressions, including a test sample's probability for the positive class that we wish to recalibrate. The first isotonic regression includes the specific probability for the positive class assuming the true class is 0, and the second isotonic regression includes this specific probability as well as an assumed class of 1. By performing two isotonic regressions, each with a different assumed class of the additional sample, we obtain two different isotonic functions for the specific confidence. These two different functions provide an upper and lower bound for the additionally confidence and under the right circumstances has mathematical guarantees of perfect calibration [22].

In binary classification, one key property of Venn Abers is that it provides an upper and lower bound for the true probability of the positive class. This interval can be used to create an interval for the confidence of the models's predicted class. This addresses a slightly different property of calibration as we can use the interval to gain insights into how sure the model is about its confidence. The actual probabilities used in Venn Abers is the probability of class one and an upper and lower bound around this interval are calculated. By including this probability twice with two different classes, the pool adjacent vialators will determine different probabilities at the specific additional probability. Venn Abers using isotonic regressions containing many data samples, tend to have its probabilities close together, meaning that more data samples in the isotonic regression leads to more accurate probabilites by the model.

There are several different methods for Venn Abers. One method includes combining the test sample whose confidence is desired to be calibrated, in the training data and train the model using the training data and the test sample twice. The first time train it using an assigned test sample class of 0 and the second time train it using an assigned class of 1. Once each model are trained, isotonic regressions are performed on the training data and test sample using

the confidences produced by the models that were just trained. This method leads to perfect calibration by using conformal mapping [.] The problem with this method however, is that it is extremely inefficient. To obtain accurate confidences for a test set, a model would need to be trained twice for each element of the test set and often model training can take many hours of time. Furthermore, the model would need to be specifically prepared so that different runs of the same models produce the same results, which is in practice not the case. For these reasons, we use a different form of Venn Abers Predictors, namely Inductive Venn Abers Predictors.

Inductive Venn Abbers Predictors (IVAP) is a version of Venn Aber predictors that use a fixed held out validation set and fits an isotonic regression twice including the validation set and the test sample whose confidence is desired to be calibrated. Simiarly to Venn Abers it fits the isotonic regression, using the probability of the positive class once with an assumed class of 0 and once with an assumed test sample of one. IVAP is a regularised version of isotonic regression and creates an upper and lower bound for the probability of the positive class.

The benefit of having an upper and lower class is extremely useful in the context of binary classification. While stating a prediction with a probability is beneficial, stating a prediction with a probability interval can be even greater. Although if one probability is preferred the probability can be calculated using the solution for the log mini-max for IVAP. To combine solutions, begin by assuming for a test sample i we have inputted the test samples input x_i^* into a neural network and the network has produced probabilites $\tilde{p}$ for the probability of class one and hence $1-\tilde{p}$ for the probability of class zero. Choose the probability for the positive class $\tilde{p}$. Now write $(\tilde{p}, 0)$ assuming the lower class and $(\tilde{p}, 1)$ for the positive class. Now for a validation set $(\{x_i', y_i')\}_{i=0}^{n_1-1}$, input the inputs $\{x_i'\}_{i=0}^{n_0-1}$ into the neural network to return the neural networks associated probabilities $\tilde{p_0}, \ldots, \tilde{p_{n_0-1}}$ for the positive class one. Then fit an isotonic regression to $(\tilde{p_o}, y_0'), (\tilde{p_1}, y_1') \ldots, (\tilde{p_{n_1-1}}, y_{n_1-1}), (\tilde{p}, 0)$ to obtain the lower bounding isotonic function f_0 for the test sample. Similarly fit an isonic regression to $(\tilde{p_o}, y_0'), (\tilde{p_1}, y_1') \ldots, (\tilde{p_{n_1-1}}, y_{n_1-1}), (\tilde{p}, 1)$ to obtain an upper isotonic regression function f_1. These function f_0 and f_1 are then the new score functions specifically for the additionally included test sample. Finally calculate:

$$p_l = f_0(\tilde{p}), \tag{4.14}$$

where here p_l is the lower bound for the model produced probability $\tilde{p}$ for class one and similarly,

$$p_u = f_1(\tilde{p}), \tag{4.15}$$

where p_u is the upper bound for the model produced probability $\tilde{p}$ again for class one. To summarise, we fit an isotonic regreesion to the validation set and the test sample twice, once assuming the test sample belongs to class zero and once assuming the test sample belongs to class one. Furthermore, the isotonic regressios are fit using the model produced probabilities for class one as well as the true and assumed outputs for the validation set and test sample respectively.

One downside of Inductive Venn Abers is that the isotonic regressions need to be fit twice for each individual desired improved confidence. Therefore the total number of isotonic regression to be performed in binary classification is two multiplied by the number of test samples desired. While this is much better than training a model from scratch, it still means that IVAP is a computationally expensive procedure.

Example:

We can also combine the upper and lower bound to create a single probability estimate for the probability of the positive class. The log mini-max solution is:

$$p = \frac{p_u}{1 - p_l + p_u}. \tag{4.16}$$

While there are nuumerous ways to combine the upper and lower probabilities extending the idea to multi-classsification for example using a onevsone approach REF we decide to use the singled valued log mini-max solution and extend in the same method as isotonic regression. That is create K kvsrest binary classifications and then combine these to get an unnormalised vector probability of the classes and then normalise by the rows sum (ref) to achieve the normalised and calibrated probabiities. Finally the confidence and the prediction is then the class with the highest probability the same as in 4.3 and 4.2.

4.4.2 Results

We apply IVAP to the CIFAR-10 dataset using the LeNet-5, VGG-16 and ResNet-50 data augmentation models, the same as in previous sections. We obtain the following results:

Model	mean$\{p_1 - p_0\}$	max$\{p_1 - p_0\}$	IVAP accuracy	IVAP ECE
LeNet-5	0.0043	0.1500	58.73%	1.73%
VGG-16	0.0035	0.2912	85.25%	1.37%
ResNet-50	0.0038	0.2609	84.58%	0.66%

Note that we have 10 classes and 10,000 test set items and we ran 10 kvsall isotonic regressions for each test data example twice (once label 0 and once label 1). Therefore, a total of $2 \times 10000 \times 10 = 200,000$ isotonic regressions was computed storing 100,000 p_0 values and 100,000 p_1 values (10 p_0 and p_1 values for each class). We then took p from the log-minimax and combined these p values together in the same method as isotonic regression. Fortunately with today's speed, performing 200,000 isotonic regressions took only approximately 8 minutes using the efficient Python library functions of the isotonic regression module in scikit learn.

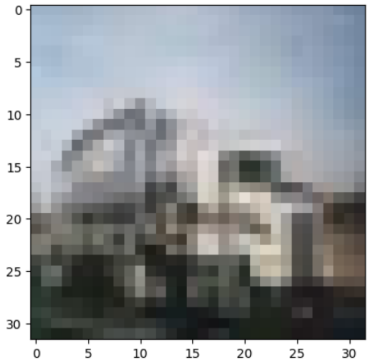


Figure 4.11: Example: Large (p_0, p_1) interval explained

To further understand IVAP (in the binary classification), we look at the 3172 test example in the truck vs all binary classification settings for the ResNet-50 model. This is the example that obtained a maximum $p_1 - p_0$ value of $0.7297 - 0.5227 = 0.207$ which spans approximately 20% of the possible confidence range. This means that the true confidence of this prediction is in the interval (p_0, p_1). In binary classification, we could stop here and use this interval of confidence to inform our decision since we have more information about the confidence and its interval. Since this is multi-classification we need to combine this interval with the other intervals for the other 9 classes. We take the log minimax solution $p = \frac{0.7297}{1-0.5227+0.7297}$ to obtain a value of $p = 0.6046$. Normalising this by the other p values for the other 9 classes, we obtain a normalised IVAP confidence of $\frac{0.6046}{\sum_{k=1}^{10} p^{(k)}} = 0.7781$. Therefore, the adjusted confidence for trucks using IVAP is 0.7781. In this example, we see that IVAP lowered the confidence of the model confidence and the isotonic confidence. We note that in general IVAP tends to be a more regularised version of IVAP and hence is less overconfident about whether a prediction is correct or not.

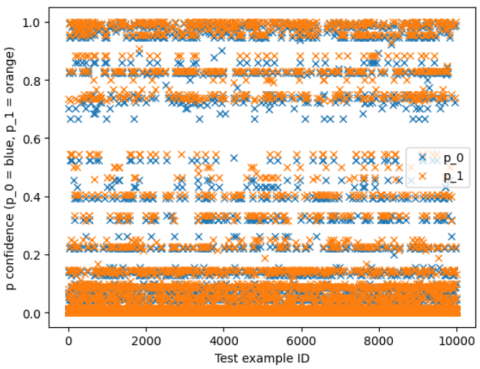
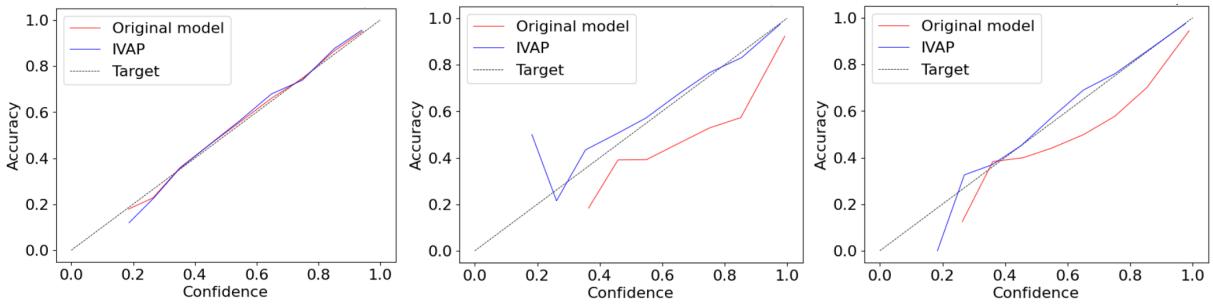


Figure 4.12: ResNet-50 truck kvsall p1vsp0 plot

Also, we can look at a plot for p_0 vs p_1 (p_0 blue, p_1 orange).
By observing the plot we see that the largest intervals between p_0 and p_1 seem to occur away from 0 and 1. Why is this? This is because there are fewer model confidences here therefore the interval is bigger. For example, when the confidence is 0.002 the interval will likely be very small around this value (e.g. (0.0015, 0.0025)). Observing the other kvsall p_0 vs p_1 plots the bigger intervals similarly occur away from 0 and 1.

Again, we can observe the confidence vs accuracy plots for all the models. The red line as usual is the uncalibrated model, and the blue line is the IVAP-calibrated model.

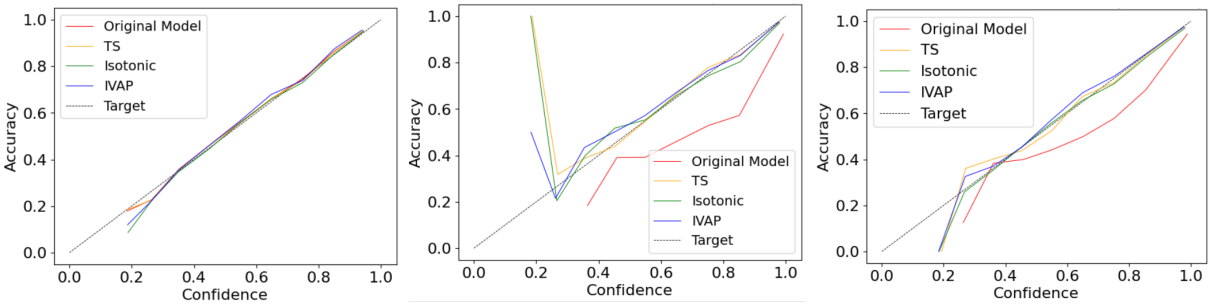


(a) LeNet-5 (b) VGG-16 (c) ResNet-50

Figure 4.13: Inductive Venn Abers predictors 100epoch conf-acc DA plots

This is IVAP a great calibration method that when applied using the model and test data, can have mathematical guarantees of the confidence in the specified interval. Similarly to isotonic regression IVAP works well when the model has inconsistent confidences across classes, and it performs well when there are many classes.

4.5 Comparison of post-processing methods



(a) LeNet-5 (b) VGG-16 (c) ResNet-50

Figure 4.14: ALL methods

In this section we briefly compare the three processing techniques that we have saw

Model	Acc	Iso Acc	IVAP Acc	Uncal	TS C	Iso C	IVAP C
LeNet-5	58.26%	58.78%	58.73%	1.15%	1.13%	0.95%	1.73%
VGG-16	84.76%	85.23%	85.25%	9.36%	1.07%	1.67%	1.37%
ResNet-50	84.71%	84.57%	84.58%	6.64%	0.98%	1.08%	0.66%

Table 4.3: Calibration comparisons

The general results are that temperature scaling performs best on the CIFAR-10 data se, consistently improving the calibration. Furthermore, IVAP and isotonic regression almost always improve the calibration of the model's predictions. One exception shows that IVAP infact increaseed the calibration error for LeNet-5, however we mention that LeNet-5 was already well calibrated to begin with, and did not need necessarily further post processing techniques. Overall these are three good choices to improve a models confidence. If a simple update is required, temperature scaling will typically suffice. If there are many classes with different model confidences for each classes, then isotonic regression or IVAP may be able to restore the model's confidences more precisely. Overall on the CIFAR-10 dataset, all techniques performed well, improving the calibration of the three models.

4.6 Further factors influencing calibration

Now that we have looked at many post-processing calibration techniques, We now look at properties affecting calibration when creating the model. In the following we keep all parameters and settings the same except we the parameter in the specified subsection. We also note that many of these factors influence generalization which directly affects the models calibration. The base parameters are: epoch: 100, optimizer: Adam with learning rate 0.001, Data: Standardized, Batch size: 128, Number of pixels: 32×32, Data augmentation: Enabled, Loss function: categorical crossentropy (NLL), Pretrained weights: Not enabled. We also decrease the learning rate if the validation loss has not decreased in 10 epoches. The dataset used is CIFAR-10. The model is trained on the training set, and the following ECE and accuracies preseneted are on the test set.

Default preprocessing for images for the LeNet-5 and ResNet-50 models is standardization of average mean and standard deviation of whole CIFAR-10 data set. Default preprocessing for images for the VGG-16 model is centering using average mean per pixels colour. The VGG-16 model seemed to perform best with this preprocessing, especially when combined with the Adam optimizer.

We also note that the current state of the art perform on the CIFAR-10 dataset is:. We note that while our results are not exactly the same, we recall that we do not use all of the training dataset, as we split it into training and validation, as we further use the validation data for recalibrating the models.

4.6.1 Transfer Learning

As shown previously, transfer learning improves the accuracy of the model. Since LeNet-5 is a simpler model, it is not ideal for ImageNet-1000. VGG-16 and ResNet-50 have pretained weights available from Keras. When transfer learning it is important that the data is preloaded in the same way.

Model
VGG-16
ResNet-50

Table 4.4: CNN from

ECE	Acc
0.0625	0.8752
0.0855	0.8524

Table 4.5: CNN from scratch

ECE	Acc
0.0819	0.8850
0.0844	0.8681

Table 4.6: Transfer learning

4.6.2 Epoch

We have seen how epoch affects calibration already. This was covered towards the start of chapter 4.

4.6.3 SGD vs Adam

We now compare the optimizers stochastic gradient descent vs Adam. For Adam we use a learning rate of 0.001, and for stochastic gradient descent we use a learning rate of 0.001 and we also use momentum = 0.9. Momentum is recommended with stochastic gradient descent because it helps in stability of convergence. We also decrease (half) the learning rate in stochastic gradient descent if the validation loss has not decreased the minimum validation loss in the last 10 epochs

Model	ECE	Acc
LeNet-5	0.0195	0.5500
VGG-16	0.0894	0.8538
ResNet-50	0.1138	0.6969

Table 4.7: SGD

Model	ECE	Acc
LeNet-5	0.0325	0.6068
VGG-16	0.0625	0.8752
ResNet-50	0.0855	0.8524

Table 4.8: Adam

While Adam optimizer seems to converge faster, achieving a higher accuracy in less epoch, the generalization tends to be worse than SGD [26]. We also note that Adam optimizer struggles to inititate convergence on the VGG-16 model, and often without the right preprocessing steps, gets stuck at 0.10 accuracy. In this case centering each image by the per pixel per colour mean seems to overcome this obstacle.

4.6.4 Batch size

Throughout this report we have used a batch size of 128. It is recommended to use batch sizes that are powers of 2. Let's see how models with different batch sizes of 32, 64 and 128 perform.

Model	ECE	Acc
LeNet-5	0.0080	0.5863
VGG-16	0.0411	0.834
ResNet-50	0.0703	0.8474

Table 4.9: models

ECE	Acc
0.0077	0.6015
0.0454	0.8201
0.0464	0.8113

Table 4.10: batch size 32

ECE	Acc
0.0325	0.6068
0.0625	0.8752
0.0855	0.8524

Table 4.11: batch size 64

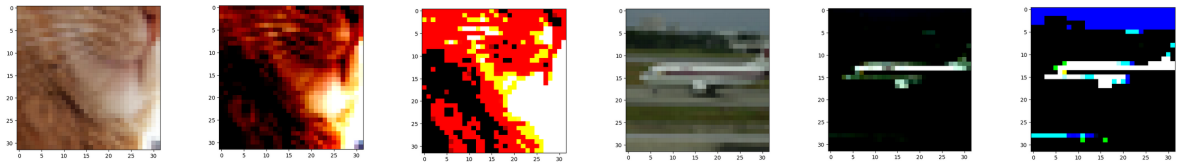
Table 4.12: batch size 128

We also note that a batch size of 32, 64 and 128 had running times of 40s, 36 and QQQ per epoch respectively. Note that a larger batch size calculates a better gradient and hence produces more stable convergence towards the minimum and optimal weights. Although, if the batch size is too large, then the model may begin to memorise the data leading to poor generalisation.

4.6.5 Data augmentation

4.6.6 Normalisation vs standardisation vs centering

In each image a different shade of a colour of a pixel is represented by an integer number between 0 and 255. Before we pass this data to the model, we must normalise the data. After experimenting, the best two performers of data preprocessing was normalizing and standardizing the data. Normalising the data means dividing by 255 to ensure all pixels are a number between 0 and 1. Standardizing the data changes the data into standard normal form which involves calculating the mean μ and standard deviation σ of the training set and computing $z = \frac{x-\mu}{\sigma}$ for each individual pixel in the training set. We then standardise the training, validation and test data by this standardising these values. Note that we standardise the validation and test data by the training mean and standard deviation because we need to ensure that the training data, validation data and test data are from the same distribution. Note that in normalisation



(a) Normalisation (b) Standardisation (c) Centering (d) Normalisation (e) Standardisation (f) Centering

Figure 4.15: Normalisation vs Standardisation example

we have rescaled the data between 0 and 1. In data augmentation we change the brightness by up to 10%, however, changing the brightness will not keep all pixel values in the specified range, therefore we execlude brightness change in normalised data in data augmentation.

The reason why we normalize the data is to reduce the chance of exploding gradients when dealing with high numbers.

Model
LeNet-5
VGG-16
ResNet-50

Table 4.13: models

ECE	Acc
0.0274	0.5631
0.0511	0.8690
0.0611	0.8487

Table 4.14: Normalised

ECE	Acc
0.0325	0.6068
0.0909	0.8418
0.0855	0.8524

Table 4.15: Standardised

ECE	Acc
0.0241	0.5600
0.0625	0.8752
0.0888	0.8910

Table 4.16: Centering

Model	ECE	Acc
VGG-16	0.0606	0.8732
ResNet-50	0.0810	0.8812

Table 4.17: Centering Using VGG-16 or ResNet-50 Imagenet preprocessing

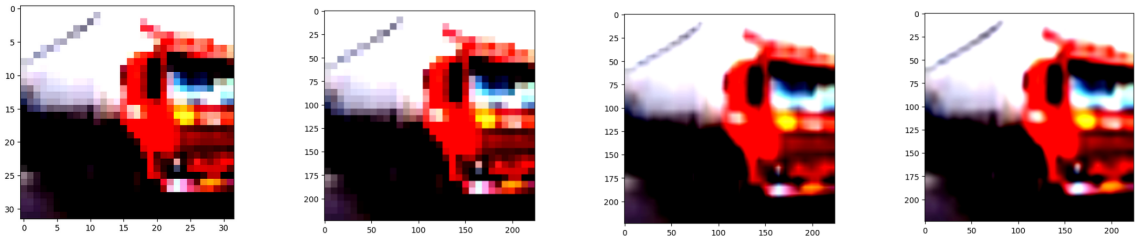
Table 4.17 above indicate imagenet preprocessing (RGB to BGR then subtract per pixel mean) was used rather than Cifar-10 train data subtracted. Note that for Cifar-10 the per pixel mean per colour is: [125.3, 122.9, 113.8]. This can be compared to imagenet's per pixel mean of [123.7, 116.3, 103.53]. Note that [123.7, 116.3, 103.53] = 255 * [0.485, 0.456, 0.406] since [0.485, 0.456, 0.406] is a commonly used pre-processing mean action. We also note that both the LeNet-5 and ResNet-50 models achieved high accuracy and good calibration using just the mean of all pixels 120.7 for Cifar-10, however, VGG-16 failed to achieve high accuracy by subtracting only this total mean. This shows that VGG-16 is sensitive to its colours means and the data-preprocessing heavily affects the VGG-16's model's convergence to optimal weights.

In the ResNet-50 model, by standardizing the data, the results improved dramatically. Show results. We note that in both the original papers of ResNet-50 and VGG-16 the original models were trained with the per pixel mean subtracted [21] [6], however, the images used were higher resolution 224×224 pixels.

4.6.7 Number of pixels

More pixels in an image, means a higher quality image and hence determining the object in the photo becomes easier. Therefore we expect the model to obtain higher accuracy and better generalization when provided with high quality images. Better generalization leads to better calibrated models. Let's see how the number of pixels affects the models calibration:

Case 1: original pixels are 32×32 images. In this case, we will train a model on the 32×32 images, and we will compare it to a model trained on the upsampled 224×224 pixel images. In this case we use interpolative resizing which repeats the pixels 7 times in each row and column. This is called upsampling.



(a) Original (b) Nearest neighbour (c) Bilinear (d) Bicubic

Figure 4.16: Upsampling interpolation methods comparisons

As is shown in upsampling: nearest neighbour interpolation copies the values of the nearby pixels, linear interpolation uses nearby pixels to determine a pixels value through linear interpolations, and cubic interplation uses a cubic polynomial interpolation to compute pixel values using nearby pixels. Typically bicubic produces the smoothest image although it it the most difficult to produce. In Table 4.19 we used 50 epoch and training time was approximately 6 mins per epoch (compared to 32 second per epoch for Table 4.18).

Model	ECE	Acc
LeNet-5	0.0325	0.6068
VGG-16	0.0625	0.8752
ResNet-50	0.0855	0.8524

Table 4.18: 32×32 pixels

Model	ECE	Acc
LeNet-5	0.0491	0.6001
VGG-16	0.0414	0.8327
ResNet-50	0.0604	0.8950

Table 4.19: Upsampled NN 224×224 pixels

Calibration is a key part of any neural network. In this chapter we have shown that there are many methods to calibrate a model to produce more accurate confidences for its predictions. Therefore, ensuring these uncertainties are calibrated for deep neural networks is critical for appliable real world situations.

Chapter 5

Comparisons of post-processing calibration techniques

5.1 Comparisons

In this section we compare the calibration techniques for many different models and datasets. Some datasets are bias, some some etc. The models are LeNet-5 (non-pretrained), VGG-16 (pretrained on the imagenet dataset) and ResNet-50 (non pretrained) and all models were trained for 100 epoch. These are the same models that we have used throughout chapter 4. The results for a range of datasets (including CIFAR-10) are below (note that temperature scaling does not change the accuracy). CIFAR 100 [11] - all results below were are a separate hold out test set after training the model and performing calibration techniques on the validation set (as-usual).

Dataset	Epoch	Model	Acc	Iso Acc	IVAP Acc	Uncal	TS C	Iso C	IVAP C
CIFAR-10	100	LeNet-5	0.5476	0.5431	0.5425	0.0194	0.0069	0.0160	0.0106
CIFAR-10	100	VGG-16	0.8629	0.8619	0.8621	0.1176	0.0224	0.0257	0.0229
CIFAR-10	100	ResNet-50	0.7231	0.7233	0.7234	0.2111	0.0209	0.0373	0.0343
CIFAR-100	13	VGG-16	0.6947	0.7066	0.7071	0.1083	0.0135	0.0369	0.0250
Scenes	100	LeNet-5	0.6807	0.6840	0.6840	0.0197	0.0141	0.0215	0.0181
Scenes	5 from 60	VGG-16	0.9247	0.9270	0.9260	0.0060	0.0058	0.0166	0.0174
Scenes	60	ResNet-50	0.8887	0.8910	0.8903	0.0470	0.0135	0.0221	0.0128

Table 5.1: Calibration comparisons

In CIFAR-100 13-epoch VGG-16 the images were upscaled to 224*224 using nearest neighbours interpolation to produce a higher accuracy and increase generalization.

The scenes dataset involves 6 classes of natural scenes. Natural scenes does not refer to nature produced vs man-made but rather to scenes that people observe while in natural modes of operation. Also, we see that in the Scenes dataset, using VGG-16 (previously said LeNet-5) we achieved an early high accuracy on the validation set, which the model failed to surpass during the next 40 epochs. Therefore, the model stopped early and returned from the 45th epoch to the 5th epoch. This is not too surprising, because we used transfer learning on all VGG-16 models, therefore the weights were in this case already close to the optimal weights for this model for this dataset. We also see that its uncalibrated ECE is exceptionally low and outperforms the post processing techniques of isotonic regression and IVAP. One reason why this model performed so well, is because it generalized well from training data to validation data. This shows that with the correct model, post processing methods are not always necessary. Therefore, choosing the correct model for the correct dataset can go a long way towards having a final calibrated model.

In this report, we went from introductory machine learning to building image classification models achieving high accuracy to improving calibration. The general findings were that the post-processing methods almost always improve model calibration. Therefore one such always consider exploring a post-processing method to the produced confidence of the model. Furthermore, the many different configurations of the model parameters can play a role in the model's calibration. In general, for many neural networks temperature scaling successfully improves the model's predictions and should always be considered when calibrating a model.

Bibliography

[1] Yu Bai et al. *Don't Just Blame Over-parametrization for Over-confidence: Theoretical Analysis of Calibration in Binary Classification*. 2021. arXiv: 2102.07856 [cs.LG]. URL: https://arxiv.org/abs/2102.07856.

[2] John Scott Bridle. "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition". In: *NATO Neurocomputing*. 1989. URL: https://api.semanticscholar.org/CorpusID:59636530.

[3] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[4] Chuan Guo et al. "On Calibration of Modern Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 1321–1330. URL: https://proceedings.mlr.press/v70/guo17a.html.

[5] Trevor Hastie. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

[6] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[7] Donald E. Hilt et al. *Ridge, a computer program for calculating ridge regression estimates*. Vol. no.236. https://www.biodiversitylibrary.org/bibliography/68934 — Caption title. Upper Darby, Pa, Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977, 1977, p. 10. URL: https://www.biodiversitylibrary.org/item/137258.

[8] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[9] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[10] Alex Krizhevsky. "Convolutional Deep Belief Networks on CIFAR-10". In: 2010. URL: https://api.semanticscholar.org/CorpusID:15295567.

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

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[13] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

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

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[16] D. Lu and Q. Weng and. "A survey of image classification methods and techniques for improving classification performance". In: *International Journal of Remote Sensing* 28.5 (2007), pp. 823–870. DOI: 10.1080/01431160600746456. eprint: https://doi.org/10.1080/01431160600746456. URL: https://doi.org/10.1080/01431160600746456.

[17] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[18] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[19] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. 2015. arXiv: 1409.0575 [cs.CV]. URL: https://arxiv.org/abs/1409.0575.

[20] Connor Shorten and Taghi M Khoshgoftaar. "A survey on image data augmentation for deep learning". In: *Journal of big data* 6.1 (2019), pp. 1–48.

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

[22] Vladimir Vovk and Ivan Petej. "Venn-abers predictors". In: *arXiv preprint arXiv:1211.0025* (2012).

[23] Vladimir Vovk, Ivan Petej, and Valentina Fedorova. "Large-scale probabilistic predictors with and without guarantees of validity". In: *Advances in Neural Information Processing Systems* 28 (2015).

[24] Cheng Wang. "Calibration in deep learning: A survey of the state-of-the-art". In: *arXiv preprint arXiv:2308.01222* (2023).

[25] Bianca Zadrozny and Charles Elkan. "Transforming Classifier Scores into Accurate Multiclass Probability Estimates". In: (2002).

[26] Pan Zhou et al. "Towards theoretically understanding why sgd generalizes better than adam in deep learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21285–21296.