Cracking CAPTCHAs

Simon Turner
Supervisor: Richard Clayton
COM3600
May 3, 2017

This report is submitted in partial fulfilment of the requirement for the degree of Computer Science by Simon Turner.

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Simon Turner
Signature:
Date: May 3, 2017

Abstract

CAPTCHAs are a key part of preventing spam by non-human users (bots) on the web. They aim to distinguish between humans and bots by presenting challenges that cannot be completed by a bot, but can be easily done by a human. Many existing CAPTCHA variants have already been shown to be ineffective at this task, but little research has been done to solve more recent CAPTCHAs, such as those produced by Google's reCAPTCHA project. These newer CAPTCHA schemes rely on software being unable to solve image or audio recognition tasks, but advances in machine learning have proven similar tasks to be more feasible for a machine.

This project aims to use modern machine learning techniques, combined with browser automation software, to prove or disprove the hypothesis that, "Google's reCAPTCHA test is unsolvable by software." In order to test this hypothesis, this project aims to construct a piece of software that can solve Google's reCAPTCHA.

Acknowledgements

I would first like to thank my supervisor, Richard Clayton, for all of his advice, listening to me talk through why my neural network was misclassifying everything as amphitheatres, amongst many other issues small and large, and for providing the project brief in the first place.

I would also like to thank any poor individual who, in the previous months, has had to listen to me ramble on about CAPTCHAs, image recognition, or street sign images. Particularly those of you who don't even study Computer Science.

I would like to thank reCAPTCHA and Google, for unwittingly allowing me to spam reCAPTCHA without rate limiting or locking me out in any way. And also for creating reCAPTCHA in the first place, of course.

Finally, I would like to thank my friends, my brother, and my parents - without any of you being who you are, I wouldn't be who I am.

Contents

1 Introduction **1**

2 Literature Review **2**
2.1 CAPTCHAs . . . 2
2.1.1 Distorted Text CAPTCHAs . . . 2
2.1.2 Audio CAPTCHAs . . . 3
2.1.3 Question CAPTCHAs . . . 3
2.1.4 Image CAPTCHAs . . . 3
2.1.5 Google's reCAPTCHA . . . 3
2.2 Image Recognition . . . 5
2.2.1 Machine Learning Concepts . . . 5
2.2.2 Image Processing Concepts . . . 7
2.2.3 Support Vector Machines . . . 8
2.2.4 Neural Networks . . . 10
2.2.5 Convolutional Neural Networks . . . 11
2.2.6 Conclusion . . . 17
2.3 Semantic Similarity . . . 18
2.3.1 Word similarity . . . 18
2.3.2 WordNet . . . 19
2.3.3 Other ontology based approaches. . . . 19
2.3.4 word2vec . . . 19
2.3.5 Latent Semantic Indexing . . . 20
2.3.6 Conclusion . . . 20

3 Analysis **21**
3.1 Programming Language . . . 21
3.1.1 Python . . . 21
3.1.2 MATLAB . . . 21
3.1.3 R . . . 21
3.1.4 Conclusion . . . 21
3.2 CAPTCHA Input . . . 22
3.2.1 Browser Automation Software . . . 22
3.2.2 Datasets . . . 23
3.2.3 Libraries for Machine Learning . . . 24
3.2.4 Requirements . . . 25
3.2.5 Evaluation . . . 26

4 Design **28**
4.1 CAPTCHA Interaction . . . 28
4.2 Neural Network . . . 32
4.2.1 Minibatch Training . . . 32

4.2.2 Loss function . . . 32
4.2.3 Backpropagation method . . . 33
4.2.4 Learning rate and weight decay . . . 34
4.2.5 Preprocessing . . . 35
4.2.6 Validation . . . 35
4.2.7 Dataset . . . 35

5 Implementation **37**
5.1 CAPTCHA test web page . . . 37
5.2 Captcha Interaction . . . 37
5.2.1 The start function. . . . 37
5.2.2 CaptchaCracker Initialisation and Setup . . . 37
5.2.3 Initialising the Neural Network with an existing weights file. . . . 38
5.2.4 Initial Checkbox . . . 38
5.2.5 Checking if the CAPTCHA has been passed. . . . 38
5.2.6 Gathering information about the new image challenge. . . . 38
5.2.7 Preprocessing . . . 40
5.2.8 Getting predictions. . . . 40
5.2.9 Clicking the correct checkboxes. . . . 41
5.2.10 Refreshing the checkboxes. . . . 41
5.2.11 Choosing to reload or verify. . . . 41
5.2.12 Handling exceptions. . . . 41
5.2.13 Using time.sleep() for controlling the speed of interaction. . . . 42
5.3 Neural Network . . . 42
5.3.1 Building the network. . . . 42
5.3.2 Training the network. . . . 42
5.4 Config . . . 43
5.5 CAPTCHA viewer . . . 44
5.6 Requirement Fulfilment . . . 44

6 Results **46**
6.1 Accuracy at solving CAPTCHAs. . . . 46
6.1.1 Random Guessing . . . 46
6.1.2 Probability thresholds. . . . 46
6.1.3 The impact of security settings. . . . 47
6.2 Neural network performance . . . 48
6.2.1 Architecture . . . 50
6.2.2 Dataset . . . 50
6.3 Conclusion . . . 53
6.4 Improvements . . . 53
6.4.1 Multilabel classification . . . 53
6.4.2 Data augmentation . . . 54
6.4.3 Dataset improvements . . . 54
6.4.4 Parallelisation . . . 55

7 Conclusion **56**
7.1 Findings . . . 56
7.2 Final Conclusions . . . 57

8 Appendix **58**
.1 Convolution . . . 58

List of Figures

2.1 Different kinds of distortion effects can be used for different CAPTCHAs. While these CAPTCHA schemes were still popular, it was common for site owners to constantly update the distortion to outsmart bots. [3] 2
2.3 Linear separable data in three dimensions.[8] 5
2.4 The XOR Problem - the data points here cannot be separated by one line, two lines would be required (as shown on the diagram). As such, the data is not linearly separable.[9] 6
2.5 An example of HoG features for an image. Taken from Dalal and Triggs paper on using HoG for person detection.[12] 8
2.6 An example Vidoop CAPTCHA.[15] 9
2.7 A feedforward neural network, with all layers fully connected. Weights and biases for each neuron not shown. Image taken from Neural Networks and Deep Learning[17] 10
2.8 LeNet uses a series of convolutional layers, interspersed with subsampling layers (subsampling layers will be discussed later), ending with a fully connected output layer. Example receptive fields are shown outlined in black. 11
2.9 The curve of the logistic function - the gradients at either extreme end on the x axis are very flat, which in terms of a neural network means very small changes in output from the neurons, and thus a very slow learning rate if the weights start completely wrong.[20] 12
2.10 The curve of the tanh function, with similar issues as the logistic function.[21] . . 12
2.11 Error rate vs epoch number for ReLU (solid line) vs Tanh (dashed line). An epoch is a forward pass through the network and a backward pass (i.e. backpropagation).[19] 13
2.12 The gradient of the ReLU function is constant after an initial change, meaning the change in output values is constant for a larger amount of weights and inputs (the inputs to this activation function).[24] 13
2.13 Diagram showing the maxpooling operation applied to an example input matrix.[26] 14
2.15 An example Inception module using 1x1, 3x3 and 5x5 convolutional layers, as well as a max-pooling layer.[33] 16
2.16 Training and test error on an image recognition task with 20 layer and 56 layer neural networks.[34] 16
2.17 The Xception architecture. The depthwise separable convolutions can be seen easily in the first section, where there are three sets of SeparableConv layers applied to the input sequentially, each having a 1x1 convolution applied to it. The diagram is meant to be read section by section top to bottom. 17

3.1 This is an example of a CAPTCHA which was actually correct, but was perhaps succeeded by an incorrect CAPTCHA, leading to no tickmark. 27

4.1 A flow diagram showing the main logic of the program. 29

4.2 A diagram showing the structure of the program, its classes and modules. Classes have their attributes described, and are written in CaptainCapital style, while modules use snake_case. The functions and methods have been left out of the diagram for clarity. Similarly, only the guesses.json file is included (and not the labels or categories files, for example) because that illustrates the link between the CAPTCHA viewer and the rest of the program. 31
4.3 Graph showing the top-1 validation accuracy over epochs using the Adam backpropagation optimiser. 33
4.4 Graph showing the top-1 validation loss over epochs using the Adam backpropagation optimiser. 33

6.1 Top-1 accuracy per epoch during training. 48
6.2 Top-5 accuracy per epoch during training. 48
6.3 Loss per epoch during training. 48
6.4 Top-1 accuracy on the validation set per epoch. 49
6.5 Top-5 accuracy on the validation set per epoch. 49
6.6 Loss per epoch on the validation set 49
6.7 A chart of the different queries seen in 726 runs of the CAPTCHA cracker. . . . 51
6.8 A particular CAPTCHA, asking for images of mountains, with the predictions from the network trained on the much larger dataset. 52
6.9 Another CAPTCHA from the large dataset tests, this time asking for store fronts. 53
6.10 The image on the middle left hand side shows both what could be construed as sand (but may be a field of some kind) and a mountain in the background. Whatever the foreground is, the important part is the mountain and this is missed as only a single label is assigned to the image. 54

List of Tables

2.1 Error rates of several CNN architectures on ImageNet. Taken from Xception's[35] paper and AlexNet's[19] 18
2.2 Comparing the VGG 19 layer architecture with Xception for a subset of the dataset. 18

4.1 SGD here refers to Stochastic Gradient Descent with Nesterov Momentum. . . . 33
4.2 Comparison of different learning rates. Accuracy and loss scores are on validation sets. 34
4.3 Comparison of different weight decay rates. 34

6.1 Correct percentage of 1000 completely random attempts. 46
6.2 Percentage of CAPTCHAs solved with different probability thresholds used to filter the results of the neural network. 47
6.3 Impact of security settings on correct percentages of the CAPTCHA cracker. . . 47
6.4 Accuracy and loss on the entire dataset, trained using 0.01 learning rate and 1^{-6} decay rate, for 10 epochs (equivalent to twice through the whole dataset). 50
6.5 Results of using the large dataset on CAPTCHAs. 52

Chapter 1

Introduction

Designing a test to tell a human from a machine is a long standing problem in Computer Science. One of the earliest attempts, Alan Turing's Turing Test, was based off a Victorian parlour game, and so any human had time to make a decision as to whether they were talking to a human or a machine. In the age of the Internet however this is impractical. A human cannot judge whether every individual creating an account on a website, or submitting any other web form, is a human. Furthermore, allowing software to post on Internet forums, create accounts, and do anything that a human user can do can lead to extreme issues. Such software (or bots, as they are commonly referred to) can flood websites with spam (estimated to be about 15% of the web's content)[1], spread malware, phish for human users' details, or even take websites down with Distributed Denial of Service attacks. It is clear: preventing malicious bots from accessing parts of webpages is important, and doing it manually via human judgement is completely impractical on such a large scale. These ideas spawned the idea of CAPTCHA - a Completely Automated Public Turing test to tell Computers and Humans Apart. The concept behind CAPTCHA is simple - design an extra part to a web form, or other part of a website that you wish to prevent bots from using, that presents a challenge that a bot cannot solve and a human can with relative ease. However, advances in machine learning, and the availability of massive amounts of data to use for such methods, have put a strain on the challenges traditionally seen as unsolvable by bots. The response by many CAPTCHA designers was to make harder and harder challenges, which gradually became unsolvable by humans not just bots. Recently, CAPTCHA designers have turned to newer approaches - using images of everyday objects and real world locations, or distorted audio. The most prominent example of this is the reCAPTCHA project. Although this newer form of CAPTCHAs may well be easier for humans than very distorted text recent years have seen advances in image recognition - a key part of reCAPTCHA's challenge. The aim of this project is to use modern machine learning approaches and browser interaction software to attempt to automatically solve CAPTCHAs made by the reCAPTCHA project, testing the hypothesis that, "Google's reCAPTCHA test is unsolvable by software". If the software manages to solve CAPTCHAs, then this newer form of CAPTCHA, while currently the cutting edge of CAPTCHA design, is not sufficient to defend against current bots. Conversely, if the software cannot solve CAPTCHAs given by Google's reCAPTCHA, then it is most likely a sufficient defence against current bots.

Chapter 2

Literature Review

There are a wide variety of CAPTCHAs in existence and as such there have been many attempts to crack them, with varying levels of success. A large quantity of research has been done on breaking existing CAPTCHAs, and also on related fields such as image recognition. So, it is important that the approaches that have been used for this and similar problems are assessed, in order to decide on an appropriate approach for this project.

2.1 CAPTCHAs

A CAPTCHA, or Completely Automated Public Turing test to tell Computers and Humans Apart, is designed to be "an automated test that humans can pass, but current computer programs can't pass."[2] There are several common ways of designing such tests, such as requiring the user to identify the characters in distorted text, or to identify characters spoken in audio, or images of common objects.

2.1.1 Distorted Text CAPTCHAs

This kind of CAPTCHA challenges the user to identify the characters in an image. Sometimes the characters form a word (or several words), but these CAPTCHAs can just be random letters and numbers. There are a wide variety of different distortions applied to the characters and the background of the image. Some examples of this kind of CAPTCHA can be seen in figure 2.1. There are serious limitations to this kind of CAPTCHA. The main limitation being that they

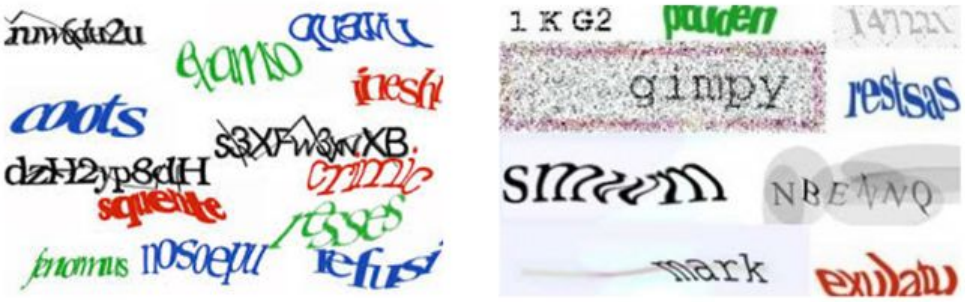


Figure 2.1: Different kinds of distortion effects can be used for different CAPTCHAs. While these CAPTCHA schemes were still popular, it was common for site owners to constantly update the distortion to outsmart bots. [3]

can be difficult to solve for humans[4] while being relatively trivial for a machine.[5][6]

2.1.2 Audio CAPTCHAs

Audio CAPTCHAs often accompany visual CAPTCHAs to improve accessibility for visually impaired users. A common type of audio CAPTCHA involves the user listening to several characters being read to them, and having to identify the characters. Varying distortion effects are applied to the audio, much like a distorted text CAPTCHA.

2.1.3 Question CAPTCHAs

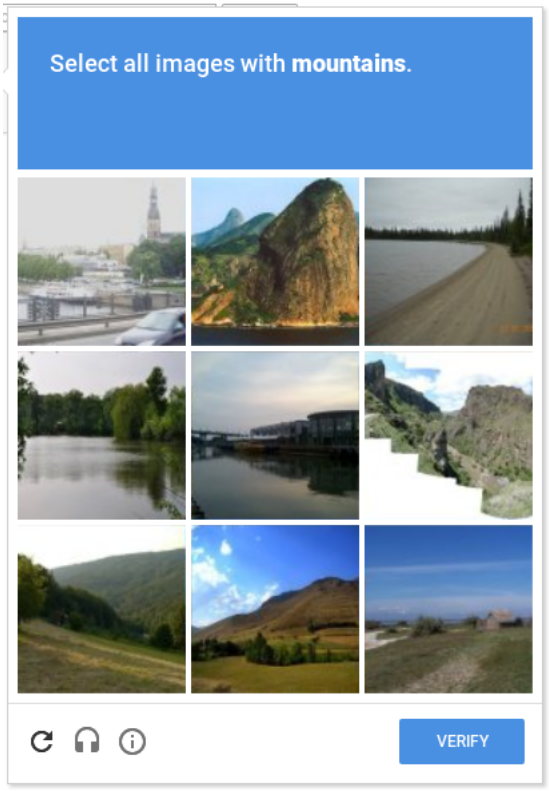
Question CAPTCHAs are challenges in the form of simple questions, such as "What number is 2nd in the series 35, 19 and thirty two?". Sometimes these CAPTCHAs are written to be more ambiguous to parse, other times they are simple questions which rely on obscurity - hoping that a bot trying to fill in the form will not be designed to look for the kind of CAPTCHA they use, so it will not be able to answer the question correctly.

2.1.4 Image CAPTCHAs

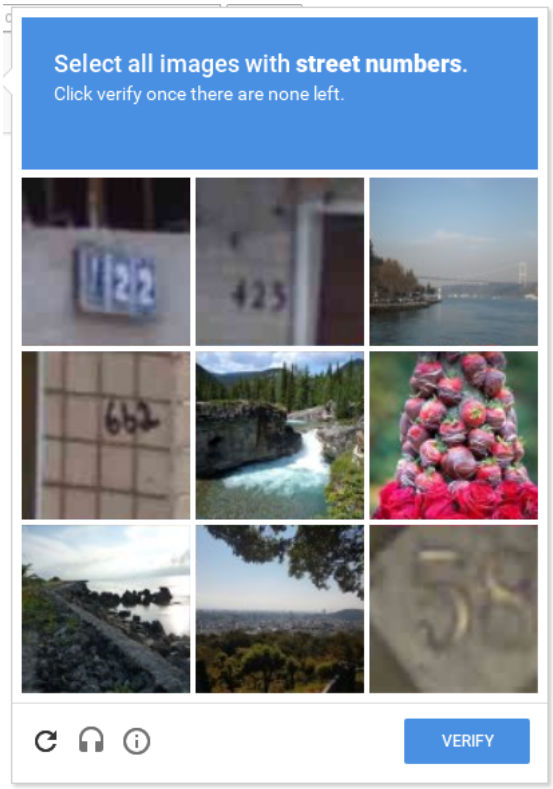
A newer kind of CAPTCHA challenges the user to identify images that fit a certain query, for example, "Select all the images with rivers." Sometimes the user has to enter a series of letters which are overlaid on the images rather than clicking on the images. The number of images, and the type of images shown, vary between CAPTCHAs. Generally though the images are of objects, animals, or real world locations that could be easily recognised. This is a difficulty with designing image CAPTCHAs however, as images need to be ambiguous enough to not be recognised by software, while also needing to be easily recognised by any legitimate user of whatever the CAPTCHA is protecting.

2.1.5 Google's reCAPTCHA

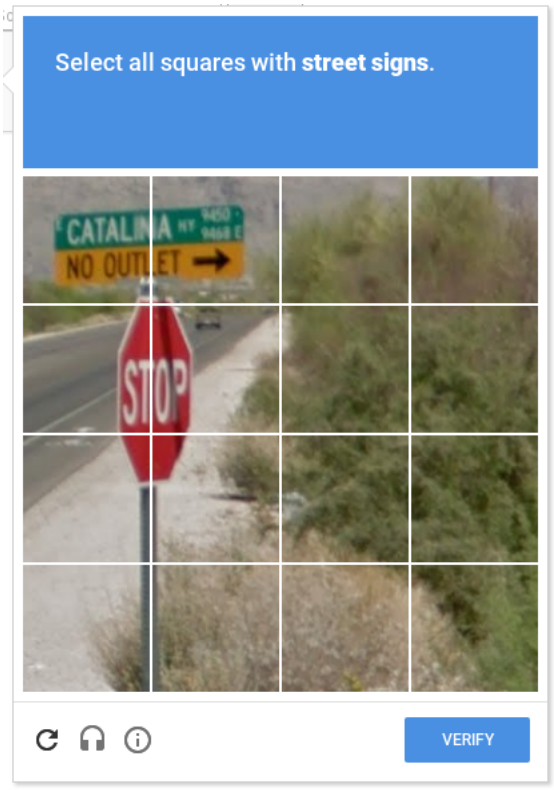
The reCAPTCHA project combines several CAPTCHAs to make their challenge. Initially a user must tick a box, after which the CAPTCHA analyses factors to do with the user's browser, including cookies, user-agent information and how the user moved the mouse and clicked on the checkbox.[7] If analysis done on these metrics suggests that the user is a bot, an image CAPTCHA is presented, which consists of a grid of images (either all different, or separate sections of the same larger image). There are several kinds of image CAPTCHA used, which are shown in figure 6.7. After a large number of mistakes have been made, a distorted text CAPTCHA is shown instead of an image CAPTCHA.
The first stage of the Google's reCAPTCHA is largely "security by obscurity", that is, what data it is gathering and what method is being used to analyse that data is unknown, despite attempts to reverse engineer it. Some attempts have been made previously to break this part of the reCAPTCHA using forged cookies, but the method used by the researchers has been fixed. This project will focus on trying to break the main part of the reCAPTCHA scheme which is designed to be completely unsolvable by a bot - the image challenge.



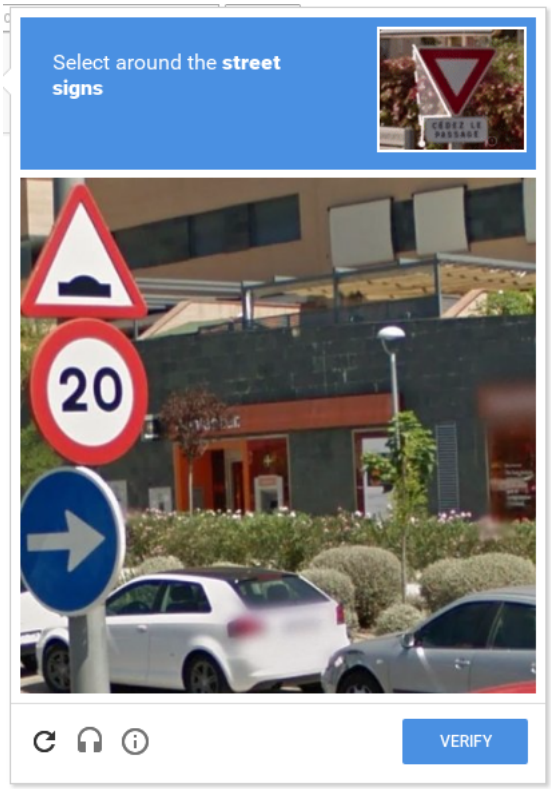
(a) This kind of image challenge asks you to select all the images and then click verify.



(b) This is a variant of figure 2.2 that requires you to keep clicking (as new images load after you click to replace the old ones) on every image that has some object in.



(c) Not all of the images used are separate, some are parts of the same image.



(d) Some challenges require you to click and drag around a part of the image.

2.2 Image Recognition

The main part of any effective approach to cracking reCAPTCHA's image challenge is image recognition, due to the fact that it presents a wide variety of images and asks for the user to click on which match a query. A sensible approach to dealing with this challenge would be to label all of the images, and then select the ones which match the query. There has been much research done on the problem of image recognition, so there are several algorithms that could be used to approach this problem.

2.2.1 Machine Learning Concepts

In order to understand how this problem might be approached, it is helpful to understand how a learning algorithm might be devised, and the different problems and concepts involved in devising such an algorithm.

Linear Separability

Data is linearly separable if it can be separated into two classes with a single hyperplane. An example of linearly separable data in three dimensions is shown in figure 2.3, showing the hyperplane that separates the data.

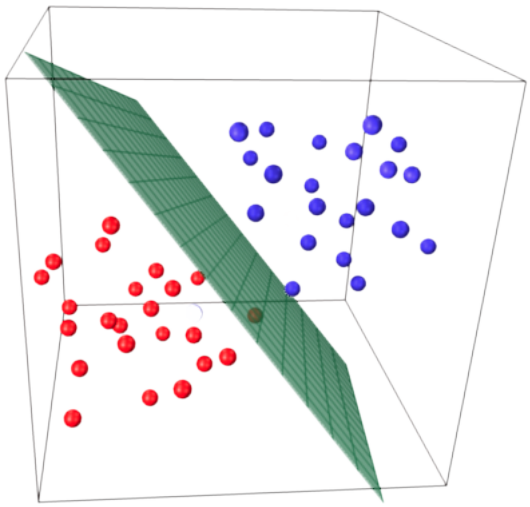


Figure 2.3: Linear separable data in three dimensions.[8]

Some data cannot be linearly separated[1], a classic example of such data is shown in figure 2.4.

Supervised and Unsupervised Learning

Supervised learning on a classification problem such as image recognition uses labelled sets of data to learn patterns in the data to make a model of the data that can be used to classify new examples. Samples can be labelled with a single label, or with multiple labels, each specifying a part of the sample which they refer to. These methods learn by feeding samples through a training algorithm, and comparing the output of that algorithm to the expected output (i.e. the label given to the sample). This comparison is then used to improve the model, by adjusting whatever parameters the model has, usually by a small amount, called a learning rate. Unsupervised learning is similar in that a training algorithm is still used, updating the

[1]Without changing the dimensions of the data, which is a method used by the Support Vector Machine approach described later.

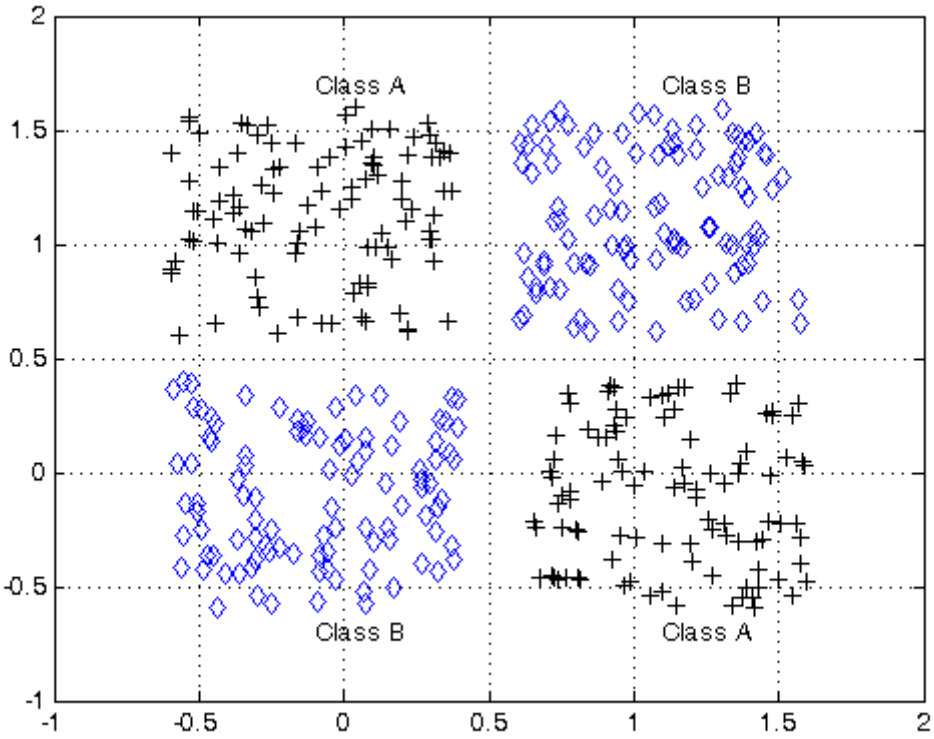


Figure 2.4: The XOR Problem - the data points here cannot be separated by one line, two lines would be required (as shown on the diagram). As such, the data is not linearly separable.[9]

parameters of the model using a learning rate and some input samples. However, the data samples are not labelled. Unsupervised methods instead find correlations in the data and use these to cluster samples, ideally producing a cluster per class, which can then be used to predict new data by judging the new sample's similarity to each of the clusters. Unsupervised methods do not produce class labels when predicting new data however. Supervised learning methods are more common for image recognition problems, so this project will focus on using those approaches.

Training, Testing, and Validation

Conventionally, datasets are split into training, testing and validation sets. Training sets are used to build a model of the data; the exact process varies a lot between different algorithms. This is often the largest part of the dataset. If the training set is too small, the model is only built based on a small amount of data, and so may be unable to correctly classify new examples. The bigger the training set the more likely the classifier is to be able to find some similarity to something that it "saw" during training[10]. That said, if the testing set is too small, the performance statistics of the classifier (such as classification accuracy) will have a larger variance. The validation set is normally split from the training set, so that a dataset is available to test the model for the purposes of tweaking hyper-parameters (parameters that control how the model learns, such as the learning rate). It is important to do such tweaking with a validation set rather than the testing set in order to avoid the model changing to fit the test set, which is meant to test how good the finished model is. The validation set is also used to give feedback during training allowing for the training process to be stopped early if metrics calculated using the validation set are at a certain value. The test set in comparison is meant

to judge how generalisable the finished model is - how well it will cope with new data from the problem domain.

2.2.2 Image Processing Concepts

In addition to understanding the concepts behind the design of a classifier, it is useful to understand the ways that image data might be represented.

Image Descriptors

There are lots of ways of representing an image. A common method involves representing the image in three dimensions, with each pixel in the image having three values (red, green and blue values between 0 and 255). This method, often done using a nested array. This can lead to large amounts of data points for even a relatively small image (for example, a 200x200 colour image would be 120,000 values). As such, there are several methods to capture the important features of an image while using a much smaller representation than all of the values for the image.

Edge Histograms

An edge histogram is the distribution of different types of edge in an image. Usually this would be 5 classes: vertical edges, horizontal edges, 45 degree edges, 135 degree edges, and non-directional edges. Applying a filter (using convolution) to the black and white version of the image is an effective method of finding a particular edge, this can be done with several filters to get the distribution of all the classes of edge.

Colour Histograms

A colour histogram is the distribution of the different colours in an image. The main complication with this descriptor is eliminating factors that might distort the histogram, such as lighting. Representing images using a different colour value system (such as CbCr) rather than RGB is one way of lessening the effect of lighting on a colour histogram.

Colour Moments

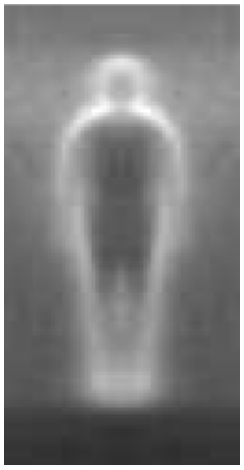
Colour moments are measures of the colour distribution of an image. The first colour moment is the mean average colour in the image, the second is the standard deviation of the colours in the image, the third is the skewness of the colours (how asymmetric the colour distribution is), and the fourth (and usually final) is Kurtosis (how high the distribution is compared to the average).

GIST Descriptor

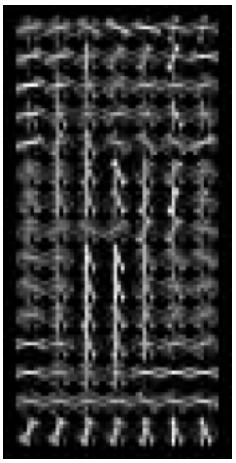
GIST descriptors are based off a paper by Oliva et al[11] which describe an image based on 5 parameters: naturalness (man-made or natural), openness (very open scene like a beach, or very enclosed like a forest), roughness (size of major elements, complexity of scene), expansion (perspective - completely flat image or image with depth), and ruggedness (deviation of ground with respect to the environment).

HoG Features

Histogram of Gradient features are often useful for finding the shape and appearance of objects. Images have intensities of colour in certain areas (such as edges) which becomes less intense in areas of background - an example of this is shown in 2.5a This change in intensity can



(a) An example of an image of a person in grayscale. The edges can be clearly seen due to the increase in intensity.[12]



(b) The associated gradients for the image.

Figure 2.5: An example of HoG features for an image. Taken from Dalal and Triggs paper on using HoG for person detection.[12]

be described by a gradient. HoG works by dividing the image into small regions, and then calculating a histogram of the different gradient directions for the pixels of the region. Some normalisation can be applied to adjust for different light levels.

2.2.3 Support Vector Machines

One supervised learning algorithm which can be used for classifying images is a Support Vector Machine.

A support vector machine works by finding the optimal separating hyperplane, which is defined as, "the one [separating hyperplane] for which the distance to the closest point is maximal"[13], where a separating hyperplane divides the data so all the data with the same label are on the same side of the hyperplane. Of course, it is perfectly plausible that there is data that is not linearly separable, that is, there is no hyperplane that can separate the two classes. In that case, a support vector machine approach applies a function to the data which transforms it into higher-dimensional data. The exact dimensionality of the data depends on the transformation function applied. In this higher-dimension form, an optimal separating hyperplane can be found, thus classifying the data. It is also plausible that the data separates into more than two classes. In such a case, the two main methods are "one vs one" or "one vs all" classification. A one vs one approach trains a different classifier for every pair of classes within the dataset, whereas a one vs all approach trains a classifier for each class that separates the class from all the other classes in the dataset.

Support Vector Machines have been used for large scale image recognition. Lin et al[14] propose a good approach for using SVMs to recognise images from ImageNet (a massive image dataset with 1000 classes of image). Their system used HoG features amongst other methods of feature selection and reduction, with a top-1 accuracy of 52.9% and a top-5 accuracy of 71.8%. While these are impressive accuracy scores on such a large dataset, there are some issues with this approach. Notably the fact that the training process took a week using 12 machines with 8-core CPUs. Overall though, SVMs with the right feature selection and augmentation methods can be successful on large datasets.

SVMs have been used on CAPTCHAs as well as general image recognition, such as Merler and Jacob's paper[15] on cracking the Vidoop CAPTCHA, which is a similar kind of image based CAPTCHA. This attempt met with limited success, with an accuracy rate of 3%.

Merler and Jacob start by splitting the CAPTCHA image into its constituent images, and then split each image into an image with the character and the actual image.

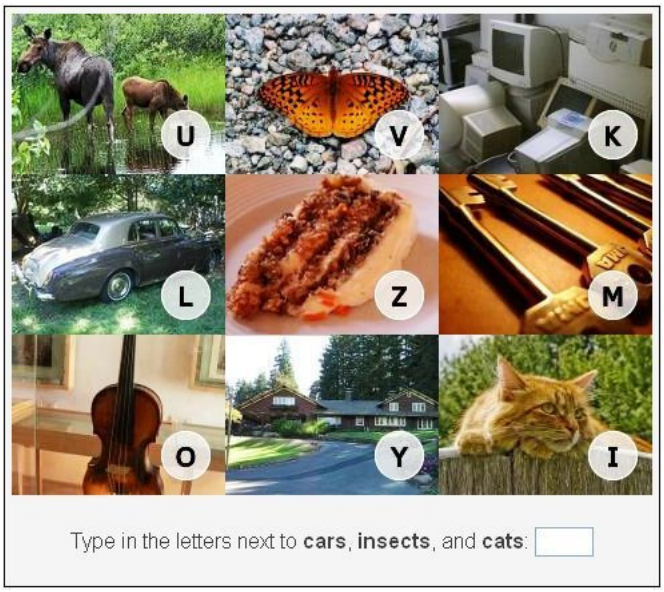


Figure 2.6: An example Vidoop CAPTCHA.[15]

The image recognition is done using a Support Vector Machine, trained on certain features of the images. Several different image features were tried, specifically colour histograms, colour moments, edge histograms, and GIST descriptors. An SVM was then trained using 500 images for each class of image, plus 500 randomly selected from all the other classes. This was done for each of the image feature types. Interestingly, although the CAPTCHA success rate was only 3%, the SVM's classification accuracy using GIST descriptors of the images was 34.7%. However, accuracy for recognising pairs and triplets of images successfully dropped off sharply. A certain amount of the low accuracy rate is due to the training set, however, as images for the training set were downloaded from an image hosting site (Flickr), only going by site users' tags. This could have led to a potentially substantial amount of incorrectly labelled images. The training set was also small, with only 500 images per class, with 26 classes of image identified by Merler and Jacob, leading to a total training set size of 13,000. This is definitely too small by today's standards, and as previously mentioned a larger training set is positively correlated with a higher accuracy. Not only is the size of the dataset a problem, but also the number of classes - 26 classes of image does not cover many of the different images the CAPTCHA might present. Some criticism can also be levelled at a key problem of feature selection. While feature selection approaches (like getting the colour histogram of an image, or the GIST descriptor) attempt to capture the key features of an image that identify it, such methods do not always succeed. Using feature selection methods based on things that humans think make images distinguishable (edges, colour differences etc) sometimes works, but there may be subtle differences not captured by these methods which are lost in the process of reducing an image to one of these descriptors. An example of this can be seen in another image recognition task (recognising the gender of pedestrians from photos). This paper by Antipov et al[16] compares using hand crafted feature descriptors (in this case Histogram of Oriented Gradient features - HoG for short) against learned features from a convolutional neural network, and performs considerably worse when it comes to testing on unseen data. The details of convolutional neural networks will be discussed later, but this shows that an approach that uses hand-crafted feature selection or reductiom can be less effective than an algorithm which learns it's own image filters to use for feature reduction. In order to discuss convolutional neural networks, a discussion of the basic concepts of neural networks in general is helpful.

2.2.4 Neural Networks

A neural network is a machine learning approach based on a number of "neurons" (sometimes called units) each connected with many other neurons in the network, with the connections weighted, and with each neuron with some activation function combining the weighted connection values and a bias. An activation function takes some weighted inputs and a bias and determines the output of a particular neuron. Supervised neural networks also have a cost function, which defines the difference between the correct classification of an input and the actual classification of an input as given by the network (for a network designed for classification problems such as image recognition). As the ideal scenario is one in which all inputs are correctly classified, the aim is to minimise the cost function by adjusting the weights and biases of the neurons that make up the network. This process of adjusting the weights and biases to minimise the cost function (sometimes called a loss function) is called backpropagation. Neurons in a neural network are organised in "layers". Neurons in the same layer are not connected to other neurons in the same layer, but will be connected to all, or at least some of the neurons in other layers. For a "feedforward" neural network, the output values of neurons propagate through the network in one direction - from input layer to the output layer, potentially through some hidden layers (called hidden as they don't take inputs or form the eventual output of the network). Feedforward networks can be described using a directed acyclic graph - an example of such a network is shown in figure 2.7. Connections between neurons are shown with arrows, neurons are indicated with circles.

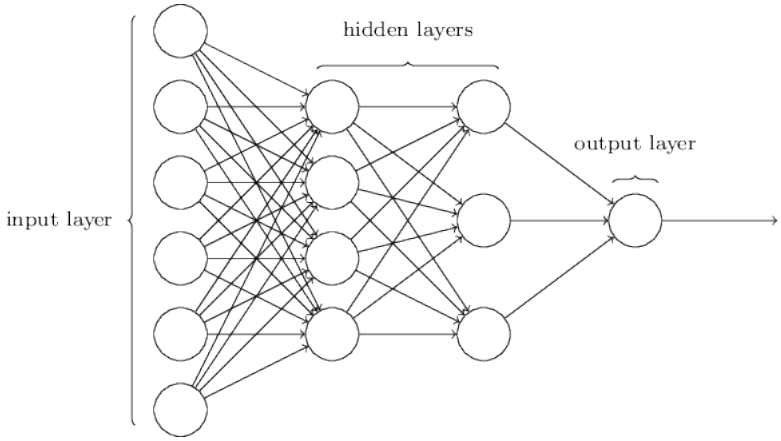


Figure 2.7: A feedforward neural network, with all layers fully connected. Weights and biases for each neuron not shown. Image taken from Neural Networks and Deep Learning[17]

Fully connected neural networks such as the one pictured have several issues when it comes to image recognition. Using 32x32x3 images (that is, images that are 32x32 resolution and have R, G and B colour values for each pixel), a neuron in the first hidden layer of such a network would have 3072 connections to the input neuron layer. If larger images were considered (such as 200x200x3 images), the number of connections per first hidden layer neuron increases to 120,000. The reason this is an issue is that each of those connections has a weight, which must be adjusted during backpropagation, leading to long training times.

There have been some attempts to limit the amount of free parameters in neural network models by limiting the number of weights that need to be calculated during backpropagation. This can be done by using neurons that take a receptive field of several neurons as an input and produce as an output a single value, thereby "squashing" down to a smaller "feature map". This concept was pioneered in Yann LeCun et al's paper[18], in which the authors presented the concept of a "convolutional network". This has since led to a whole field of neural network design, creating, along with other papers, the field of "deep learning".

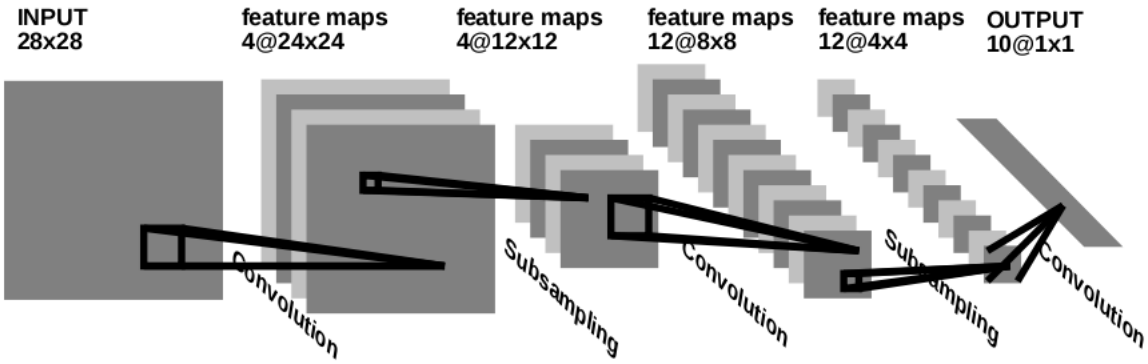


Figure 2.8: LeNet uses a series of convolutional layers, interspersed with subsampling layers (subsampling layers will be discussed later), ending with a fully connected output layer. Example receptive fields are shown outlined in black.

2.2.5 Convolutional Neural Networks

Convolutional Networks, or Convolutional Neural Networks (often abbreviated to CNNs), are now a major part of image recognition research.

The key part of a Convolutional Neural Network is the convolutional layer. The concept of convolution between a 2D filter and an image is described in appendix .1, but the mechanics of the layer have not. Each neuron in a convolutional layer takes as its input a certain amount of neurons in the previous layer, called the receptive field. The size of this field is defined by the filter, or kernel, size but is often relatively small, such as 3x3 or 5x5. After the output for a neuron has been calculated (as described in the earlier section on convolution, but with a shared bias added to the neuron), the next neuron's output is calculated, with the receptive field (that is the input "pixels" which the neuron is connected to) shifting by the stride of the filter. These results are then squashed, with each neuron's output from its receptive field becoming a single point in a feature map. Each layer produces multiple feature maps, each produced by the use of a different filter, with each being defined by the weights of connections between the neuron and its receptive field neurons (i.e. the previous layer). These feature maps function as shared weights for the layer's neurons. These feature maps become image features - in lower levels simple curves and lines, in later levels more complex patterns. The structure of this kind of multi-layer CNN can be seen in LeNet's architecture, shown in figure 2.8.

This idea of learning filters is particularly compelling for image recognition. As described earlier, previous image recognition approaches relied on applying hand-crafted filters and feature detectors to images, thus reducing the features of the image. CNN's similarly reduce features, and similarly design these feature detectors, but do so automatically. Rather than a human designing filters, the learning algorithm itself learns the filters that will detect the features to distinguish the classes. Not only is this more efficient than a human, but it is also more effective, as a CNN will learn filters that a human would never have devised.

Although LeNet was an important step in the field, there have been many CNN architectures since LeNet. Some consideration needs to be given to the best choice for recognising CAPTCHA images, and in general to the details of these architectures' designs.

AlexNet

One such architecture is AlexNet[19], an 8 layer CNN (five convolutional and three fully connected layers). One improvement AlexNet made over earlier networks was the use of Rectified

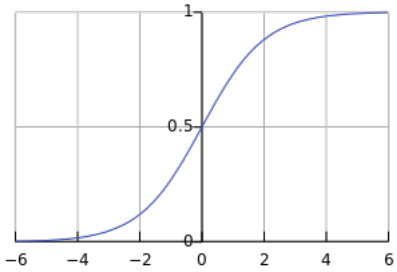


Figure 2.9: The curve of the logistic function - the gradients at either extreme end on the x axis are very flat, which in terms of a neural network means very small changes in output from the neurons, and thus a very slow learning rate if the weights start completely wrong.[20]

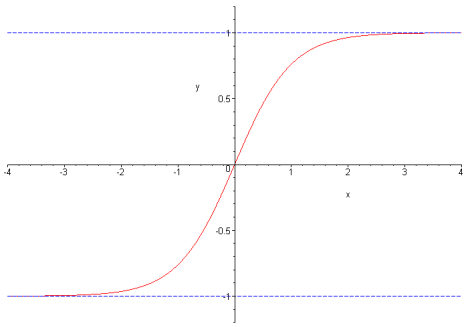


Figure 2.10: The curve of the tanh function, with similar issues as the logistic function.[21]

Linear Units (ReLU) rather than sigmoidal or hyberbolic tangent neurons[2]. The use of a logistic sigmoid or a hyberbolic tangent (tanh) activation function can lead to learning slowdown. It is easy to see this by looking at the comparative graphs of these functions.

The sigmoidal logistic activation function can be defined as:

$$\frac{1}{1+\exp\left(-\sum_j w_j x_j - b\right)} \tag{2.1}$$

Where j is a particular weighted input - so the denominator's exponent is taken to the negative sum of the products of all the inputs and their weights, minus the bias applied to the neuron. The logistic function's graph can be seen in figure 2.9. The tanh function is a scaling of the logistic function, as can be seen in figure 2.10. Little time is spent discussing the tanh function - as while better than the standard logistic function, it still suffers from the issues which are solved by the linear rectifier function.

Both sigmoidal functions can be shown to cause learning slowdown[22]. As LeCun explains, if the weights of the network are very far off the weight values that would lead to accurate classifications, sigmoid neurons become saturated. That is, the amount of improvement in accuracy and cost is much slower. Krizhevsky compares sigmoidal functions with the main alternative, that being a linear rectifier. A clear difference in the amount of time taken to reach the same accuracy can be seen in the following figure from that paper[19].

Rectified Linear Units (ReLU) have a more constant gradient, avoiding some of the flat gradients that occur from using sigmoidal neurons, and thus avoiding the learning slowdown when the weights are considerably off the optimum values.[23]

[2]Some literature uses the term "unit" rather than "neuron". This is down to a debate over how biologically inspired neural networks are, and thus whether calling the building block of a neural network a neuron is useful. For the purposes of this, the terms are sometimes used interchangeably.

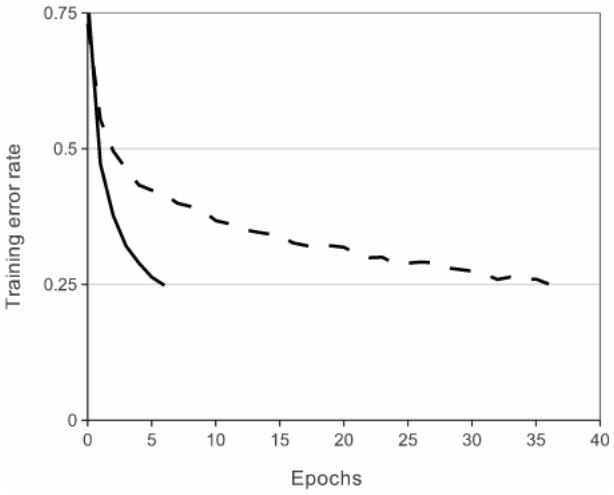


Figure 2.11: Error rate vs epoch number for ReLU (solid line) vs Tanh (dashed line). An epoch is a forward pass through the network and a backward pass (i.e. backpropagation).[19]

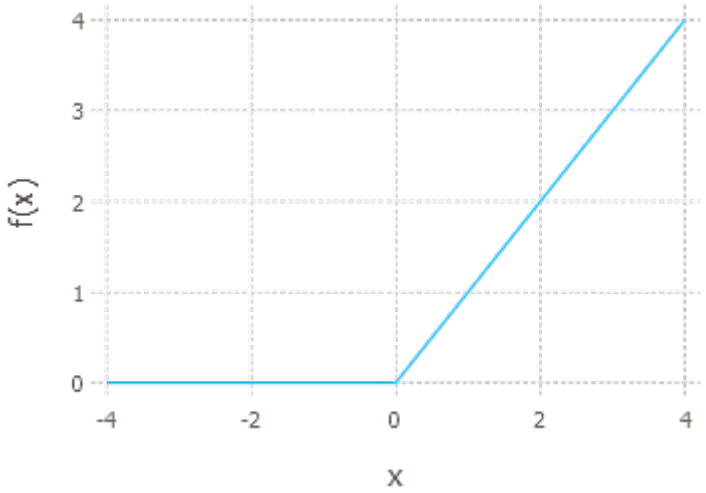


Figure 2.12: The gradient of the ReLU function is constant after an initial change, meaning the change in output values is constant for a larger amount of weights and inputs (the inputs to this activation function).[24]

Subsampling Both AlexNet and LeNet proposed the idea of using subsampling layers, layers which would take an input and downscale it using some function. AlexNet specifically uses a technique called Max Pooling, which for a given size (say 2x2) takes the maximum value from a portion of the input of that size and outputs the maximum value. This has the effect of picking the values from the neurons that activated the most (i.e. that have the largest activation values). This is a useful operation between layers, as it reduces the amount of values going into the next layer, and as such reduces the size needed for later layers, and thus, the number of weights. As well as reducing the amount of computation needed to train the network, Max Pooling also reduces overfitting, by removing small variances between similar input images (for example, two images where one is a translation of the other) - only core features (those that activated the most in a feature map) are retained after Max Pooling. Another kind of subsampling often used by neural network architectures is Average Pooling, which takes the averages of the feature maps from a convolutional layer and outputs these averages as a vector to the next layer. This has similar benefits to Max Pooling. The difference between these methods in practice depends on the dataset. According to Boureau et al's paper [25] Max Pooling tends to be better performing when the features are sparse, i.e. a large variety of image features with each of those features not present in very many images. As Boureau goes on to conclude, the correct solution may involve the use of both methods - which is common in many neural network architectures.

Dropout Another technique AlexNet uses to prevent overfitting is dropout. This involves setting the output of a random selection of hidden layer neurons to zero. That is, each neuron has a 0.5 probability of having its output set to zero. These dropped out neurons do not contribute to the forward pass and are not adjusted during backpropagation. The use of dropout allows the network to act like several different networks averaged together, without the cost of such an approach. Dropout has a considerable impact on error rate, Hinton's paper[27] on this showed a 6% drop in error rate after using dropout (compared to the same network without dropout), and a 5% drop in error rate compared with another approach that averaged the results of 5 separate networks (Hinton's approach only used a single network).

Early Stopping and Weight Decay Other approaches to prevent overfitting not used by AlexNet include early stopping and weight decay. The principle behind early stopping is to stop training the network before it fully converges but once most of the learning has been done. This is often implemented by stopping the training program once the loss on the validation set stops improving for a few epochs (which suggests the neurons are becoming saturated). This idea and it's benefits are discussed in more detail in Morgan and Bourlard's paper on the topic[29]. Their results showed that once the network's got past the point where the validation loss stopped improving, the accuracy of the network on test data got worse. Weight decay is a different approach that attempts to deal with the same problem. The concept of weight decay is to subtract a value from the overall weight change, so that if a particular weight is not being strengthened by being increased, it slowly decays. The size of this decay factor is discussed later, but there is no easy method besides trial and error of estimating it, as discussed

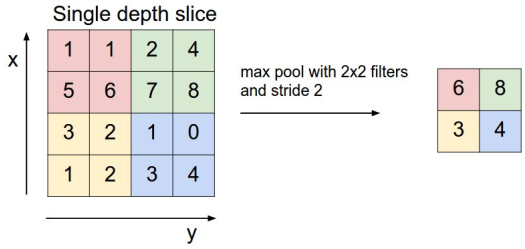
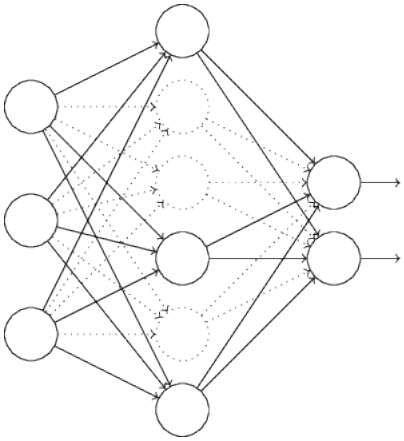
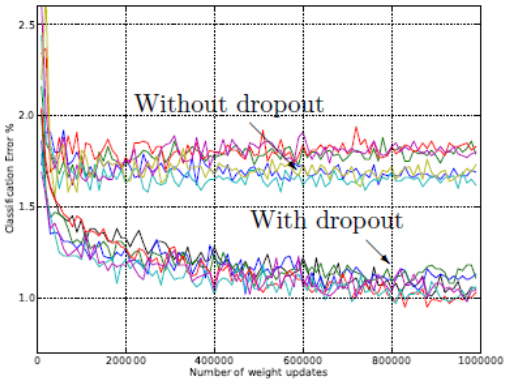


Figure 2.13: Diagram showing the maxpooling operation applied to an example input matrix.[26]



(a) Diagram showing the effect of dropout, with dropped out neurons indicated with dashed lines.[17]



(b) Accuracy vs number of weight updates required in training, shown with and without dropout.[28]

in Rognvaldsson's paper on the topic[30].

VGGNet

A similar approach to AlexNet is VGGNet[31], another convolutional neural network. VGGNet is similar in many respects to AlexNet, but it has many more layers (19 in the largest configuration, as compared to AlexNet's 7). This is mostly down to the use of stacks of convolutional layers, using smaller convolutional filter sizes (3x3 rather than AlexNet's 11x11). One advantage of using stacks of smaller layers like this is the number of weights, which is lower overall, while the performance for the whole network is better (24.7% error rate vs AlexNet's 40.7% error rate for the same task)[31]. VGGNet also uses a new approach for the output layer of its stacks of fully connected layers - Softmax. Rather than using a standard ReLU or Sigmoidal activation function for the output layer, a Softmax activation function is used, as the range of output values corresponds to a probability distribution (i.e. is between 0 and 1, all outputs will sum to 1) and thus can be used as the probability of each class for a particular input.

Inception

Building on the concept that deeper networks are more generalisable, and thus more accurate on unseen data, the Inception CNN uses networks in networks, making for a very deep CNN. The concept of Network in Network was first outlined by Lin et al[32], where fully connected neural networks were used as layers inside the convolutional neural network. The Inception network builds on that concept, using "Inception modules" made up of several convolutional and pooling layers (each applied to the same input), with the feature maps of those layers concatenated together as the output. An example of the inception module used by Szegedy et al[33] in their GoogLeNet architecture can be seen in figure 2.15.

Although deeper neural networks seem to perform better than a more shallow network, simply adding more layers is not an optimal solution. In fact extra layers can increase error rates dramatically, as seen in figure 2.16.

ResNet

One approach that attempts to deal with this problem is ResNet[34]. ResNet uses "Residual learning", that is, rather than simply feeding an input through the layers of the network, every few layers they add the original input to the output of those layers. Formally speaking, if there is some mapping of a class to an input denoted by $H(X)$, a normal CNN would estimate $H(X)$ by computing some $F(X)$ through the use of the layers that have already been described. What

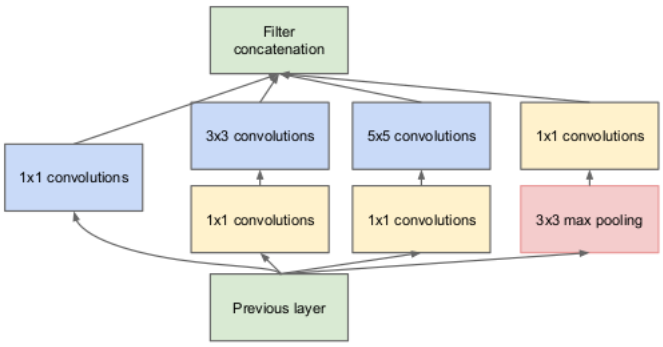


Figure 2.15: An example Inception module using 1x1, 3x3 and 5x5 convolutional layers, as well as a max-pooling layer.[33]

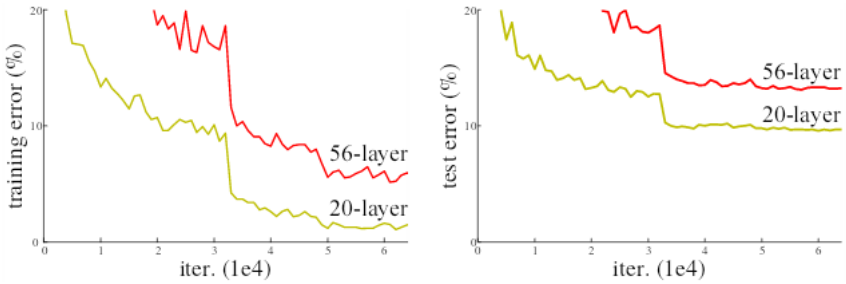


Figure 2.16: Training and test error on an image recognition task with 20 layer and 56 layer neural networks.[34]

ResNet does is estimate $F(X) + X$, by using "shortcut connections" to add the input to some stack of layers to the output of those layers. He et al suggest that $F(X)+X$ is easier to optimise than just $F(X)$, and the results seem to back up this hypothesis, with even the smallest ResNet implementation (with 34 layers) having a top-5 error rate (that is an error rate for the 5 labels which the network is most confident in) 1.39% better than GoogLeNet. It should be noted however, that later versions of the Inception architecture also incorporate this idea of residual connections.

Although deep architectures are effective, they have problems in terms of training time and hardware cost. Inception and ResNet both have large numbers of layers and also large numbers of neurons in their final layers. This costs massive amounts of video memory (over 8GB for 110x110 size images) which requires specialised high end hardware. Without this hardware, networks either train slowly (taking up to a week to finish learning) or cannot start training as the hardware simply cannot handle the amount of data.

Xception

Extending the concept of Inception's network in network modules, Chollet in his paper on the Xception architecture[35] suggests the use of modules that perform several convolutions in sequence (like in Inception), which he calls depthwise (as the convolutional layers are applied to every channel of the input) separable (as there are several convolutions in sequence) convolutions. As previously described, and shown in figure 2.15, Inception modules use several 1x1 convolutions followed by 3x3 or 5x5 convolutions, before combining the outputs. Chollet proposes several 3x3 convolutions (one for each channel, so 3 in the case of RGB images), followed by a single 1x1 convolution on the combination of those convolutions. This is a similar architecture to Inception, but while Inception requires massively deep networks to achieve it's

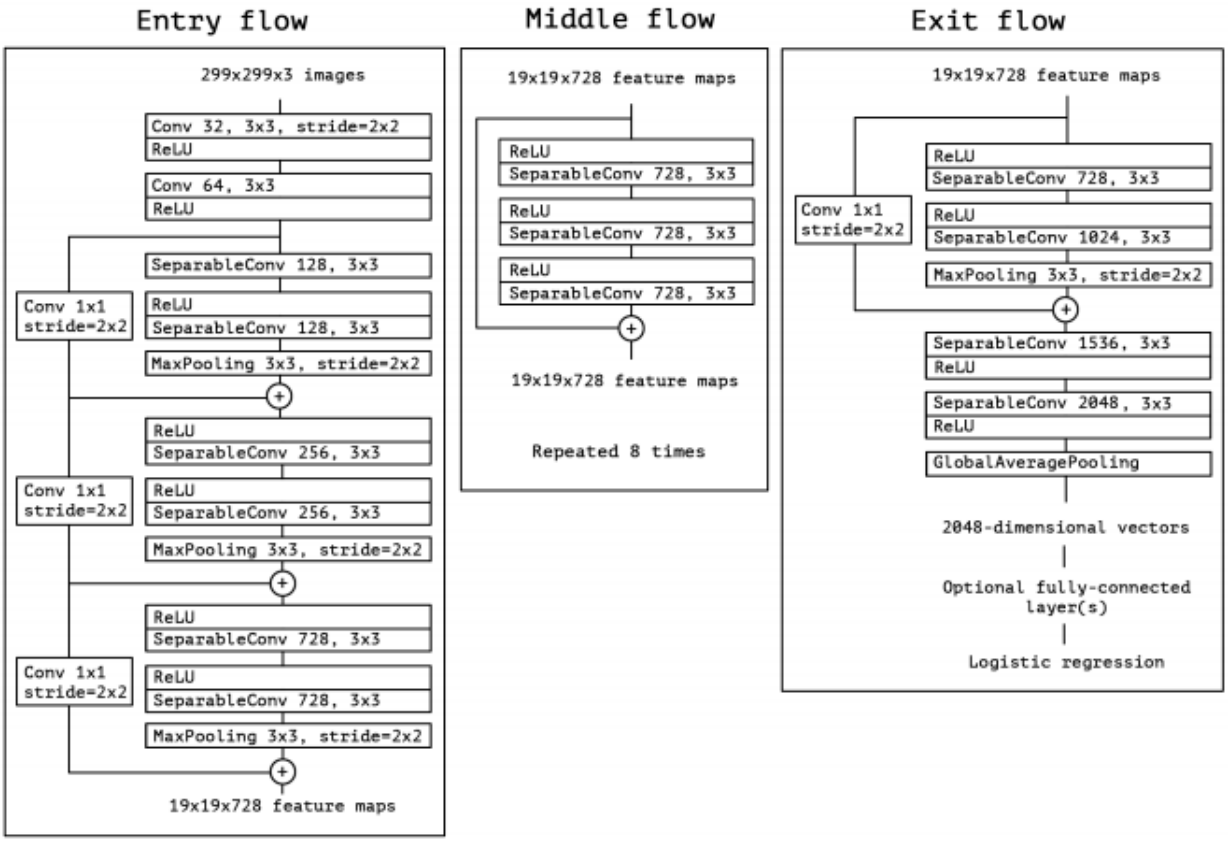


Figure 2.17: The Xception architecture. The depthwise separable convolutions can be seen easily in the first section, where there are three sets of SeparableConv layers applied to the input sequentially, each having a 1x1 convolution applied to it. The diagram is meant to be read section by section top to bottom.

performance, Xception's design, using these depthwise separable convolution layers, requires only a similar amount of layers to VGGNet, and also requires no fully-connected layers (besides the final softmax layer) at the end of the network. Xception also uses the residual connections seen in ResNet and later iterations of Inception, which theoretically improves the performance due to the reasons already discussed in section 2.2.5. A full description of this architecture can be seen in figure 2.17.

2.2.6 Conclusion

Recent trends in computing power, availability of data, and the improvement of algorithms for utilising those things suggest that deep learning methods, and particularly Convolutional Neural Networks, are the correct solution for this image recognition problem. While using feature selection methods such as GIST or HoG, combined with a simple classifier like SVM seems like a compelling method, it fails to perform as well as a CNN approach on large sets of unseen data. The reason for this is at least partly due to the flaws of feature selection/reduction algorithms. Any specially designed method (such as an edge detection filter, or a colour histogram) will inevitably discard important features. A better approach than hand crafted feature detectors is to try many different filters, keeping the best, and applying subsampling inbetween as a feature reducer. This is essentially how a convolutional network works. This leads to the question, "which of the CNN architectures are suitable for the task?" One thing to be taken into consideration is the error rates of the architectures on existing datasets. Some error rate

statistics on ImageNet are shown in table 2.1.

Network	Top-1 Error (%)	Top-5 Error (%)
AlexNet	40.7	18.2
VGG-16	28.5	9.9
VGG-19	27.3	9.0
Inception v3	21.8	5.9
ResNet (152 layers)	23	6.7
Xception	21	5.5

Table 2.1: Error rates of several CNN architectures on ImageNet. Taken from Xception's[35] paper and AlexNet's[19]

These results suggest that Xception is the best performing network, but there is only a very small difference between that architecture and Inception. These results are also on ImageNet, a completely different dataset to the one being used by this project.

Performance is another factor to consider here. An architecture like ResNet's 152 layer network is very deep, and thus very resource heavy. This project has restraints on the amount of VRAM available, unlike a company like Microsoft or Google (for whom training a network using tens of gigabytes of VRAM for months is feasible), and also on the amount of time available for waiting for a network to converge. With the graphics card available, which has 2GB of VRAM, only VGG and Xception had implementations available which could work with the dataset being used and the VRAM available. To test which of these architectures perform better, a small test was devised, using a fraction of the dataset to train, and comparing validation accuracy and loss on a fraction of the validation set. Everything but the network architecture was kept constant (learning and decay rates, and backpropagation method).

Architecture	Top-1 Accuracy (%)	Top-5 Error (%)	Loss
VGG19	18.14	33.43	3.4
Xception	34.84	62.47	2.5

Table 2.2: Comparing the VGG 19 layer architecture with Xception for a subset of the dataset.

These results in table 2.2 suggest that Xception performs better with the dataset at hand, which fits the other accuracy statistics from ImageNet. As such, this project will use the Xception architecture.

2.3 Semantic Similarity

Once the classifier has labelled the CAPTCHA images, it is perfectly plausible that some of the labels are the same meaning as the CAPTCHA query, but are different words. For example, several of the images could be labelled as "general store" or "bakery", and the CAPTCHA query could be "store fronts" - these are similar concepts, but completely different phrases. In order to cope with this, the classification labels and the query could be assessed for semantic similarity, that is how close the phrases are in meaning to each other.

2.3.1 Word similarity

A simple method would be to check whether a label is contained within the CAPTCHA query. Phrases are split into lists, so every element in the label list is checked for membership of the query list. Words are also singularised to avoid plurals and singulars being considered different. For instance, a label like "shop front" has the word "front" which is contained within "store

front" which could be a CAPTCHA query. This does not strictly measure semantic similarity, but often conflates terms which are similar in meaning. While "office building" and "apartment building" have different meanings, they are not completely dissimilar. However, this method is somewhat flawed as labels can easily be over-conflated, such as "street" and "street sign" which are importantly different concepts. That said, the advantage of such a method is it is efficient, simple, and requires no text corpora to work out similarity.

2.3.2 WordNet

Another potential approach involves using databases of English words organised into hierarchical trees based on groupings of words that are roughly synonymous (called synsets). Resnik[36] suggests measuring the length of the path in the tree between one phrase and another - further apart words are likely less similar. This works well for simple hierarchical structures with few words (such as limited specialist vocabularies of several thousand words), but does not scale well to a more exhaustive database, such as WordNet[37]. Possible improvements to the simple method include using the depth of the word in the tree, and using the information content of the word or phrase, calculated based on a corpus of English text - both approaches are used by Li in their paper on using WordNet for semantic similarity.[38] One issue with using WordNet is that it is not an exhaustive tree of the whole English language, and as such many of the labels are not contained within it. Also, WordNet only considers the relations between individual words, not whole phrases. Using it requires judging the similarity of all of the words in a label with all of the words in a CAPTCHA query. This method though is not perfect, as a phrase like "shop indoor" has similar words with "store front" - i.e. shop and store are similar - but the meanings are importantly different. One phrase says that the image is of the indoor of a shop, and the other implies the image is of the outside. These are completely different scenes, but with some similar words.

2.3.3 Other ontology based approaches.

The issue of WordNet not containing all of the labels could be solved by simply using a more comprehensive ontology. Wikidata[39] is such an ontology, as it uses data from Wikipedia. This means that it is human created and maintained, unlike any corpus based approach, and also is much larger than WordNet. Additionally, it contains short phrases alongside words, so this avoids the downsides of losing phrasal context. However, a similarity method using this ontology would have to be devised that could efficiently deal with the gigantic tree structure, in order for it to be used in practice.

2.3.4 word2vec

Another potential approach uses Mikolov's Skip-gram model[40], a neural network based approach that learns which words are similar to each other based on converting words to vectors in a vector space, positioned based on similarity. This approach requires corpora that contain all of the labels, in context with lots of words similar to those labels. Again this method is word based, so has the issues with that described in the previous section. But, it does use a corpus to get the similarity measures, so this avoids the issue of WordNet's limited vocabulary, but does require a corpus with all of the labels in it. The issue with using such a method that requires a corpus is in finding a corpus which contains all the labels. Despite the size of available corpora, several corpora were tried and none had all of the labels. A larger corpus like the British National Corpus would probably contain all of the labels, but using such a giant corpus would raise questions of efficiency. As word2vec works with individual words rather than phrases, it suffers from the same issues around discarding phrasal context that WordNet methods do.

2.3.5 Latent Semantic Indexing

A better method would preserve phrasal context, like Latent Semantic Indexing. Dumais et al in their paper on the topic[41] propose a method which builds a continuous vector space from a corpus of phrases, and then this space can be used with a query phrase to find the similarity between any phrases, or individual terms in the space. Another advantage of the method is the space can be added to after initial creation, meaning that the whole space need not be recreated if extra labels needed to be added. Of course, LSI still requires a corpus, and as such has the problems already outlined in the discussion of word2vec.

2.3.6 Conclusion

Semantic similarity methods alleviate the downsides of using a dataset that, by necessity, is not identical to the dataset used by reCAPTCHA. reCAPTCHA presents CAPTCHAs with different kinds of images, and different labels to that which the network has been trained on, and so the network may label images correctly but not with the exact labels looked for by reCAPTCHA. The issue with these methods is over conflation of labels. An image of a bakery could be an indoor image, and so not qualify as a store front (unlike an outdoor image, which should be conflated). Perhaps if the labels were instead description phrases, even sentences, rather than short one or two word labels this would be alleviated. However this poses other issues as such sentences would have to be generated from the images or more image training data would have to be used. A good semantic similarity method is hard to design, requiring a good corpus (not only containing all the labels, but also containing them in lots of different contexts) and a good method. Moreover, a bad method will over-conflate labels leading to poorer performance. An ontology based method using Wikidata could be suitable, but further research would need to be done into how that ontology could be used efficiently. LSI seems like a suitable method for semantic similarity, but due to the issues outlined with corpora, the word similarity method has been used for this project.

Chapter 3

Analysis

As well as assessing existing approaches to the problems, which tools, languages, libraries and datasets the project will use needs to be assessed.

3.1 Programming Language

3.1.1 Python

Python is a high level, interpreted, dynamic programming language. It is well suited to this task, as it has a plethora of libraries for machine learning as well as several libraries for browser automation. It is also easy to read and write, but where it gains in ease of use it sometimes loses in performance compared to compiled languages like C++ and Java.

3.1.2 MATLAB

MATLAB is a programming language and environment generally designed for numerical computing tasks. The main advantages of MATLAB over a language like Python are that lots of datasets provide .mat files (MATLAB's binary file format), or toolkits designed to work with MATLAB. This makes it easier to train a machine learning algorithm. The main issue with using MATLAB is the fact that it is proprietary and is also expensive (unlike Python which is open source and free). This would cause problems for development on personal machines.

3.1.3 R

R is a programming language and environment mostly designed for statistical computing and data mining tasks. This would be suited for the project, as machine learning approaches tend to be based around statistical methods. R has an advantage in that some implementations of the language have functions written in C under the hood, making for faster calculations on a large scale. However, many Python machine learning libraries, like NumPy, also have functions written in a language like C or Fortran, so the speed difference may not be noticeable. R also makes heavier usage of system RAM, which can be a problem for large datasets.

3.1.4 Conclusion

Python is the appropriate language for this project, due to the amount of libraries available for the tasks needed to be completed. Additionally, Python has already been used in similar projects, so it has shown its suitability to the task. Although the performance of Python's core language may be worse than some compiled languages, as mentioned many libraries have optimised their performance somewhat so this should not be a major issue.

3.2 CAPTCHA Input

As the correct answers for each CAPTCHA are not exposed in any way, even to the site owner, the only way to evaluate the effectiveness of the CAPTCHA cracker is to "click" images in the CAPTCHA, and "click" verify, like a human user would. There are several libraries for this in Python, as this is a common task in software testing.

3.2.1 Browser Automation Software

There are several approaches to browser automation (that is, interacting with a browser via a program, so as to interact with a page e.g. visiting a page and clicking a button) in Python.

Splinter

One such approach is through using Splinter[42], which is a library that acts as an interface for a web driver such as Selenium, Firefox or a headless web driver like PhantomJS (headless meaning it runs in the command line rather than opening a graphical browser window). Although such web drivers have libraries to interface with them directly, Splinter provides several functions to easily find content on a page (such as the checkbox in reCAPTCHA), click on page elements, and navigate to iframes, amongst other things. Splinter is available for Python 3, and generally allows and encourages pythonic code.

requests and BeautifulSoup

Requests[43] is a library that allows for sending HTTP requests, like GET and POST, easily. This could be used to send GET requests to fetch the CAPTCHA page, or the CAPTCHA images, or other requests to verify the chosen CAPTCHA solution with the reCAPTCHA API.

BeautifulSoup[44] is a library that has several functions for parsing HTML. With requests used to download the pages and submit responses, BeautifulSoup could be used to find the relevant parts of the page (such as images, checkboxes, the question text in the CAPTCHA). This combination could fulfil much of the functionality of Splinter. However, it is impractical due to the real time nature of the task. CAPTCHAs need to be solved in a certain amount of time, or the CAPTCHA window will minimise. Such a long winded download, parse, send request method would likely not be fast enough. Additionally, using requests would require further research into the requests made by reCAPTCHA when a user clicks "Verify", whereas with Splinter the button can just be clicked.

Selenium

Selenium[45] is a web driver (an application that interacts with a web browser, often by opening a normal web browser window and sending commands to it) which also has its own Python library with functions to use it directly. Although Selenium has the same functionality as Splinter (as previously mentioned, Splinter can use Selenium as its web driver), Splinter is further abstracted from the details of the web driver, making for easier to read code.

Conclusion

Splinter is the best choice for this project, as it is the simplest to use due to its level of abstraction from the details of the web driver being used, or the exact HTTP requests needed to be made. An approach using BeautifulSoup could be more appropriate for a case where detailed parsing of the page needed to be made, or where there were no time constraints. Using Selenium makes little sense, as Splinter can use it as its driver, without needing to interact with it directly.

3.2.2 Datasets

A key problem with any machine learning approach is in choosing the training, validation and testing dataset. There are several datasets of labelled images available on the internet for these purposes. It is not the case that only one of these datasets can be used, or that the whole of any of these datasets need to be used. A more effective solution would train on a subset of several of these datasets, selected based on relevance to the subjects of the images observed in reCAPTCHAs. That said, there is a danger of creating a too specific dataset as that can lead to overfitting.

ImageNet

ImageNet[46] is a dataset of over 14.7 million labelled images, with 22,000 classes. The training time for training on such a large dataset must be considered, but at the same time a large variety of images can reduce the effect of overfitting and thus improve accuracy on an unseen test set.

LabelMe

LabelMe[47] is another image dataset, using humans to label parts of each image. Although LabelMe's images are similar to ImageNet, LabelMe has more information about each image than necessary for this task, in that each object's location in the image is also indicated. That said, such information could be useful in that each image would be labelled with several labels, potentially improving recognition of images that are less clear as the network would be trained on a wider range of images of an object (in terms of clarity and size). That said, it could also lead to more misclassification, as an image with a large object and a very small object in it would be labelled as both objects, possibly leading to misclassifying images of the large object as the small object.

Caltech256

Caltech256[48] is an image dataset with 256 different categories of image in it. This image dataset is quite good for general everyday objects, animals and places but it lacks in photos of outdoor scenes which are more prevalent in ImageNet and LabelMe.

CIFAR100

CIFAR100[49][50] is an image dataset with 100 different categories of image in it. Similar to Caltech256 in content, and so also has the limitations of that dataset in terms of its lack of photos of real world locations.

SVHN - Street View House Numbers

SVHN[51] is a dataset with over 45,000 images of house numbers, taken from Street View images. Although this is a very specific dataset, there are CAPTCHAs in reCAPTCHA that specifically ask to select the street numbers, so having a set of street number images in the datasets will be useful.

GTSR - German Traffic Sign Recognition

GTSR[52] is a dataset with over 50,000 images and over 40 classes of street signs in Germany. Each of the street sign images has several versions at different light levels, which is not ideal as most reCAPTCHA street signs are photos taken during the day making light level less relevant. All the photos seem to be taken from a similar angle, which is also an issue, as many real world

scenes contain signs at many different angles. Although there are issues with this dataset, it is the only publicly available dataset of traffic sign photos.

Places2

Places2[53] is a dataset of photographs of outdoor and indoor scenes, with 365 different classes. The standard dataset is 1.6 million images. This is a similar dataset to ImageNet, but has complete label files available which makes training easier.

Conclusion

A good dataset for this problem needs a combination of datasets to ensure the dataset is generalisable. A subset of a general scene dataset like Places2 is sufficient to recognise many different real world scenes. The addition of SVHN improves performance on street number CAPTCHAs, as the addition of GTSR does for street signs.

3.2.3 Libraries for Machine Learning

There has been an explosion of deep learning and general machine learning libraries in recent years. While many of these libraries are similar, there are some key differences. This section will focus on the libraries that can be used with Python.

Tensorflow

Google's library for deep learning and neural networks, written in C++ and Python with interfaces in both languages. Tensorflow[54] provides a lot of functions for doing common operations in neural networks, such as gradient descent, or making convolutional layers.

TensorLayer

TensorLayer[55] is built on top of Tensorflow, providing a higher level interface to make constructing a relatively standard neural network easier. The advantage of using this over Tensorflow itself is in convenience, as it does more of the work for you. Of course, this comes with some lost flexibility and potentially lost efficiency.

Keras

Keras[56] is very similar to TensorLayer in that it is a library built on top of a lower level library (either Tensorflow or Theano). The main difference between Keras and TensorLayer is that Keras is more established, with more contributors and having been around for longer.

Theano

Similar in many ways to Tensorflow, but there are some efficiency issues as Theano[57] can do a lot of background tasks that take time (such as compiling code into machine code).

ScikitLearn/Numpy/Scipy

[58][59][60] While a combination of these libraries would be capable of making a convolutional neural network, none of these libraries are designed just for neural networks. This would mean developing a neural network from scratch basically, which while feasible would likely result in a less efficient, less performant network. It would also take valuable time which could be spent focussing on the design of the network, rather than the fine details.

Caffe

Caffe[61] is similar to TensorLayer and Keras in terms of how abstracted it is. However, its Python interface, pycaffe, is not as fully fledged and requires you to do a fair amount of defining the network outside of Python code. Unlike TensorLayer and Keras however, it is built in C++, which could make it more efficient than either of those libraries (although not necessarily better than a library like Tensorflow, which can make deployment versions of a network in C++).

Conclusion

Keras using Tensorflow is the best choice for the project, as it is high level, allowing for time to be spent focussing on design choices rather than implementation detail. It is also well established, with a large team of contributors. Equally important is Keras's well made Python interface, which will allow for the neural network to interface with the rest of the program easily.

3.2.4 Requirements

An important part of assessing how this project will be carried out is explicitly outlining the requirements for the finished system - that is, what will and will not be done, as well as what should or could be done on top of critical requirements. The cost of each requirement (i.e. how long it will take to develop, relative to the other requirements, is estimated as one of 1,2,4,8,16 - that is, a requirement with a cost of 16 will be double the work to develop a requirement of cost 8).

	Requirement	Estimated Cost
1	It must tick the checkbox, bringing up the image captcha challenge.	1
2	It must download the image/images and segment them correctly (i.e. a single full image that is meant to be 9 images should be split into 9 images).	4
3	It must read the challenge text (such as "Select all the drinks.") and store it so it can be used for identifying which images are closest to the object specified.	2
4	It must classify the images provided to it from the program interacting with the CAPTCHA, giving each image a label (or several labels with accuracy values).	16
5	It must interact with the CAPTCHA, attempting to solve the challenge using the labels of the images provided by the other parts of the system.	2
6	It should judge labels from the classifier for semantic similarity to the challenge text object (e.g. store front is similar to shop in meaning, so an image labelled as a shop should be considered a good candidate for a CAPTCHA asking for store fronts) and use those similarity values to judge suitable solutions for the CAPTCHA challenge.	8
7	It should determine (from classifier's expected accuracy of label values) when it is unlikely to get the correct answer to a CAPTCHA, and click reload if that is the case.	4
8	It could attempt to solve "click and drag" image CAPTCHAs by finding the part of the image that an object resides and interacting with the CAPTCHA accordingly.	8

3.2.5 Evaluation

Evaluating the performance of the program can be done through several metrics. Firstly, the classifier itself will output accuracy statistics, and the amount of time taken to achieve that accuracy score. Secondly, the semantic similarity measure will have similar statistics to the classifier. Most importantly, the overall program can be judged by how many CAPTCHAs it solves correctly out of all the CAPTCHAs attempted. There is no way of discovering that an individual CAPTCHA was solved correctly, only that a chain of CAPTCHAs were solved correctly. reCAPTCHA does not require perfect accuracy, but it does require several CAPTCHAs to be solved in an approximately correct fashion, consecutively, as can be seen from the example in figure 3.1 which seems correct but was not counted as such. This makes evaluating the effectiveness of the whole system very difficult. Certain hints are given to indicate incorrect images being chosen, such as red text below the images saying "Please select all the remaining images." However this does not always occur. Individual CAPTCHAs can be examined by hand to see the effectiveness of the whole system, but this makes it time consuming to provide accurate statistics. The only statistic that can be gathered automatically of the whole system is the number of times a CAPTCHA correct tick is detected in the browser. Although these problems make improving the system more difficult, in terms of effectively breaking the reCAPTCHA system a score based on how many times any of the tests is passed and a tick is seen (and thus be able to submit a form that the CAPTCHA is attached to) is accurate and useful.

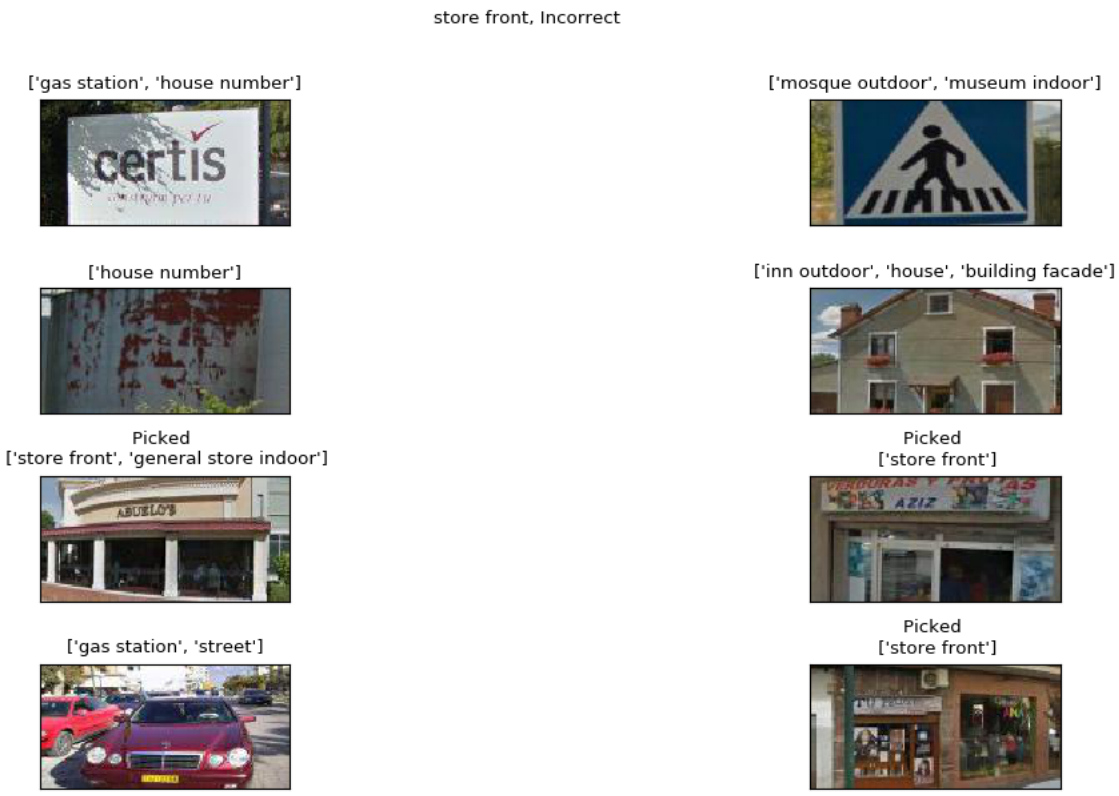


Figure 3.1: This is an example of a CAPTCHA which was actually correct, but was perhaps succeeded by an incorrect CAPTCHA, leading to no tickmark.

Chapter 4

Design

The program's design will be discussed in two main parts. One part interacts with the web page with the CAPTCHA challenge, and the other part (the neural network) gets labels for the images given to it.

There are two main parts to the CAPTCHA solving system. The first part needs to interact with the CAPTCHA - downloading the images, retrieving the CAPTCHA query and getting any other useful information that will assist in solving the CAPTCHA. The second part is the neural network responsible for returning class labels for the images in the CAPTCHA. Both of these subsystems are complex and as such will be discussed first from a high level in this section and then from a more detailed view in the Implementation section following.

4.1 CAPTCHA Interaction

A program was designed using Splinter to interact with the reCAPTCHA. From this point, some terminology will be used to describe the program. Image checkboxes (or just checkboxes) are the individual clickable images in the reCAPTCHA image challenge. The initial checkbox is the first "I am not a robot" checkbox. The CAPTCHA query is the piece of text specifying what the user should click on, e.g. "street signs". The flow of the program is described from a high level in figure 4.1, and in a more detailed breakdown below.

1. The initial checkbox is clicked on. The program checks whether it passed the CAPTCHA at this stage, if so it increments a correct counter, and the number of guesses counter.

2. All the information about the image challenge is gathered. This includes the CAPTCHA query, the URLs for the images and the number of rows and columns in the CAPTCHA grid. This step also involves saving the images from the CAPTCHA to the disk so they can be inputted to the neural network. If the image URLs are the same as the previous CAPTCHA's URLs, an exception is thrown so that a new CAPTCHA can be acquired. This prevents the program becoming stuck in an infinite loop, trying to solve the same CAPTCHA with the same, wrong, solution.

3. The saved images are then preprocessed. That is, they are resized to 93x93 (many of the CAPTCHA images are already this size), changed to RGB colour if they are black and white, and normalised using the same process as used on the training images.

4. These images are then passed to the neural network, which returns a list of classes for each image, if it can.

5. The predictions are checked to see whether any of them match the CAPTCHA query, using the similarity method described in 2.3. If so, the checkboxes associated with those predictions are clicked.

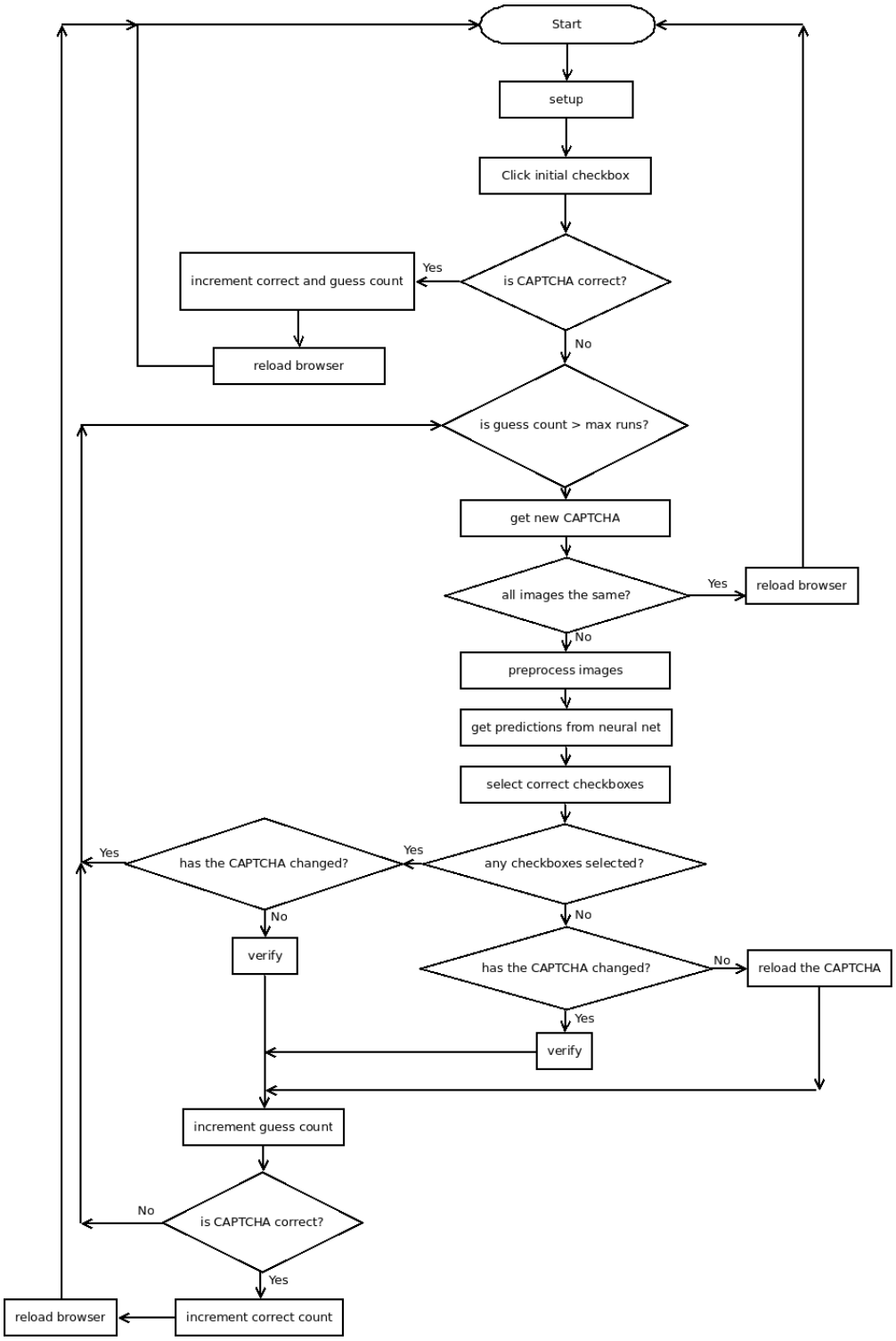


Figure 4.1: A flow diagram showing the main logic of the program.

6. If there are checkboxes matching the query, and if the CAPTCHA has changed, the program continues getting the new images, and making predictions for those. Otherwise if the CAPTCHA has not changed, it clicks the Verify button.

7. If none of the predictions match the query, then the program clicks the Reload button to get a new CAPTCHA, unless the CAPTCHA has just changed, in which case there is a possibility all the previous choices were correct and there are actually no more images matching the query, so the program clicks Verify. This and the previous step's logic is designed to handle the different kinds of CAPTCHA outlined in 6.7 - the complexity of the control flow is down to handling the diverse outcomes that result from clicking some CAPTCHA checkboxes.

8. In the case of either clicking verify or reload, the number of guesses counter is incremented.

9. After clicking either verify or reload, the program checks whether the CAPTCHA has passed (i.e. a tick has appeared in the initial checkbox). If so, the correct counter is incremented.

These steps continue in a loop, reloading the browser if the program crashes for some reason (such as an element on the page not being detected correctly).

The overall structure of the program can be seen in figure 4.2. The arrow direction indicates a "uses" or "has a" relationship, for example, CaptchaCracker uses the config module, uses the captcha_files module, and has a CaptchaElement. It was decided to use an Object Oriented structure as it was useful to separate the logical objects of the program (the CAPTCHA and its Checkboxes) from the interaction code, and the interaction code (e.g. "find this specific element with this CSS selector") from the high level behaviour (e.g. "click checkboxes and click verify").

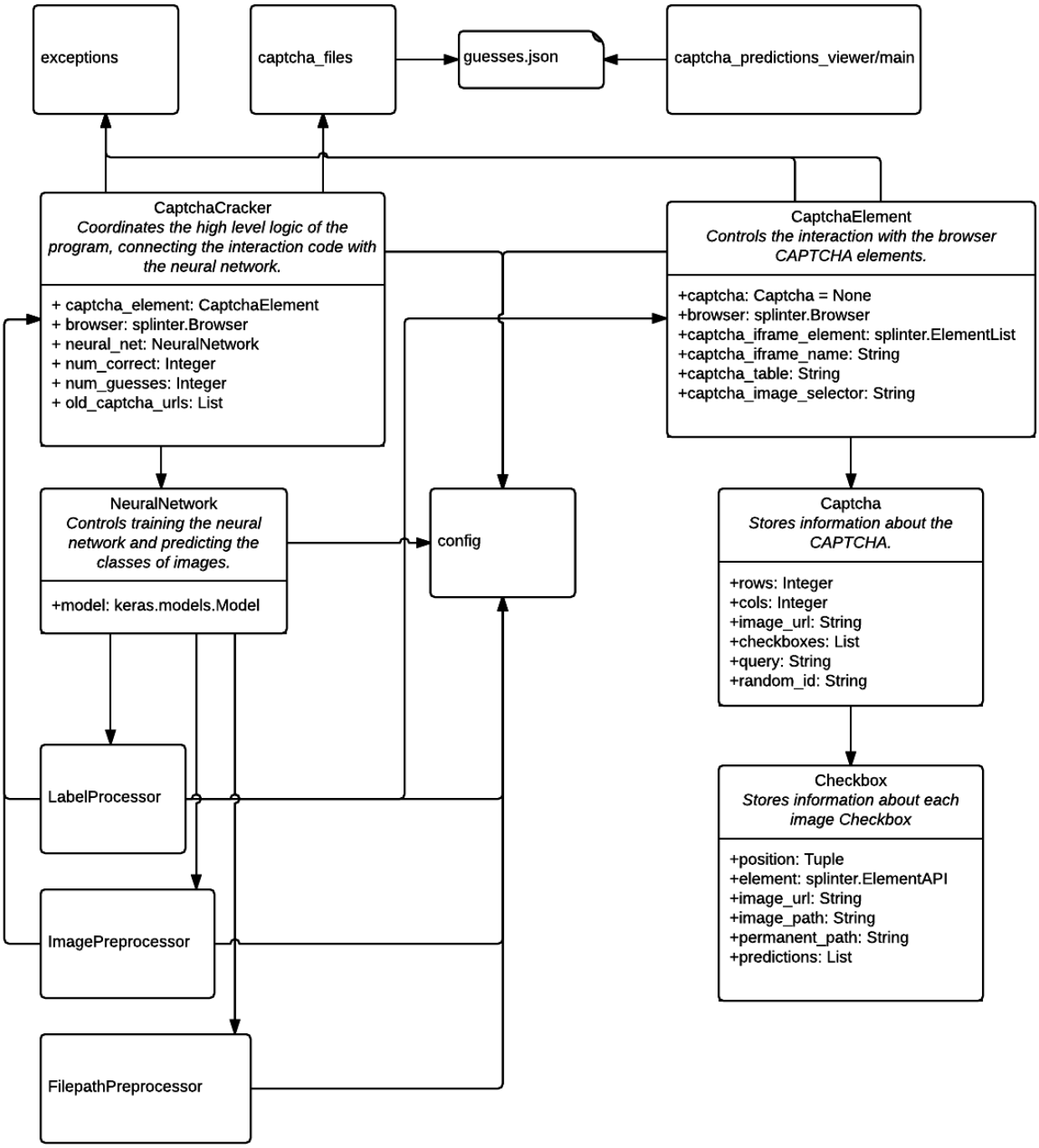


Figure 4.2: A diagram showing the structure of the program, its classes and modules. Classes have their attributes described, and are written in CaptainCapital style, while modules use snake_case. The functions and methods have been left out of the diagram for clarity. Similarly, only the guesses.json file is included (and not the labels or categories files, for example) because that illustrates the link between the CAPTCHA viewer and the rest of the program.

4.2 Neural Network

The Xception architecture is used for the neural network. The reasoning behind this choice of architecture has already been discussed in subsection 2.2.6. While the network architecture choice has already been discussed, there are several other factors to consider when designing a neural network approach, both for training and for actual usage.

4.2.1 Minibatch Training

With a large dataset, loading all of the image data into memory at once is very intensive. Additionally, running all of the images in the dataset through the network before updating the weights (a method usually called batch training, confusingly) could have disastrous effects (as the resultant weight update would be massive). To solve these problems, the image data is split into minibatches of 25 images. Each minibatch is preprocessed, and fed through the network, updating the weights after each minibatch. This leads to frequent, small weight updates so errors can be corrected by the network easier. It also means only the data for 25 images needs to be in memory at any one time. Smaller minibatch sizes are preferable, as weight updates are regular and small. Wilson and Martinez's paper[62] on the subject goes as far as suggesting that online training (updating the weights every sample - effectively a minibatch size of 1) is both faster to converge and more accurately assesses the gradient. While this is true in some cases, in practice the efficiency of online training is questionable. Wilson and Martinez argue that a model using this method may converge faster, but in practice small batch sizes introduce additional overhead costs and does not efficiently use the GPU (multiple cores are not needed as there is only one sample being considered at a time). For this problem, minibatch training was found to be both efficient and effective.

4.2.2 Loss function

Categorical Crossentropy is used for the loss function. The loss function, as previously discussed, serves to judge "how wrong" a particular output for a given input is. There are several such functions that have been devised and could be used for this problem, including classification error and mean squared error. Classification error is already used to analyse how effective a network is after training, as it is calculated by dividing the number of correct predictions by the total number of predictions. This is a useful measure as it shows how useful the network is at the task it was designed for. However, it is not so useful as a loss function, as the loss function is used for updating the weights, and classification error gives no information about how wrong a prediction was, just that it was wrong. A better method than classification error is mean squared error. Mean squared error is calculated by getting the mean of the squares of the predictions minus the correct outputs. As the output of the network is a vector of probabilities, one for each class, and the labels are one-hot (i.e. a vector of length equal to the number of classes, with a 1 a the index equal to the class label, and 0s for all other values) this can be easily and efficiently calculated, especially on a GPU. The problem with mean squared error is that incorrect and correct predictions, while distinguishable, are not very distinct. Some good of this issue are given in this article [63]. Crossentropy error, then, is a better loss function as it gives the same information that mean squared error does (a judgement of how wrong a prediction is) but emphasises wrong predictions less. Crossentropy error is defined as:

$$C = -\frac{1}{n}\sum_{x} y \ln a + (1-y)\ln(1-a)$$

Where n is the number of training samples the crossentropy error is being calculated for, x is those training samples, a is the actual output from the network and y is the correct output. The article previously mentioned [63] also gives examples of crossentropy, in comparison to mean

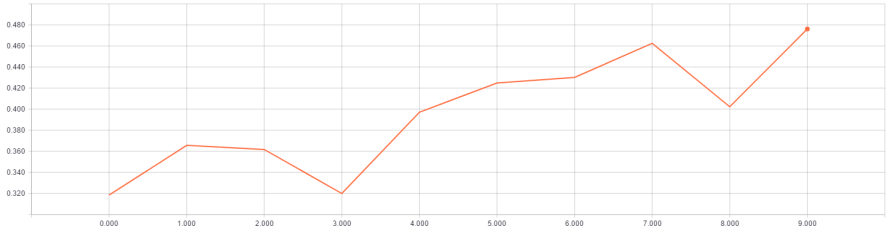


Figure 4.3: Graph showing the top-1 validation accuracy over epochs using the Adam backpropagation optimiser.

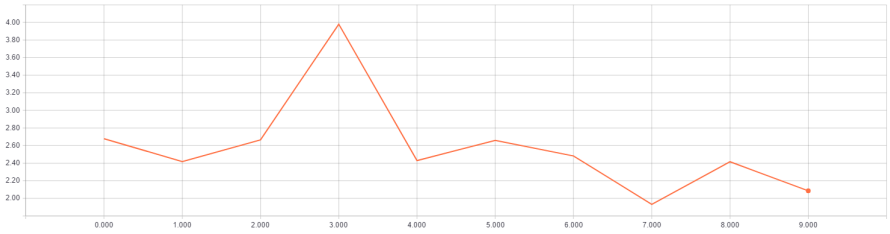


Figure 4.4: Graph showing the top-1 validation loss over epochs using the Adam backpropagation optimiser.

squared error. Another advantage of crossentropy error is that whereas mean squared error can lead to weight changes getting smaller and smaller over time, crossentropy used with a softmax or sigmoid activation function does not suffer from this problem.

4.2.3 Backpropagation method

Stochastic Gradient Descent with Nesterov Momentum is used for the backpropagation method, but there is another method which could be used - Adam. There is a considerable amount of literature on the details of SGD and Nesterov Momentum, but the discussion around this is out of the scope of this dissertation. A more detailed discussion can be found in CS231N's course material[64] and Nielsen's online book, Neural Networks and Deep Learning[65]. Two networks are compared in table 4.1, both using the same dataset, one using Adam and the other SGD with Nesterov Momentum.

Method	Initial Learning Rate	Initial Decay	Top-1 Accuracy	Top-5 Accuracy	Loss
SGD	0.01	0.000001	50.36	80.7	1.78
Adam	0.001	0	47.61	77.28	2.08

Table 4.1: SGD here refers to Stochastic Gradient Descent with Nesterov Momentum.

Stochastic Gradient Descent seems to be the best option of the two optimisers, considering there is little difference between the two but SGD is slightly better. SGD does require careful choosing of learning and decay rates, however, unlike Adam which changes the learning and decay rates as part of the algorithm. While there is little difference in accuracy, the validation accuracy and loss graphs for Adam suggest it is unsuitable for this problem. As can be seen in figures 4.4 and 4.3, the validation accuracy and loss is unstable compared to the earlier validation and accuracy graphs (which were using SGD). The reason for this instability is not clear, but whatever the cause, Adam has not helped to converge the network any faster than SGD, and it is not clear from the graphs that this network trained with Adam has even fully converged, given the same amount of time as the network using SGD.

4.2.4 Learning rate and weight decay

In order to work out the optimal learning and decay rates for this dataset, the network was trained on a smaller sample of the data with a range of learning and decay rates. To select the learning and decay rates, first a range of values from 10^-6 to 10^0 were tried. Subsequently, values between the best results were tried, continuing by halving the interval like in a binary search algorithm. For the learning rate experiments the decay rate was kept constant at 1^-6, and for the decay rate experiments the learning rate was kept constant using the best learning rate from the previous experiments. It would be more rigorous to test all the combinations of these values, as the interplay between them could produce interesting effects. However, this would take an inordinate amount of time. To do such an experiment would require a more intelligent search.

Learning Rate	Top-1 Accuracy (%)	Top-5 Accuracy (%)	Loss
1^{-6}	8.24	17.45	4.23
1^{-5}	15.15	26.95	3.77
0.0001	26.07	48.38	2.98
0.001	32.85	60.42	2.59
0.003	32.73	59.8	2.65
0.005	29.87	52.69	2.94
0.01	30	53.41	2.86
0.1	8.66	25.63	4.62
1	8	23.78	4.67

Table 4.2: Comparison of different learning rates. Accuracy and loss scores are on validation sets.

The learning rate which shows the best top-1, top-5 accuracy and loss in table 4.2 appears to be 0.001, followed closely by 0.003 and 0.01. As such, 0.001 will be used as the learning rate for the decay experiments.

Decay Rate	Top-1 Accuracy (%)	Top-5 Accuracy (%)	Loss
0.000001	32.85	60.42	2.59
0.00001	32.8	61.26	2.58
0.0001	33.79	62.05	2.52
0.001	34.84	62.47	2.5
0.01	25.74	48.39	2.99
0.1	15.7	29.13	3.69
1	10.92	22.09	4.13

Table 4.3: Comparison of different weight decay rates.

From table 4.3 is can be seen that the most effective weight decay rate appears to be 0.001 if combined with a learning rate also of 0.001. After 0.001 the accuracy rates drop sharply,

suggesting that anything higher than that causes the neural network to "forget" previous correctly learnt labels, leading to more errors. Generally a higher decay than learning rate is recommended. This is due to the fact that decay rate is meant to prevent weights that are not improving the network's loss very much from increasing. If the decay rate is higher, this means that a smaller amount of weights will be increased - only those whose loss is low enough to outweigh the decay rate in the backpropagation update equation. This varies from network to network however - deeper, more complex networks often require higher decay, as they have far more weights, and so need to be more discriminative on which weights increase each update. For this network, it seems that 0.001 is an optimal decay rate, at least for the validation set.

4.2.5 Preprocessing

As images from the training set are different sizes, and sometimes are black and white, they need to be resized and converted to RGB images. Images are also normalised by dividing each pixel's RGB values by 255, subtracting 0.5 and multiplying by 2. This scaling method is taken from the Xception network source code[66], and is a method which Chollet found to be effective. A standard approach involves ensuring the dataset when normalised has a mean of 0 and a variance of 1 as this lessens the chance of saturated neurons, which is likely what Chollet was trying to achieve here.

4.2.6 Validation

Validation sets are important for judging the performance of the network on unseen data as the network trains. Judging performance based on how accurately the network predicts training examples is one method, but may suggest better performance as the network learns a pattern that works with the training data but does not work with unseen data - a local minimum in the error graph, rather than the actual minimum error. To get a more accurate idea of how the network might perform on real unseen CAPTCHA data, validation data is used, split (10% of the training set) from the main training data and shuffled (like the rest of the training data). While many machine learning approaches use both a validation and a testing set to test the performance of a method (validation to test during training, testing to test afterwards), this is not useful for the problem at hand - the best testing set is the actual CAPTCHA images, as the testing set is meant to show the finished network's ability to generalise to unseen data in the problem domain.

4.2.7 Dataset

A dataset was designed based on the places2, German Traffic Sign Recognition and Street View House Number datasets. These were chosen as places2 contains a large number of photos of outdoor scenes such as store fronts and apartments, which regularly come up in CAPTCHAs (as shown later in figure 6.7), as do street signs and street numbers, which are the focuses of the GTSR and SVHN datasets. Using the whole of Places2 led to very long convergence times for the neural network, as combined with GTSR and SVHN the dataset would be over 1.7 million images. As such, certain classes that had been seen regularly in CAPTCHAs were chosen.

- Apartment Building
- Badlands
- Bamboo Forest
- Outdoor Bazaar
- Beach
- Beach House
- Botanical Garden
- Building Facade
- Bus Station
- Cabin
- Canyon
- Chalet
- Construction Site
- Corn Field
- Cottage

- Creek
- Department Store
- Desert
- Desert Road
- Downtown
- Embassy
- Farm
- Fastfood Restaurant
- Field
- Field Road
- Forest
- Forest Path
- Forest Road
- Formal Garden
- Fountain
- Gas Station
- General Store
- Hayfield
- Highway
- Hospital
- Hotel
- House
- Hunting Lodge
- Industrial Area
- Inn
- Kasbah
- Lagoon
- Lake
- Lawn
- Mansion
- Market
- Marsh
- Mosque
- Motel
- Mountain
- Moutain Path
- Mountain Snowy
- Museum
- Oast House
- Ocean
- Office Building
- Palace
- Railroad Track
- Residential Neighbourhood
- River
- Store front
- Shopping Mall
- Sky
- Snowfield
- Street
- Street Number
- Street Sign
- Subway Station
- Train Station
- Valley
- Village

This is a total of 71 classes, with over 300,000 training images. Considerably smaller than the whole of Places2, GTSR and SVHN, but still varied enough to both cover the common CAPTCHA queries, but also cover lots of images that are not related to the CAPTCHA queries but still appear in CAPTCHAs. The focus when choosing which classes to include was on outdoor scenes - buildings, scenery - common things that might be seen in the background of a street view image, as a lot of the CAPTCHA images seem to be drawn from Google Street View.

Chapter 5

Implementation

5.1 CAPTCHA test web page

In order to test the reCAPTCHA cracker, signing up for the Google reCAPTCHA program and registering a website domain was required. Once this was done, the security setting of the CAPTCHAs could be tweaked. This had three values, from "Easiest" to "Most Secure", with an increased security setting turning on Google's "security features", which are not described in detail (the impact of these is discussed in section 6.1.3) A site key was provided, alongside a snippet of HTML to include the JS for the reCAPTCHA.

In order to test the CAPTCHA cracker, a simple web page was designed. The page contains a single form with an input field, a label for the input and a submit button. Attached to the form is the reCAPTCHA div element, and the reCAPTCHA API JavaScript is included in the HTML Head. The reCAPTCHA JS inserts the iframes required for the initial checkbox and image challenge.

5.2 Captcha Interaction

This section discusses the browser interaction part of the project, particularly focusing on the captcha_input.py, captcha_interaction.py, captcha_elements.py and captcha_files.py files. The program's logic is tied together by the captcha_input.py file, so the logic will be described from the perspective of stepping through the running of that file. For the purposes of this section, "the program" refers to the browser interaction parts, and not the code needed to train the neural network, which is described in the following section.

5.2.1 The start function.

The program begins by running the start function, which bundles together the main logic of the program. Importantly, the CaptchaCracker is instantiated outside the start function, as values in CaptchaCracker should only be created once, and some behaviour described in later sections needs to restart the main part of the program.

5.2.2 CaptchaCracker Initialisation and Setup

Before running start, a new CaptchaCracker instance is created and some values are initialised using the __init__ function. Once start is called, the setup function of the CaptchaCracker class is called first. Setup is used for parts of the program that need to be instantiated if the program is restarted. Some parts of the program only need creating once - such as the NeuralNetwork object and the Splinter Browser object. Additionally in these functions the num_guesses and num_correct counts are set to 0. Once this is done, setup navigates the browser to the test

CAPTCHA page, and a new CaptchaElement instance is created to allow for interaction with the CAPTCHA on that page. To be clear, for the purposes of leaving the program running for a long time to crack as many CAPTCHAs as possible, the program needs to be robust and be able to start trying to solve CAPTCHAs again after an error. So the behaviour of the setup and init functions is designed so some parts of the program (such as guess counts and NeuralNetwork objects) are left untouched after a crash (and thus these parts are initialised in the init function), but other parts that need to be redone after a crash (like navigating to the right page) are initialised in setup so they will be reinitialised when the start function is called after a crash.

5.2.3 Initialising the Neural Network with an existing weights file.

As mentioned, during the initialisation of the CaptchaCracker, a new NeuralNetwork instance is created, specifying the path to a hdf5 (see section 5.2.3) file which contains the weights for the neural network produced during training. Initialising the network is discussed later in section 5.3.1.

HDF5

HDF5 is a binary data storage format allowing for the storage of large amounts of data in a relatively small file size. It is more compact than Numpy's npy format, and works well with Keras, through the h5py Python library.

5.2.4 Initial Checkbox

The program begins as a human user would do, by finding the initial "I am not a robot." checkbox and clicking it. This is just a case of switching context to the iframe that contains the checkbox, finding the element by its CSS selector, and clicking it using the click_element function. This function is a wrapper around Splinter's click function, as Splinter does not catch errors resulting from the element not being clickable at that precise moment - a more robust method (which the click_element function uses) involves catching the exception thrown and retrying up to 3 times - this avoids some errors due to slightly different timings due to load times, animations etc.

5.2.5 Checking if the CAPTCHA has been passed.

At this point, the program checks if a tick has appeared in the initial checkbox, which would mean that the CAPTCHA has been passed - the CAPTCHA thinks the program is a human. When the tick appears like this, the "recaptcha-checkbox-checked" class is added to the "recaptcha-anchor" element. Finding this element and checking for that class indicates whether the CAPTCHA has been passed or not. In order to allow for the animation of the tickbox into a tickmark, the program sleeps for 0.5 seconds at this point. If the CAPTCHA has been passed the browser is reloaded, the number of guesses and the number of correct guesses are incremented, and print_stats is called, which prints those numbers and a percentage calculated by dividing num_correct by num_guesses.

5.2.6 Gathering information about the new image challenge.

The purpose of the get_new_captcha function is to get all the information needed about the image challenge, and create the necessary Captcha and Checkbox objects to choose which checkboxes to click. There are several functions used to do this which are described below.

Finding the number of rows and columns in the image challenge grid.

The CAPTCHA images are displayed in a HTML table element, which has a class similar to "rc-image-tile-33" where the "33" indicates that the grid is 3x3. By splitting the classes of the table on the space character to exclude all other classes the table element might have, and then splitting on "-" and getting the last element of the resultant list, the program can get that number. This number can then be split in half and each part can be assigned to the rows and cols variables (after being cast to an integer from a string). Knowing how many rows and columns are in the grid is an essential step to being able to find all of the images and their checkboxes.

Finding the image used for the CAPTCHA.

Each checkbox in the CAPTCHA grid has an image element, with a lengthy unique URL to an image. However, this image is not the one the user sees for each checkbox. Instead, it is a single combined image for the whole CAPTCHA, each checkbox having the same URL to this combined image. CSS is used by the CAPTCHA to display only a part of the image per checkbox. For now, the program gets the src attribute of one of the checkboxes to get the URL for the combined image so that this can be split and saved later.

Getting the CAPTCHA query.

The CAPTCHA has a div element with the full CAPTCHA query in it, but the program needs the few words that indicate what needs to be selected. Luckily the CAPTCHA highlights this piece of text in bold, using a strong element to do it, so the program finds the div with the right class ("rc-imageselect-desc-no-canonical") and finds the strong element within that. This text is then turned into its singular form using the inflection library's singularize function, so that a simple difference like "mountains" and "mountain" between the CAPTCHA text and the labels will not cause problems.

Finding the clickable checkboxes.

The elements in the grid which can be clicked on are divs in a table. To find these elements, the program loops through the rows and columns, using nth-child CSS selectors to find the particular table cell in the grid for this row and column, joining selectors with the "¿" character which selects elements that match the element right of the "¿" and are children of the element on the left. For each checkbox element, a Checkbox object is created which stores its position in the grid as a tuple, the splinter element object itself, and the URL from the image element within the table cell.

Detecting whether a CAPTCHA has changed and downloading the images.

When the program is faced with a brand new CAPTCHA, all of the image URLs will be the same URL pointing to the combined image. Whereas, when the program has selected some images and new images have been presented to replace them, those images will have different URLs to the combined image. This seems to be the only way the change in images can be detected - by checking for any differences between each checkbox's URL and the main image URL. Once it has been ascertained whether the program is dealing with a new CAPTCHA or just some new images for the same CAPTCHA, the program either saves images for the new images (in the latter case) or saves the combined image and splits it into the parts that are displayed on each checkbox. There is also a possibility that this is the same CAPTCHA as previously seen, nothing has changed despite the program's attempts, and so the program should get a new CAPTCHA. The previous CAPTCHA's URLs are stored as an instance variable of

the CaptchaCracker class, and compared against - if the same as the new CAPTCHA's URLs, then an exception is thrown, otherwise that variable is set to the new CAPTCHA's URLs. This exception, like several others, is caught and leads to a browser reload. Before saving new images, old images in the root directory (where new CAPTCHA images are saved) are deleted, simply by finding all files with a jpg file extension and deleting them. As there are no other image files related to the project this is sensible behaviour, although images could be saved in a subdirectory to be extra safe. Downloading and saving the images involves getting the images using a HTTP GET request with the requests library, and saving the resultant stream as an image using the Pillow library. In the case of the combined image, the image is then cropped repeatedly. For each row and column in the grid, the image is cropped using that row and column number and the width and height (that is the width and height of the image divided by the total number of rows and columns, respectively). Specifically, the dimensions are $x = column * width, y = row * height, newwidth = column * width + width, newheight = row * height + height$. In the case of later, changed images (which are single images rather than a single collection of images in the same image file), no cropping is required and the images are just saved to the disk. In addition, the images are saved permanently in another location, specifically /datasets/captchas/captcha-query/random-string where captcha-query is whatever the CAPTCHA text for this CAPTCHA was, and the random string is an Universally Unique Identifier (UUID). This is so previously seen CAPTCHAs can be analysed at a later date. UUIDs are used as they have low collision chances, considering the large numbers of CAPTCHAs even for a single CAPTCHA query this an important feature. Many other random generated strings would likely have worked for this purpose however.

5.2.7 Preprocessing

Images just saved to the disk are now loaded, and resized, coloured and normalised as described in the design section. This preprocessing is the same as the preprocessing done to training images, so that the chance of the network classifying the image correctly is maximised. Image preprocessing is handled by the functions in the ImagePreprocessor class, with some assistance in renaming files from the FilepathPreprocessor.

5.2.8 Getting predictions.

The CAPTCHA images are loaded from the disk again, this time into a NumPy array using scikit image's imread_collection and the image paths of the checkboxes. These images are then fed through the network, getting a predictions array for each image. For each image, the predictions array is a probability that the image belongs to each class. This array is then filtered, only including class probabilities above a certain threshold (the optimal value of this is discussed in section 6.1.2). These probabilities are then sorted, returning a list of class labels for each image, with the highest probability class labels first. These label numbers are then converted to their corresponding label names by reading a text file with the name of the class (processed to change slashes and underscores into spaces) and the corresponding class label number on each line into a Python dictionary, and then finding the corresponding class name for the label number in the dictionary. These class names are singularised like the CAPTCHA query was previously. This reading of the categories file and singularisation is handled by the LabelProcessor class. This leaves the program with a list of lists, each sublist being a few phrases which are the neural network's class predictions for each image, in order of the image's appearance in the CAPTCHA.

5.2.9 Clicking the correct checkboxes.

Once the program has the neural network's predictions, it selects which predictions match the CAPTCHA query, and clicks the associated checkboxes. This is done by looping through the class labels for each image, and checking if any of the words in any of the class labels match any of the words in the CAPTCHA query (as previously described in section 2.3). If a match is found, then the corresponding checkbox is added to a set (rather than a list to make sure checkboxes are only added once). The set of checkboxes is then looped through, with each checkbox's element being clicked on in turn.

5.2.10 Refreshing the checkboxes.

As certain checkboxes may have been clicked, new images could have appeared, so it is important to find the new checkboxes and their associated URLs so it can be checked whether the CAPTCHA has changed, using those checkboxes.

5.2.11 Choosing to reload or verify.

At this point, the program's behaviour depends on whether there were any matching checkboxes and whether the CAPTCHA has changed since clicking on those matching checkboxes. The behaviour is described in pseudocode in algorithm 5.1.

Algorithm 5.1 Choosing to reload or verify.

```
if matching_checkboxes then
    if captcha_changed() then
        continue
    else
        verify()
        num_guesses ← num_guesses + 1
    end if
else
    if captcha_changed() then
        verify()
    else
        reload()
    end if
    num_guesses ← num_guesses + 1
end if
```

If the CAPTCHA has changed and the program still thinks there are matching checkboxes, then the program should continue clicking rather than verifying. If the program selected checkboxes and nothing has changed, it should just verify as there is nothing that can be done. If the program cannot find any matching checkboxes, then there is a chance the program is right, particularly after the CAPTCHA has changed because there may be no correct answers in the new images. If the program cannot find any matching checkboxes and no changes have happened, likely as not the program is wrong and should reload.

5.2.12 Handling exceptions.

There are several exceptions that can be thrown while the program is running. The first is the SameCaptchaException, which is thrown as described previously when all the image URLs are the same as the previous image URLs. Other exceptions are thrown because of an element not being found in the DOM for some reason, sometimes because the CAPTCHA has been solved

and so the elements have disappeared. In all of these exceptions' cases, the program can do nothing but refresh the browser page and run the start() function again. The reasoning for this behaviour has already been discussed in section 5.2.2.

5.2.13 Using time.sleep() for controlling the speed of interaction.

As mentioned, there are animations which limit the speed at which CAPTCHAs can be solved, as in order for the image checkboxes to be clicked the webpage elements must have updated in the DOM. Similarly, a small amount of time must be left to wait for the CAPTCHA initial checkbox tick animation. This is unfortunate as it slows the program down, but there is no way round it as reCAPTCHA is designed this way - in fact higher security settings increase the amount of time animations take so as to inconvenience bots like this one.

5.3 Neural Network

The Neural Network has only so far been discussed in terms of providing predictions of class labels for the main CAPTCHA cracking tool, but in order to do this it had to be trained, and for this to happen the training data had to be prepared and associated files had to be created from that data. This section discusses the code to achieve this.

5.3.1 Building the network.

The network is initialised in different ways depending on the constructor variables. If a weights file is supplied that will be loaded once the network model object has been made. Otherwise the network will start training as if there is no weights file then it can be assumed one needs to be created. There is also the possibility that the user wants to continue training from a partially trained network at a set point, in which case continue_training is set to True and the start_epoch variable is provided to indicate which epoch from which the training should resume. The neural network model itself is built using code imported from Keras's applications module. Once the model object is built, Keras requires it be compiled, which involves setting the learning rate, weight decay, and the method of backpropagation, as well as the metrics to use during training and the loss measure to use. After the model is built and compiled, the fit_generator function previously mentioned is used along with the training and validation data generators already described to train the network and save the weights to a h5fs file. If the user does not wish to train the network, Keras's load_weights function is used to load the weights file and alter the network model's weights accordingly. Without a weights file like this, the weights are randomly initialised.

5.3.2 Training the network.

To train the network, the program feeds minibatches of images through each layer of the network, updating the weights for each minibatch. Each epoch is a sequence of minibatches, with the total number of samples per epoch normally less than or equal to the dataset size. As the weights are updated every minibatch, the number of samples per epoch only defines practical details, such as when Keras's checkpoint callback functions are ran. The program uses two callbacks - one which outputs data for Tensorboard (a visualisation tool from Tensorflow that allows the user to view graphs of accuracy and loss) to use, and another which saves the weights file if the validation loss has improved. After each epoch is complete, minibatches of validation data are fed to the network, and one accuracy and loss figure is calculated for all of the validation minibatches. As mentioned, if this loss figure decreased from the previous epoch, the weights are saved to a h5fs file. One extra callback which could have been used was Keras's EarlyStopping callback, which stops training the network after a certain number of epochs with

no validation loss improvement. While the weights are already not saved if the validation loss does not improve, the network still continues trying to train. It would be more efficient for it to simply stop training at that point. That said, the number of epochs to go with no validation loss improvement needs to be configured carefully - too small and the network might not be converged enough when stopped.

Preprocessing

As the program is using a supervised learning algorithm to update the weights of the neural network, labels are needed alongside images. The training images are stored with each class being a folder, with some classes having subdirectories. For example, images with a class like "general store outdoor" would be stored in the directory "/general_store/outdoor", and images with the class "store_front" would just be stored in the directory "/store_front". The easiest way of storing the labels for these images is in a text file, with each line being a filepath and a label number. The filepath is relative to the dataset directory, like the paths given in examples above. To create this file, the program walks through the directories in the training dataset directory, getting the relative paths and using the count of how many root level directories it has seen already for the class label. It adds these relative paths and class label numbers to a list, and writes them to the text file, with a space in-between the path and the class label. This can now be read easily as each line is a label for a file with two clearly separated parts. This is handled by the FilepathPreprocessor, as it handles large numbers of filepaths and turns them into a labels file, rather than processing labels (which is what the LabelProcessor class handles). When training, the filepaths from the labels file have to be converted to their absolute versions so that images can be loaded using those paths (e.g. a filepath like store_front/2131.jpg is converted to E:\datasets\captcha-dataset\store_front\2131.jpg). The images from those filepaths are then resized (loading with scikit image as before), the filenames for the resized images are changed (adding "_widthxheight", e.g. "_93x93"), the images are converted to RGB, normalised as previously described, and then the labels for that minibatch and the images for that minibatch are yielded by the generator (i.e. given back to the main thread of execution, which is Keras's fit_generator function, which is doing the training). It should be noted here that rather than loading all the images into memory and splitting that massive array into minibatches, the lines from the labels file are loaded into memory, and those labels and filepaths are split into minibatches. Then for each batch of filepaths those images are loaded. This ensures that at any one time, only a small array of images (as large as the minibatch size) are loaded into memory, allowing for larger image sizes and much larger datasets on less hardware. Earlier iterations of the program loaded all of the images into memory and preprocessed all of them at once (or split the massive NumPy array into minibatches), but this did not scale to large datasets such as Places2, and is generally inefficient, even with a small dataset.

5.4 Config

Both the CAPTCHA interaction part of the program and the network rely on a config.py file, which contains a dictionary with several settings used across the programs. These settings are the CAPTCHA testpage URL, the path to the neural network labels file, the image size as a string (for preprocessing files) and as a tuple, the path to the neural network weights file, the path to the Tensorboard logs, the path to the dataset, the path to the categories file (containing the class label numbers and the names of the classes), and the number of classes in the dataset (this can be derived from the dataset, but it is inefficient to do so when the network is just being used to solve CAPTCHAs and not being trained). This allows for convenient access to values that are important to all of the program, or at least several parts of it.

5.5 CAPTCHA viewer

In order to gather results, it is useful to be able to look at CAPTCHAs after the cracker has attempted to solve them. As such, CAPTCHA objects and their associated Checkbox objects are serialised into JSON (specifically the guesses.json file, by default). The original paths for the checkbox images (i.e. the image_path variables) are not stored, as the images stored at those paths are regularly deleted. Instead, the images are saved to a permanent path when the images are saved during the CAPTCHA cracking process. This permanent path is stored in the JSON along with the associated Checkbox object. The aim of this serialisation, as mentioned, is to be able to recreate the CAPTCHAs later. This allows for the frequency of CAPTCHA queries to be counted, and also for a tool which presents a grid like the original CAPTCHA with the images, but also with the predictions and with the information of which images were picked. Matplotlib is used to create the graphs, specifically a graph of which queries the CAPTCHA cracker got correct, and which queries it saw in general. Matplotlib is also used to show the subplots of each image in each CAPTCHA, with their predictions above the images, whether the image was picked, the CAPTCHA query and whether the guesses were correct and resulted in a passed CAPTCHA.

5.6 Requirement Fulfilment

Most of the requirements outlined for the system were fulfilled. However, as described in section 2.3, the semantic similarity measure is not perfect and more work could be done to improve this. Similarly, in order to solve the "click and drag" CAPTCHAs, the system would need to use a multilabel classification method and be able to draw the outline of detected objects matching the CAPTCHA query. The improvements related to multilabel classification are discussed further in section 6.4.

	Requirement	**Estimated Cost**	**Fulfilled?**
1	It must tick the checkbox, bringing up the image CAPTCHA challenge.	1	Yes
2	It must download the image/images and segment them correctly (i.e. a single full image that is meant to be 9 images should be split into 9 images).	4	Yes
3	It must read the challenge text (such as "Select all the drinks.") and store it so it can be used for identifying which images are closest to the object specified.	2	Yes
4	It must classify the images provided to it from the program interacting with the CAPTCHA, giving each image a label (or several labels with accuracy values).	16	Yes
5	It must interact with the CAPTCHA, attempting to solve the challenge using the labels of the images provided by the other parts of the system.	2	Yes
6	It should judge labels from the classifier for semantic similarity to the challenge text object (e.g. store front is similar to shop in meaning, so an image labelled as a shop should be considered a good candidate for a CAPTCHA asking for store fronts) and use those similarity values to judge suitable solutions for the CAPTCHA challenge.	8	Partially
7	It should determine (from classifier's expected accuracy of label values) when it is unlikely to get the correct answer to a CAPTCHA, and click reload if that is the case.	4	Yes
8	It could attempt to solve "click and drag" image CAPTCHAs by finding the part of the image that an object resides and interacting with the CAPTCHA accordingly.	8	No

Chapter 6

Results

This project set out to test the statement, "Google's reCAPTCHA test is unsolvable by software." The main metric that can be used to discuss whether the program described in the previous chapters can be used to solve reCAPTCHA tests is counting the number it gets correct (i.e. the number of tickboxes seen).

6.1 Accuracy at solving CAPTCHAs.

In order to ascertain performance using this metric, it is reasonable to repeatedly attempt to solve CAPTCHAs, counting the number of CAPTCHAs correctly solved (where that can be detected by a tickmark appearing). From this a percentage accuracy score can be calculated. There are several different system variations that can be used for this, as such it is useful to consider these different setups and compare their performance. First, a control scenario is useful in that it defines a baseline for the system. Such a control scenario could be the program already described, but with the alteration that it randomly guesses rather than using the neural network.

6.1.1 Random Guessing

A clicker was designed to pick random checkboxes rather than using the neural network's predictions. It was ran for 1000 CAPTCHAs as in the later experiments. The results can be seen in table 6.1.

Guesses	Correct Percentage
1000	0

Table 6.1: Correct percentage of 1000 completely random attempts.

Randomly clicking barely ever correctly solves CAPTCHAs. From this, it can be concluded that brute forcing reCAPTCHA is largely impractical. It is possible that, ran for further experiments, the random clicker would eventually solve a CAPTCHA, so the real performance is unlikely to be exactly 0%. However it is low enough to consider a brute force approach infeasible. It can also be useful as a benchmark to compare with the results of later experiments.

6.1.2 Probability thresholds.

There are several factors to consider in the implementation of the CAPTCHA cracking system, as mentioned in the Implementation and Design chapters. One factor to be considered is the probability threshold. The neural network returns a list of probabilities of a particular image

belonging to a particular class. In practice, much of this list is full of very small, near 0 values, with any of these values indicating that the image is unlikely to be of the associated class (so the 64th value being 3.7^{-8} is usually indicative that the image does not belong to class 64). As the program needs to decide which class labels are what the network thinks is correct, the program filters these values out by using a probability threshold. A table showing the percentage accuracy for several probability thresholds can be seen in table 6.2.

Guesses	Probability Threshold	Correct Percentage
1000	0.01	9
1000	0.1	4.8
1000	0.5	0.9

Table 6.2: Percentage of CAPTCHAs solved with different probability thresholds used to filter the results of the neural network.

From these results (table 6.2), a low probability threshold seems desirable, as the performance halves with a small increase in the probability threshold. The fact that setting the probability threshold higher than 0.01 caused a significant drop in performance (as seen in table 6.2 suggests that the difference in probability between correct and incorrect labels for an image is low, as large amounts of correct labels are cut out with anything above 0.01. So while the network is able to somewhat distinguish the classes, it does so with little confidence. This suggests that the dataset used does not provide enough data, or enough varied data, to fully characterise the classes.

6.1.3 The impact of security settings.

Up until this point, all previous experiments were ran using the "Easiest" reCAPTCHA security setting, but it is not clear what this affects. If it only affects checks on the initial checkbox then sites using the "Most Secure" setting are still vulnerable to this approach. To assess the impact of the security setting, the best result from the previous threshold experiments was re-ran with different security settings.

Guesses	Security Setting	Correct Percentage
1000	Medium	7.6
1000	Most Secure	0.9

Table 6.3: Impact of security settings on correct percentages of the CAPTCHA cracker.

Most noticeable in table 6.3 is the impact of the Most Secure setting. Transition animations (e.g. new images fading in, clicked images fading out) in CAPTCHAs using the Most Secure setting are noticeably longer, which requires the cracker to be specially adapted to work with those timings, and also in general makes the cracker less efficient at solving CAPTCHAs over time. It also seems that the CAPTCHAs are less forgiving in terms of the number of wrong checkboxes which are acceptable, and how many correct CAPTCHAs have to be completed before a tickmark is granted. Clearly, more than just the initial checkbox checks are affected by the security setting, as getting a tickbox from simply clicking the checkbox barely ever happens, even on the easiest security setting.

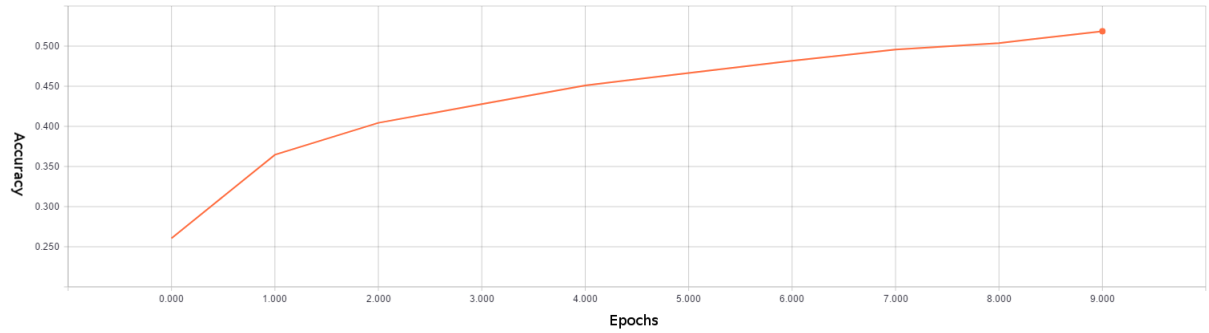


Figure 6.1: Top-1 accuracy per epoch during training.

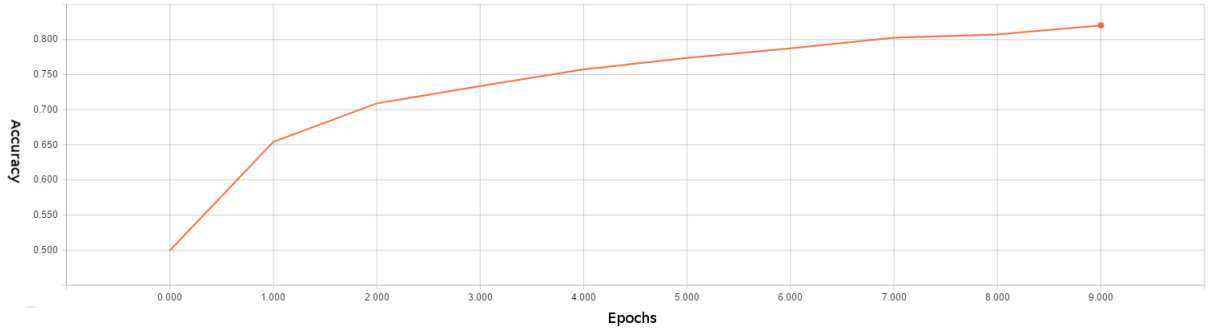


Figure 6.2: Top-5 accuracy per epoch during training.

6.2 Neural network performance

The program can crack some CAPTCHAs, but it does not do so at an accuracy rate comparable to similar solutions on older CAPTCHA schemes. Could this low accuracy rate be related to the neural network architecture used, or the training dataset? To discuss this, it is useful to look at both validation and training accuracy and loss to judge whether the network was fully and properly trained and whether it performed well on validation data.

Figure 6.1 shows that the network is performing as expected. The accuracy increases rapidly to begin with, as random weights are corrected to weights far closer to the image classes. The curves in all graphs are a gradual increase after a rapid start - this is a good indication that the network is mostly not overfitting, as overfitting would lead to a downward curve partway through training (as the network trains on data that doesn't fit the patterns learned because of overfitting), and a lower validation accuracy curve. Also, the accuracy curves have not entirely flattened out - this is not necessarily bad - it ensures the network is not overfitting, but it could

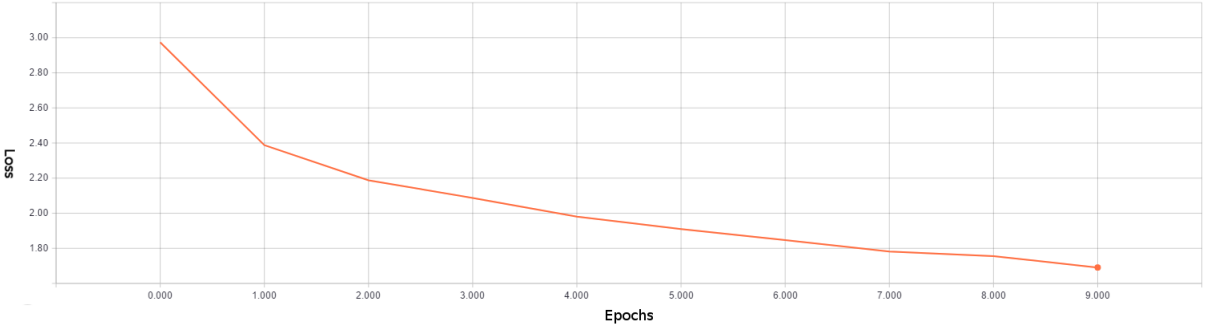


Figure 6.3: Loss per epoch during training.

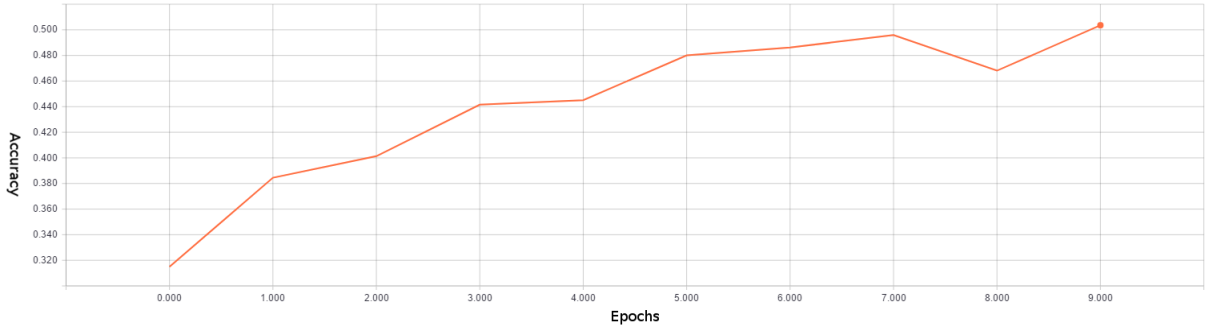


Figure 6.4: Top-1 accuracy on the validation set per epoch.

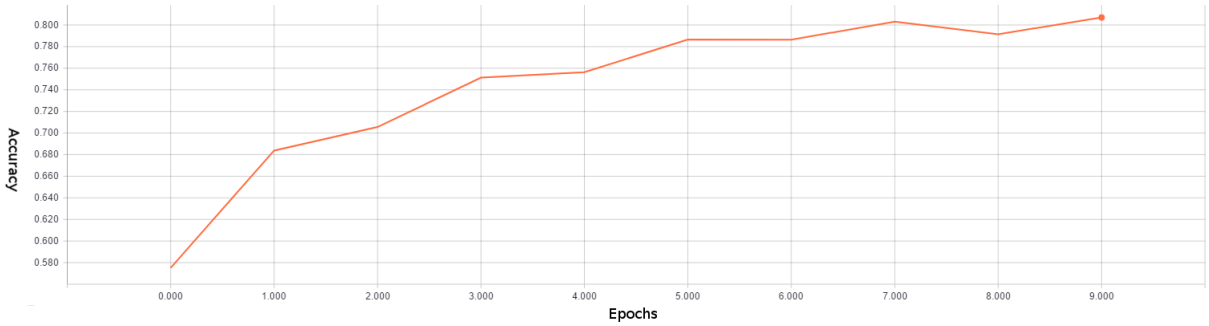


Figure 6.5: Top-5 accuracy on the validation set per epoch.

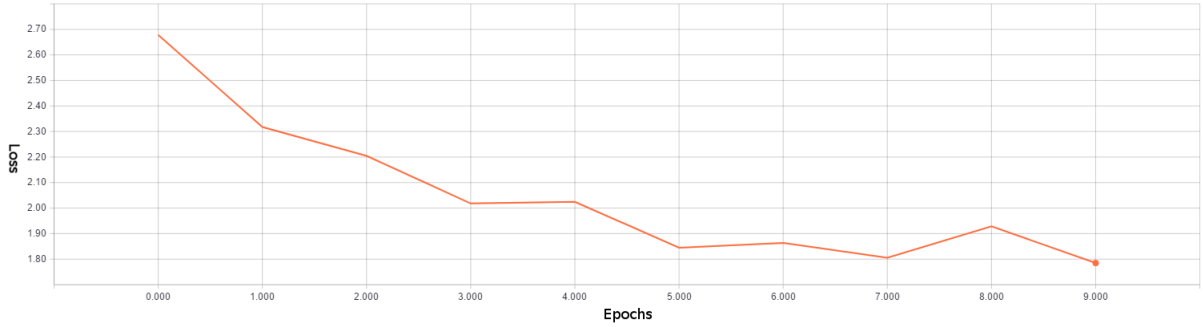


Figure 6.6: Loss per epoch on the validation set

mean the network has not fully finished training. The gradient is not too steep around epochs 9 and 10, so it is unlikely that additional learning could occur. In the validation loss (figure 6.6) and accuracy (figure 6.4) graphs, there is a slight change in gradient. This could be a small amount of overfitting during epoch 9. The network could be learning patterns up to that point that fit the training data but do not generalise well to the unseen data presented in validation for that epoch, leading to a dip in performance. However, this does not present too much of a difficulty as the performance of the network improves in epoch 10, suggesting the network is not particularly overfit. A small amount of overfitting is tolerable - if the graphs here showed a lot of sudden, large gradient changes, further methods to reduce overfitting would have to be used, or a more varied dataset.

Top-1 Accuracy (%)	Top-5 Accuracy (%)	Loss
50.36	80.7	1.78

Table 6.4: Accuracy and loss on the entire dataset, trained using 0.01 learning rate and 1^{-6} decay rate, for 10 epochs (equivalent to twice through the whole dataset).

It was previously mentioned that a learning rate of 0.001 and a decay rate of 0.001 would perform better than the accuracy and loss scores mentioned in 6.4 on validation data. However, this did not translate to accuracy on actual CAPTCHAs. In tests on the Medium security setting, the network with 0.001 learning and decay rates performed at 6.4% accuracy, while using a 0.01 learning rate and 1^{-6} decay rate performed at 7.6% accuracy. The reasoning behind this is unclear - a better validation accuracy should translate roughly to a better CAPTCHA accuracy. It is most likely down to flaws with the dataset, in that the validation dataset does not represent unseen CAPTCHA images well enough.

All that said, table 6.4 shows that this neural network still has reasonable accuracy scores on the dataset, but as shown by the previous CAPTCHA tests, an accuracy score of 50% does not translate to 50% accuracy on CAPTCHAs. This could be caused by two problems. The first of these problems could be that the network architecture may not be able to generalise well to new data. Alternatively, the training dataset may not be well suited to the task, either missing classes, not having enough data for certain classes, or having data that leads to systematic errors.

6.2.1 Architecture

Firstly, the suggestion that the issue is with the architecture seems unlikely. In Chollet's paper on the Xception architecture, he provides results for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC, 1000 categories, 1.2 million training images) which suggest that the architecture itself is perfectly capable of achieving very good accuracy scores on unseen data. These results have already been discussed in comparison to other architectures in section 2.2.6. The reasons for the accuracy difference between the ImageNet performance shown in Chollet's paper, and the results attained by training on the dataset used for this project, are more difficult to ascertain. It is likely that the network used for Chollet's experiments was trained for longer, and more experiments could have been done to find optimal hyperparameters (learning rates, decay rates). Equally the performance of the network on validation data has already been shown to be reasonable, which suggests not that the problem lies with the network but that it lies with the dataset's representation of CAPTCHA data.

6.2.2 Dataset

The dataset was designed to achieve the breadth of classes needed to solve a variety of CAPTCHAs, while also getting the depth of data needed to properly recognise frequent classes. Certain classes

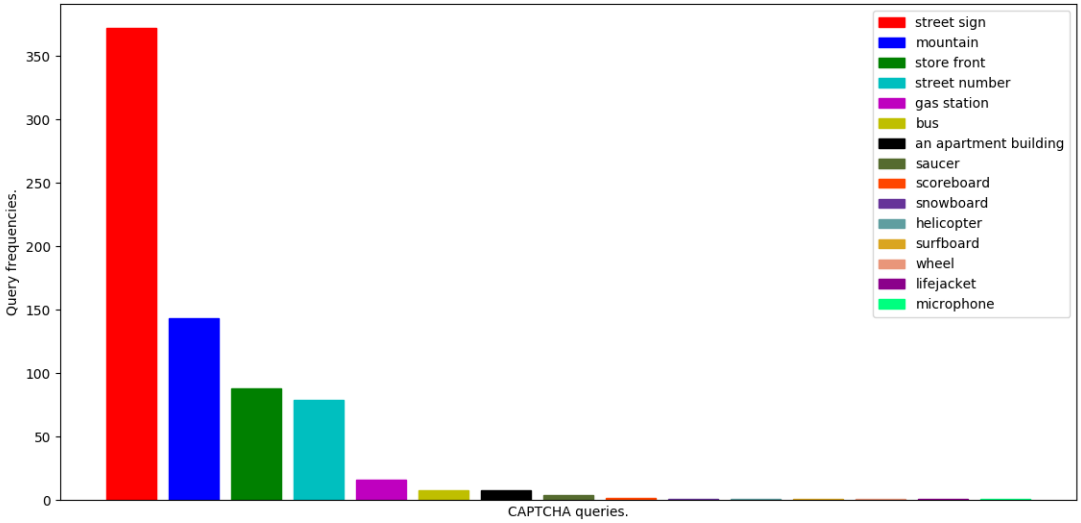


Figure 6.7: A chart of the different queries seen in 726 runs of the CAPTCHA cracker.

appear more frequently, as can be seen by measuring the frequency of CAPTCHA queries. Figure 6.7 shows the frequencies for different CAPTCHA queries over 726 CAPTCHAs. There are clear common classes, with "street signs" occurring for more than half of the CAPTCHAs. That said, these queries are not fully representative of the images shown. Some of the images in a street sign CAPTCHA are street signs, and of course some are not. While the queries can be used to build an effective dataset, they cannot be relied upon, as the network must be able to recognise which images do not match the query as well as it can recognise images that do. It seems likely that the loss of performance on validation and on the CAPTCHAs is due to the quality of the dataset, and in terms of the CAPTCHA performance, down to the mismatch between the validation dataset and the CAPTCHA dataset. This is particularly noticeable in some areas, such as street signs, where there are very few datasets available and as such the variety of the data to train and validate on is limited, translating to poor performance on much more varied CAPTCHA street sign images. Gathering a large enough dataset to perform well on a variety of CAPTCHAs, while also ensuring the images used are high enough quality to well distinguish the classes is a difficult task.

Using a large dataset to solve CAPTCHAs

Considering that the sheer variety of Google's dataset is a problem for constructing the dataset, a seemingly obvious solution would be a larger dataset, constructed of a large number of classes. To test this, a dataset was constructed from a combination of places2, the German Traffic Sign Recognition and Street View House Numbers datasets. This makes for a large dataset (over 1.7 million images, 368 classes), and the performance in terms of validation accuracy and loss is noticeably different from the smaller dataset, with 32% top-1 validation accuracy, 59% top-5 validation accuracy and 3.078 validation loss. It is possible that more experimentation could be done to improve these results (using different learning rates, decay rates), but to properly test improvements on such a large dataset is a very time consuming process, taking several days at a time. Of course, these are still reasonable scores, and don't imply on their own that the CAPTCHA cracker with such a dataset would be incapable of solving CAPTCHAs.

The issue with such a dataset for this problem, is that many of the classes in the dataset never occur in CAPTCHA queries. While a certain amount of these classes is useful to identify

Guesses	Percentage Correct
1000	0.1

Table 6.5: Results of using the large dataset on CAPTCHAs.

Figure 6.8: A particular CAPTCHA, asking for images of mountains, with the predictions from the network trained on the much larger dataset.

images that are not anything to do with the CAPTCHA query, too many of them increase the likelihood of an image being misclassified when it comes to more difficult image classes like street signs and store fronts. This translates to dramatically worse CAPTCHA performance, as can be seen in table 6.5. As previously mentioned, not all the CAPTCHA solutions here were wrong, but it did not get enough correct to pass the challenge. It is helpful to look in more detail at some individual CAPTCHAs.

In figure 6.8, the picture on the middle right does contain some houses - possibly a village, so the label is correct, but irrelevant to the CAPTCHA. Similarly, the image in the bottom left corner is labelled as "badland", although this is badly singularised, it is still correct - and again not relevant. This is an issue with a dataset with more classes - the amount of data per class also needs to be increased so that similar classes can be distinguished.

Again, in figure 6.9 the issue with too many classes and not enough depth to each class can be seen. Store fronts are particularly tricky as they appear similar to many other urban street scenes. In the top right there is a picture of what is most likely a store front, but is classified as a fastfood restaurant, traffic sign or a motel. Besides the traffic sign, both of those classes seem reasonable - it does seem like the image might be of a restaurant or cafe.

This does not dismiss the idea of a larger dataset being more effective entirely. What it does prove is that using a simple combination of publicly available datasets is not enough to solve CAPTCHAs effectively. Instead, any dataset, large or small, needs to be carefully constructed, making sure that enough varied data is included for each class to be able to distinguish those classes even when similar in some respect, as is ever too common in CAPTCHAs.

Figure 6.9: Another CAPTCHA from the large dataset tests, this time asking for store fronts.

6.3 Conclusion

In summary, this program successfully disproves the hypothesis that "Google's reCAPTCHA test is unsolvable by software." A program has been devised that, even on the "Most Secure" setting, can still solve almost 1% of CAPTCHAs, and on the easiest setting can solve 9%. While this is sufficient to prove that reCAPTCHA is not guaranteed as secure, it does not prove that the reCAPTCHA scheme is obsolete. The results gathered on different neural network architectures and different datasets suggest that to crack reCAPTCHA effectively a specially made dataset is required, as a simple combination of publicly available datasets is largely ineffective. It can also be concluded that the security settings are largely effective at causing issues for even intelligent bots. The results showed that brute force was largely impractical on any security setting and perhaps is more difficult on higher security settings, but the security features used by Google (both in terms of animation speeds and also in terms of less tolerance of CAPTCHA solutions) prevent more than brute force. While reCAPTCHA cannot be said to be broken from the work presented here, there are several improvements which could lead to better performance.

6.4 Improvements

6.4.1 Multilabel classification

Almost all of the images seen in CAPTCHAs contain multiple different objects, often several buildings or types of terrain. A single label classification system with a dataset with single labels for each image, like the one used for this project, can often fall into the trap of being correct but not having the right label for the CAPTCHA at hand. For example, in figure 6.10 one of the images is labelled with "desert sand" which seems correct, as the image contains what looks to be a desert, or possibly a field of some kind. But the image also contains a mountain in the background, and as the CAPTCHA is asking for mountains, this image is not picked. With a multilabel solution part of that image would have been labelled with "desert" and part would have been labelled "mountain", and so this image would have been picked. This is even more common with images of street signs, where often the background is of a different class,

Figure 6.10: The image on the middle left hand side shows both what could be construed as sand (but may be a field of some kind) and a mountain in the background. Whatever the foreground is, the important part is the mountain and this is missed as only a single label is assigned to the image.

leading to the whole image being labelled according to the background contents rather than the street sign. A multilabel solution would also allow for better solving of CAPTCHAs where the CAPTCHA is a single image chopped into parts, as the program could take the whole image, identify the outline of the different objects in the image (such as street signs), and then work out which checkboxes this outline overlapped with.

The issue with this approach is the requirement of a dataset of images with every part of the image labelled. That said, although there are less datasets like this, such datasets do exist, such as LabelMe which was mentioned earlier.

6.4.2 Data augmentation

Certain datasets, like the street sign dataset, and also to a lesser extent the street number dataset are quite limited in the variety of images included. As both these classes appear regularly in CAPTCHAs, improvements to these datasets would be likely to have large impacts on performance. One possible way of increasing the variety would be to use data augmentation - that is, applying transformations to the images in the dataset to create new versions. For example, by applying rotations to the street signs or house numbers. This ensures the network learns the characteristics of a street sign, rather than expecting all street signs to be the same orientation.

6.4.3 Dataset improvements

Certain parts of the dataset could easily be improved, without resorting to a multilabel solution. Some relatively frequent classes are missing, such as “cars” and other vehicles. A good solution would be to use the frequency data already obtained, and cross-reference with the existing classes in the dataset to see which classes might need to be added. Breadth of classes is not the only problem here - in fact a larger problem is the quality of images. The lack of variety in the

street sign dataset has been mentioned previously, but it could be improved by using images from CAPTCHAs and hand labelling them. This could be done by using an online service such as Amazon's Mechanical Turk to outsource the work of labelling to paid labellers.

6.4.4 Parallelisation

For much of the CAPTCHA cracking process, the neural network is not actively providing predictions. Rather than having a single cracker occasionally ask for predictions for images, it would be better to design for several cracker programs to be able to run at once, connected to the same neural network providing predictions to all of them with some kind of queueing system to manage the multiple cracker requests. This would make the overall system considerably efficient in terms of volume of attempted (and also cracked) CAPTCHAs.

Chapter 7

Conclusion

This project aimed to test the hypothesis that, "Google's reCAPTCHA test is unsolvable by software." This is a fundamental part of the reCAPTCHA scheme - if reCAPTCHA can be solved by software, it cannot be said to be able to "tell computers and humans apart" - the purpose of any CAPTCHA scheme. The results presented here proved that the reCAPTCHA scheme is solvable by software, and thus disproved the hypothesis. However, that is not to say that reCAPTCHA is obsolete. On the contrary, this system is most likely more elaborate than many spam bots, and requires extra resources that are not cheap, such as high-end Graphical Processing Units. It is also not effective enough to claim reCAPTCHA as broken. The program cannot solve CAPTCHAs very quickly due mostly to browser animations, particularly on higher security settings, and most crucially it does not solve CAPTCHAs with anywhere near the reliability that tools designed to crack previous CAPTCHA schemes have achieved.

7.1 Findings

Through experimentation, it was found that using the Xception architecture was the most suitable for this and perhaps for similar image recognition problems. It was also found that a dataset with fewer classes achieved better results than a much larger, more varied dataset. It cannot be concluded from this that a larger dataset would not be effective at this problem, in fact it seems unlikely that that would be the case considering the existing literature's use of massive datasets to great effect. Further experimentation would need to be done with different datasets (possibly even involving the creation of new datasets) in order to conclude the best dataset for the problem, but it seems likely that the use of a multi-label dataset would be a better approach. Using a dataset like LabelMe and classifying every object in CAPTCHA images would allow for much better decision making when it comes to deciding what to click on, as it would provide the program with far more information about the images presented. Backpropagation methods were also compared, and from the findings presented, it is clear that while Adam may promise much in terms of automatically adjusting learning and decay rates, care must still be taken in terms of the initial values in order to get expected results, much like Stochastic Gradient Descent. Again, although the findings presented here showed unusually unstable loss for the dataset used, these findings are not enough to dismiss the backpropagation method, even just for this particular problem. In the experiments done here, reCAPTCHA's security settings were effective in disrupting attempts to crack the scheme. Features like longer animations on images in the CAPTCHA may be a minor inconvenience for human users, but had a much larger impact on a bot, as they dramatically decreased the rate at which CAPTCHAs could be attempted. However, while reCAPTCHA is largely implemented in a secure fashion, there was a surprising key weakness that allowed for this program to be constructed. There is virtually no limit to the amount of CAPTCHAs a user can attempt to solve. Once, an old style CAPTCHA was encountered after several hundred incorrect guesses, but this is not consistent.

Simply closing the CAPTCHA window, forcing the bot to click the checkbox again would be an inconvenience, and if the same user has attempted 1000 CAPTCHAs, perhaps they are not human and should be prevented from retrying, even if just for a period of time. This surprising lack of a retry limit was what made testing and developing this software viable.

7.2 Final Conclusions

The software was made to break reCAPTCHA to the level of the previous reCAPTCHA scheme. While this has not been achieved, what has been suggests that the current reCAPTCHA scheme's days are numbered. Further work in designing the image recognition method, and a better dataset, could lead to much better performance. While this software only solved 9% of CAPTCHAs under the best conditions, this still proves that image recognition is a challenge which does not fully separate humans and computers. There are other areas of sensory recognition where the challenges may be harder; reCAPTCHA already incorporates an audio CAPTCHA which may be more difficult than the image challenge. But these are also only temporary challenges in which machines are constantly improving. Future CAPTCHA designers will need to think more carefully about the fundamental differences between human and machine.

Chapter 8

Appendix

.1 Convolution

Convolution has many applications in image processing, including edge detection, blurring, embossing, and several other filtering techniques. Image processing specifically uses two dimensional convolution - that is, the convolution of an image and a given filter matrix (also known as a kernel). It is useful to imagine taking a square of some size, placing it on the image matrix, doing some calculation for the value of the cell in the output matrix that corresponds with the location where the square was placed, and then moving the square by some defined amount (the stride of the convolution). Formally speaking, the convolution of two matrices can be defined according to the equation below, where p is a pixel in the image, k is a value in the kernel and d is the size of the kernel (as the kernel is a square matrix, this will be a value like 3 for a 3x3 kernel, or 5 for a 5x5 etc).

$$V = \left| \frac{\sum_{i=1}^{d} \left(\sum_{j=1}^{d} k_{i,j} \cdot p_{i,j} \right)}{\sum_{i=1}^{d} \left(\sum_{j=1}^{d} k_{i,j} \right)} \right| \tag{1}$$

That is, calculate the product of every kernel value with every pixel covered by the kernel (i.e. up to the side length of the kernel, d, as it is a square), and sum these values together, dividing by the sum of all the kernel values (or 1 if the sum is 0) and get the absolute value of that. This value is used as the output value at the point where the kernel was placed in the image. This process would then be repeated, moving the kernel by some amount to the right (or down by a certain number of rows at the end of a row).

Bibliography

[1] Hector Garcia-Molina and Zoltan Gyongyi. "Web Spam Taxonomy". In: *First international workshop on adversarial information retrieval on the web (AIRWeb 2005)* (2005), pp. 1–9. URL: http://ilpubs.stanford.edu:8090/771/1/2005-9.pdf.

[2] Luis Von Ahn et al. "CAPTCHA: Using Hard AI Problems For Security". In: *Advances in Cryptology, Eurocrypt, Lecture Notes in Computer Science* 2139 (2001), pp. 294–311.

[3] Gopi Chand. *Making CAPTCHA in JSP*. 2014. URL: http://www.c-sharpcorner.com/UploadFile/9a9e6f/making-captcha-in-jsp/ (visited on 04/30/2017).

[4] Elie Bursztein et al. "How good are humans at solving CAPTCHAs? A large scale evaluation". In: *Proceedings - IEEE Symposium on Security and Privacy*. 2010, pp. 399–413. ISBN: 9780769540351. DOI: 10.1109/SP.2010.31.

[5] Ian J. Goodfellow et al. "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks". In: *CoRR* abs/1312.6 (2013), pp. 1–13. arXiv: 1312.6082. URL: http://arxiv.org/abs/1312.6082.

[6] Claudia Cruz-Perez et al. "Breaking reCAPTCHAs with unpredictable collapse: Heuristic character segmentation and recognition". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7329 LNCS (2012), pp. 155–165.

[7] ReCaptchaReverser. *InsideReCaptcha*. 2014. URL: https://github.com/neuroradiology/InsideReCaptcha (visited on 11/23/2016).

[8] Sai Rahul. *Random Thoughts - Linear Separability*. URL: http://blog.sairahul.com/2014/01/linear-separability.html (visited on 11/25/2016).

[9] Primoz Potocnik. *Solving XOR problem with a multilayer perceptron*. URL: http://lab.fs.uni-lj.si/lasin/wp/IMIT%7B%5C_%7Dfiles/neural/nn04%7B%5C_%7Dmlp%7B%5C_%7Dxor/ (visited on 11/25/2016).

[10] Giles M. Foody et al. "Training set size requirements for the classification of a specific class". In: *Remote Sensing of Environment* 104.1 (2006), pp. 1–14. ISSN: 00344257. DOI: 10.1016/j.rse.2006.03.004.

[11] Aude Oliva and Antonio Torralba. "Modeling the shape of the scene: A holistic representation of the spatial envelope". In: *International Journal of Computer Vision* 42.3 (2001), pp. 145–175. ISSN: 09205691. DOI: 10.1023/A:1011139631724.

[12] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*. Vol. I. 2005, pp. 886–893. ISBN: 0769523722. DOI: 10.1109/CVPR.2005.177.

[13] Olivier Chapelle, Patrick Haffner, and Vladimir N. Vapnik. "Support vector machines for histogram-based image classification". In: *IEEE Transactions on Neural Networks* 10.5 (1999), pp. 1055–1064. ISSN: 10459227. DOI: 10.1109/72.788646. arXiv: 9411012 [chao-dyn].

[14] Yuanqing Lin et al. “Large-scale image classification: Fast feature extraction and SVM training”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* October (2011), pp. 1689–1696. ISSN: 10636919. DOI: 10.1109/CVPR.2011.5995477.

[15] Jacob J Merler M. *Breaking an Image Based CAPTCHA*. Tech. rep. 2009. URL: http://www.cs.columbia.edu/%7B~%7Dmmerler/project/Final%20Report.pdf.

[16] Grigory Antipov et al. “Learned vs. Hand-Crafted Features for Pedestrian Gender Recognition”. In: *Proceedings of the 23rd ACM international conference on Multimedia - MM '15*. October. 2015, pp. 1263–1266. ISBN: 9781450334594. DOI: 10.1145/2733373.2806332. URL: http://dl.acm.org/citation.cfm?doid=2733373.2806332.

[17] Michael Nielsen. *Neural Networks and Deep Learning Chapter 1*. 2016. URL: http://neuralnetworksanddeeplearning.com/chap1.html (visited on 11/24/2016).

[18] Y. LeCun et al. “Learning algorithms for classification: A comparison on handwritten digit recognition”. In: *Neural networks: the statistical mechanics perspective* 2 (1995), pp. 261–276.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances In Neural Information Processing Systems* (2012), pp. 1–9. ISSN: 10495258. DOI: 10.1016/j.protcy.2014.09.007. arXiv: 1102.0183.

[20] Wikipedia. *Generalised logistic function*. 2009. URL: http://en.wikipedia.org/w/index.php?title=Generalised%7B%5C_%7Dlogistic%7B%5C_%7Dfunction%7B%5C&%7Doldid=328966015 (visited on 12/02/2016).

[21] Wikimedia. *Tanh Function*. URL: https://upload.wikimedia.org/wikipedia/commons/d/d0/Tanh.png (visited on 12/02/2016).

[22] Yann LeCun et al. “Efficient BackProp”. In: *Lecture Notes in Computer Science* 1524 (2002), pp. 9–50. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[23] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision*. Vol. 11-18-Dece. 2016, pp. 1026–1034. ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852.

[24] Int8. *ReLU Function*. URL: http://int8.io/wp-content/uploads/2016/01/relu.png (visited on 12/02/2016).

[25] Y-Lan Boureau, Jean Ponce, and Yann LeCun. “A Theoretical Analysis of Feature Pooling in Visual Recognition”. In: *Icml* (2010), pp. 111–118. URL: http://www.ece.duke.edu/%7B~%7Dlcarin/icml2010b.pdf.

[26] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. 2016. URL: http://cs231n.github.io/classification/ (visited on 12/02/2016).

[27] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors.” In: *CoRR* abs/1207.0 (2012), pp. 1–18. ISSN: 9781467394673. arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

[28] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[29] Nelson Morgan and Hervé Bourlard. “Generalization and Parameter Estimation in Feedforward Nets: Some Experiments”. In: *Adv. NIPS 2* (1989), pp. 630–637. URL: http://papers.nips.cc/paper/275-generalization-and-parameter-estimation-in-feedforward-nets-some-experiments.

[30] Thorsteinn S Ragnvaldson. “A simple method for estimating the weight decay parameter.” In: *CSETech* 77 (1996).

[31] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks For Large Scale Image Recognition”. In: *ICLR*. 2015.

[32] Min Lin, Qiang Chen, and Shuicheng Yan. “Network In Network”. In: *CoRR* abs/1312.4 (2013).

[33] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 07-12-June (2015), pp. 1–9. ISSN: 10636919. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842.

[34] Randall C. O’Reilly et al. “Recurrent processing during object recognition”. In: *Frontiers in Psychology* 4.APR (2013). ISSN: 16641078. DOI: 10.3389/fpsyg.2013.00124. arXiv: 1512.03385. URL: http://arxiv.org/pdf/1512.03385v1.pdf.

[35] François Chollet. “Deep Learning with Separable Convolutions”. In: *CoRR* abs/1610.0 (2016), pp. 1–14. arXiv: 1610.02357. URL: https://arxiv.org/abs/1610.02357.

[36] Philip Resnik. “Using Information Content to Evaluate Semantic Similarity in a Taxonomy”. In: *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1 - IJCAI’95*. Vol. 1. 1995, p. 6. ISBN: 1-55860-363-8, 978-1-558-60363-9. URL: http://arxiv.org/abs/cmp-lg/9511007.

[37] George Miller, Christiane Fellbaum, and Randee Tengi. *WordNet*. URL: http://wordnet.princeton.edu/wordnet (visited on 12/02/2016).

[38] Yuhua Li, Zuhair A. Bandar, and David McLean. “An approach for measuring semantic similarity between words using multiple information sources”. In: *IEEE Transactions on Knowledge and Data Engineering* 15.4 (2003), pp. 871–882. ISSN: 10414347. DOI: 10.1109/TKDE.2003.1209005.

[39] Wikimedia Foundation and Wikipedia Editors. *Wikidata*. URL: https://www.wikidata.org.

[40] Han Zhao, Zhengdong Lu, and Pascal Poupart. “Self-adaptive hierarchical sentence model”. In: *IJCAI International Joint Conference on Artificial Intelligence* 2015-Janua (2015), pp. 4069–4076. ISSN: 10450823. arXiv: 1504.05070. URL: http://arxiv.org/pdf/1301.3781v3.pdf.

[41] Susan T Dumais et al. “Automatic Cross-Language Retrieval Using Latent Semantic Indexing”. In: *AAAI Technical Report SS-97-05.* (1997), pp. 18–24.

[42] Splinter Developers. *Splinter Documentation*. URL: https://splinter.readthedocs.io/en/latest/ (visited on 10/28/2016).

[43] Kenneth Reitz. *Requests Documentation*. 2015. URL: http://docs.python-requests.org/en/master/ (visited on 10/28/2016).

[44] BeautifulSoup Developers. *BeautifulSoup Documentation*. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (visited on 02/10/2016).

[45] Selenium Developers. *Selenium Python Documentation*. URL: https://selenium-python.readthedocs.io/ (visited on 10/28/2016).

[46] Stanford University and Princeton University. *ImageNet*. URL: http://image-net.org/ (visited on 12/04/2016).

[47] Bryan Russell and Antonio Torralba. *LabelMe*. URL: http://labelme.csail.mit.edu/.

[48] G Griffin, A Holub, and P Perona. *Caltech-256 object category dataset*. Tech. rep. 1. 2007, p. 20. URL: http://authors.library.caltech.edu/7694.

[49] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. 2009, pp. 1–60. URL: http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:Learning+Multiple+Layers+of+Features+from+Tiny+Images%7B%5C#%7D0.

[50] Toronto University. *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/%7B~%7Dkriz/cifar.html.

[51] Yuval Netzer and Tao Wang. *Reading digits in natural images with unsupervised feature learning*. Tech. rep. 2011, pp. 1–9. URL: http://research.google.com/pubs/archive/37648.pdf.

[52] Institut für Neuroinformatik. *German Traffic Sign Recognition Benchmark*. URL: http://benchmark.ini.rub.de/?section=gtsrb%7B%5C&%7Dsubsection=dataset.

[53] B. Zhou et al. "Places: An Image Database for Deep Scene Understanding". In: (). arXiv: 1610.02055.

[54] Tensorflow. *Tensorflow*. URL: https://www.tensorflow.org/ (visited on 12/02/2016).

[55] Hao Dong. *TensorLayer*. URL: https://github.com/zsdonghao/tensorlayer (visited on 12/02/2016).

[56] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras (visited on 12/02/2016).

[57] Theano. *Theano*. URL: http://deeplearning.net/software/theano/ (visited on 12/02/2016).

[58] ScikitLearn. *ScikitLearn*. URL: http://scikit-learn.org/stable/ (visited on 12/02/2016).

[59] SciPy. *SciPy*. URL: https://www.scipy.org/ (visited on 12/02/2016).

[60] NumPy. *NumPy*. URL: http://www.numpy.org/ (visited on 12/02/2016).

[61] Yangqing Jia and Evan Shelhamer. *Caffe*. URL: http://caffe.berkeleyvision.org/ (visited on 12/02/2016).

[62] D. Randall Wilson and Tony R. Martinez. "The general inefficiency of batch training for gradient descent learning". In: *Neural Networks* 16.10 (2003), pp. 1429–1451. ISSN: 08936080. DOI: 10.1016/S0893-6080(03)00138-2.

[63] James D. McCaffrey. *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training*. 2013. URL: https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/ (visited on 04/25/2017).

[64] Stanford University. *CS231N Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/neural-networks-3/ (visited on 05/01/2017).

[65] Michael Nielsen. *Neural Networks and Deep Learning Chapter 2*. 2016. URL: http://neuralnetworksanddeeplearning.com/chap2.html (visited on 05/01/2017).

[66] Francois Chollet. *Keras Applications Xception*. 2017. URL: https://github.com/fchollet/keras/blob/master/keras/applications/xception.py.