Classifying *Quick, Draw!*: Using Supervised Learning Algorithms for Sketch Classification

Joseph Adamson
ID: 2094607
MSc Computer Science

Supervisor: Mohan Sridharan

School of Computer Science
University of Birmingham
Birmingham, B15 2TT

Date: October 09, 2020

Abstract

The aim of the study is to develop and apply two supervised learning algorithms to classify sketch images for Google's open-source Quick, Draw! data set. The candidate algorithms - k-nearest neighbours and an artificial neural network - were chosen on the basis of their past effectiveness on sketch related classification tasks discussed in the literature. To meet the learning outcomes of the project each algorithm was implemented from scratch. The final classification accuracy achieved on a subset of the original data was 87.7% and 87.2% for k-nearest neighbour and neural network implementations respectively.

Contents

1 Introduction

Free-hand sketch is a universal form of communication that allows people to express ideas and document the world around them. Sketching requires no formal training or special equipment, it is not bound by age, language or culture. As a medium free-hand sketch can be highly descriptive despite its concise and abstract nature, making it useful in areas like communication and design. For this reason sketches have been widely studied in computer vision and pattern recognition [1]. The recent prevalence of touch screen devices and other drawing related applications have made sketch creation and collection easier, leading to the appearance of larger datasets like Sketchy [2] and Google Quick, Draw! [3]. This has resulted in a reappraisal of some classic sketch related research topics (e.g sketch recognition [4, 5, 6] and sketch based image retrieval [7]) as well as some new topics that have been made possible with recent developments in deep learning (e.g. sketch generation/synthesis [8]).

The aim of the work described in this project was to develop a pair of supervised learning algorithms that could used to build effective classification models for Google's Quick, Draw! dataset. The project was undertaken as a learning exercise, in order to build upon the knowledge gained from this year's MSc taught course as well as serve as an introduction to studies relating to sketch data and image recognition. To satisfy the learning outcomes of the project, it was decided that the algorithms chosen for the task and any other additional software needed was to be built entirely from scratch in Java. Results were used to further optimize the models and to draw conclusions about the properties of the dataset.

This report begins by formalizing the problem in more detail, outlining the project's motivations in the context of the related work before moving on to a detailed description of the algorithms used. **Section 3** will cover the design considerations for the project and general approach. **Section 4** will give a more detailed analysis of the data whilst **Section 5** will go over the most important aspects of the implementation. **Section 6** will cover experiments with the algorithms and the subsequent results. Lastly there will be appraisal of project in the lead up to the conclusion.

2 Problem Description and Background

2.1 Motivation

The idea for project was inspired by the "*Quick, Draw!* Doodle Recognition Challenge" hosted in 2018 on Kaggle, an online repository of user submitted datasets and a community for data scientists [9]. The objective was to build an effective classifier for Google's existing Quick, Draw! dataset. The project uses this same premise to develop and implement two machine learning algorithms to classify sketch images from the dataset. However, the goal for the project was not to build models that can compete with state of the art competition entries, instead it was done as a learning exercise. Therefore, it was not necessary to follow any of the competition's rules regarding the data or model evaluation.

The Quick, Draw! dataset consists of 50 million+ black and white doodle drawings across 345 different categories. The doodles have been collected from the players of *Quick, Draw!*, a *Pictionary* style browser game created by Google A.I. Labs. The game gives the player a specific category (animal, object etc.) and they have to draw it within a 20 second time limit. Whilst the player is drawing, the game's integrated neural network attempts to guess the category. All drawings, regardless of if they were successfully identified or not, are stored as part of the dataset under the category they were intended for. Submitted images are individually moderated but only for inappropriate content, meaning player errors, such as scribbled or unfinished doodles, are kept as part of the data. As such, part of the challenge for the project was to build models that could effectively learn from this very noisy dataset and be able to classify new instances successfully.

2.2 Related Work

2.2.1 Sketch Representation and Challenges

The draw of sketch classification in general lies in the unique challenges sketch images present as well as the variable ways they can be formatted and stored for computation. For example, a sketch can be saved as a sparse matrix or a simple black and white image that ignores its sparsity. Alternatively, since sketch drawing is a dynamic process, drawings can be suitably captured and represented as a sequence of pen coordinates. In this regard, sketches share some similarity with hand-written characters, although their abstract nature tends to set them apart from handwriting, which is usually subject to a specific set of rules [1].

Where a photograph represents a real-world object, a sketch is a subjective interpretation of the object. Human created sketches are far from consistent when compared to photographs; authors tend to use dramatic simplifications or exaggerations to the objects being represented, as well as different drawing styles and degrees of realism [6, 5]. Depending on the author's subjective opinion about what they consider as the most salient features of an object, different parts may or may not be included in a sketch. For example, an author may opt to draw an animal as a head with/without a body [1]. The depiction of an object can also be affected by an author's mental viewpoint or real-time perspective [1].

2.2.2 Sketch Datasets

Sketch datasets can be grouped as either single-modal or multi-modal in nature depending on the original task they were compiled for. Single-modal datasets consist only of sketches and are primarily used for recognition (e.g. Tu-Berlin [6], Quick, Draw! [9]). On the other hand multi-modal datasets support cross-media tasks such as retrieval/matching by providing sketches paired with other data (e.g photos, 3D models, text) [2, 10].

Past approaches for collection have included: (i) using crowd sourcing; with participants creating sketches in generous time frames [6], either from scratch or using visual prompts [2, 10]. (ii) extending existing datasets for matching tasks (QuickDraw-Extended [11]) (iii) collection via user participation in online games [9]. The Quick, Draw! dataset provides a significant increase in the volume offered as the first million-scale sketch dataset. It also offers the most diverse author-base, with submissions coming from *Quick, Draw!* players from across the globe for a more representative collection of sketch data than previous benchmarks [1].

2.2.3 Approaches to the Quick, Draw! Dataset

Existing studies on the Quick, Draw! have focused mainly on recognition and sketch generation, with the former being pursued in order to develop more robust models that can perform well on high-noise datasets [5]. In general, sketch recognition can be categorized into 'offline' and 'online' recognition settings [1] which are dictated by the format of the data. An offline system will take the whole sketch as input (either as a pixel matrix or a grey-scale image) and make a prediction based on the complete sketch. Online recognition systems use accumulated pen strokes to make a prediction on a sketch that may or may not be incomplete.

In contemporary research on sketch data offline approaches have been more common [1]. However, online classification methods have been used to great effect in real-time applications like drawing aids and online games [1] and in other research areas like sketch generation and sketch-photo retrieval [8, 11]. The Quick, Draw! dataset is available in multiple formats which has allowed for experiments in both offline and online recognition tasks.

Most state of the art recognition approaches to the Quick, Draw! dataset have utilized deep learning architectures, taking advantage of the dataset's multiple formats. Convolutional neural networks (CNN) have been used with great effect with pixel representations of the data [9, 5, 12] while recurrent neural networks (RNN) have achieved similar results with pen stroke representations [9, 8, 13]. However, the sheer amount of data available has also provided an opportunity for novel experiments with more traditional machine learning models. This has presented some interesting challenges for offline classification models using the dataset's sparse array and 28 x 28 grey scale image formats.

K-nearest neighbours (k-NN) has been used to classifiy subsets of the data with varying degrees of success [5] [14]. Guo et al. [5] retained a subset of 500,000 images total for each of the dataset's 345 categories in order to test a selection of modified k-NN classifiers. Their baseline implementation calculated a representation vector for each category based on the average of all feature vectors in a category, in order to reduce computational complexity of the model. This reduced each category of 1000 individual feature vectors to one. Unfortunately this approach was not effective for categories where representations were more diverse (e.g animals), achieving a low accuracy of 23.6% [5]. On the other hand Kradolfer [14] used a simple k-NN classifier implementation with greater success, achieving an accuracy of 82% on a much smaller subset of the original data (5 categories, 7500 per sample).

Attempts at offline recognition tasks have been made with other models, including random forest [14](81% accuracy) and artificial neural networks [14, 15] (85% accuracy) although again, significantly smaller subsets of the original data were used. Out of all the existing research covered for offline recognition experiments on the dataset, deep learning models been the most consistently successful, reaching an accuracy of 62.1% for Guo et al.'s subset and $\geq$ 90% on the smaller subsets mentioned previously [14, 15]. However, CNN classifiers were susceptible to some of the same problems that simpler models had trouble with. Both k-NN and CNN classifiers performed poorly on categories that were similar in composition and shape (i.e. hose and snake, apple and onion) failing to pick up the more nuanced details that set these objects apart [5].

Fernandez et al.'s online recognition approach to the Quick, Draw! dataset focused on a deeper analysis individual traits of the data [13]. The study looked at the quality of drawings from three categories (mountain, book and whale) from the original subset in an attempt to define categories by the level of complexity of representation. They hypothesized that their RNN would have more difficulty classifying sketch categories that contained greater variation in their interpretations (i.e. whale) as opposed to simpler objects (mountain, book). However, the results in this instance found that relatively simple categories with high variable details (book) actually had a lower classification rate [13].

2.2.4 Summary

Previous work regarding recognition has mainly focused on offline approaches which have been concerned with improvements in accuracy and analysis of the data's properties. The data itself has proven to be useable in classification tasks with a wide range supervised learning algorithms, with deep learning architectures primarily being used to attain competitive levels of accuracy. Due to the sheer amount of data available there has been a tendency in using subsets of the original dataset, although in the majority of cases an explicit rational has not not offered for the selection of specific categories [14, 5, 15]. Outside of more competitive recognition tasks, small scale experiments using less of the data could still be used to draw insights about the qualities of the dataset. Especially if the categories are chosen carefully.

Implementation of the chosen algorithms had to provide a non-trivial level of challenge to meet the learning objectives of the project. Out of all possible algorithms previously mentioned, k-nearest neighbours and an artificial neural network were selected. I already had a high level understanding of how both of these algorithms worked but I had no previous knowledge on their implementation. Their contrasting learning styles of both algorithms would also provide an interesting point of analysis for the project. So far we have tried to contextualize our approach to sketch classification with a discussion on related work on the subject, next we will discuss additional background for the algorithms that will be used for the project.

2.3 Supervised Learning

Machine learning tackles problem solving by 1) gathering a dataset and, 2) algorithmically building a statistical model on that dataset. A successful model is assumed to have 'learnt' rules from the data that are useful for solving the problem [16]. Machine learning approaches tend to fall into three categories depending on the available data and the problem to be solved; supervised learning, unsupervised learning and reinforcement learning. With approaches to sketch recognition tending to be based on supervised learning [1] this methodology was chosen for the project.

In supervised learning a dataset is a collection of labelled examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$. For each example $\mathbf{x}_i$ denotes a feature vector, where each dimension $x^{(j)}$ is an individual feature [16]. Features are independent variables that say something meaningful about an example; they can be numeric (e.g. age, salary), ordinal (e.g. low, medium, high) or categorical (model of a car) in nature. y_i denotes an example's label which can be a categorical or continuous value. The form of the label defines the type of problem to be solved; when predicting a categorical value the problem is classification, when predicting a continuous value the problem is regression.

The methodology of supervised learning is defined as follows; given a set of labeled examples $\{(\mathbf{x}_1, y_1)\ldots(\mathbf{x}_n, y_n)\}$ (the training set) predict the label of a new previously unseen data point $\mathbf{x}_{n+1}$. The underlying assumption is that if there is some relationship between $\mathbf{x}_i$ and y_i then this relationship should hold for new data [17].

2.3.1 Models

The goal of a supervised learning algorithm is to use the available data to produce a model which can reflect this relationship. A model can either consist of a mathematical function, capable of making predictions based on what it has been learned from the training data (i.e. parametric models; neural networks, support vector machines etc.), or it can be a method that makes predictions by comparing input to the training data (i.e. non-parametric models; k-nearest neighbours) [16].

2.3.2 Supervised Learning Steps

The general process of applying supervised learning to a real-world problem is outlined below by [17].

I Data collection and analysis: Data can be assembled from one or many sources and can be in a range of formats (i.e text, image, audio). The qualities of the data have to be considered in order to structure the data in an effective way.

II Data preprocessing: Collected data will need to be conditioned for computation; features are transformed into a format (i.e. a vector of numbers) that a learning algorithm will be able to process, labels too might have to be encoded numerically if they are categorical in nature. Any redundant/noisy data might need to be removed that could otherwise affect the potential performance of a model.

III Model development and training: The data is divided into three chunks (training, validation and test) and a learning algorithm is selected. The model is trained using the training set and its hyperparameters are tuned using the validation set to avoid overfitting.

IV Test and evaluate the model: After the training and development phase the model's performance is evaluated on the previously unseen data from the test set before being adapted for general use. Evaluation is based on prediction accuracy; put simply the percentage of correct predictions divided by total predictions.

V Model deployment: If the model achieves the desired accuracy on the test set it can be deployed for general use. However, if the results are not optimal the choice of learning algorithm or the current structure of the data might need to be reconsidered. In this way the workflow is an iterative process.

2.3.3 Generalization and Overfitting

A model will have good generalization if it is able to successfully identify patterns in the training set that can be applied to successfully classify unseen test data; poor generalization results in either underfitting or overfitting [16]. Underfitting is caused by a model that has a high bias, meaning the model is unable to predict the labels of the data it has been trained on effectively because it is too general (Figure 2.1 left). This may be an indication that the model used may be too simple for the data (using a linear model (i.e. linear regression) for non linear data) [16].

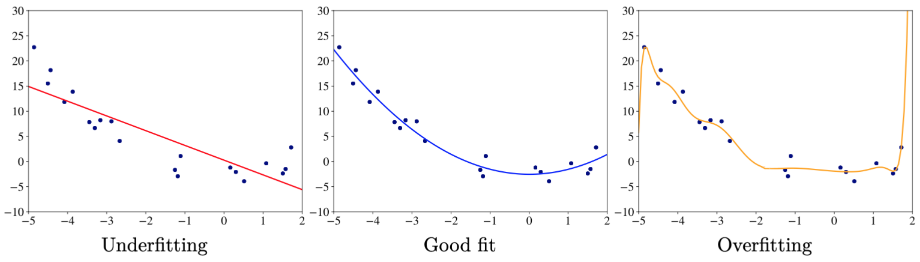


Figure 2.1: Training models: Underfitting (left), An appropriate fit (middle), Overfitting (right) [16]

Overfitting on the other hand predicts the training data very well but performs erratically with other data, reflecting a high variance. Such models are unable to generalize enough to adequately identify new data points as a result of learning the training data too well (Figure 4.3 right) [16]. Both overfitting and underfitting can be avoided through use of validation set. Validation serves as an intermediary phase that allows us to monitor performance. Results from the test and validation sets are compared, with the training process halted once the error starts to increase as a result of overfitting [18].

2.4 K-Nearest Neighbours

K-nearest neighbours (k-NN) is a simple, robust classifier that is often used as a benchmark for more complex classifiers such as support vector machines and artificial neural networks. It is a non-parametric algorithm, meaning that it makes no explicit assumptions about the relationship between x (features) and y (labels) in a dataset [17]. k-NN is an example of an instance based or 'lazy' learning algorithm because it delays the induction process until it is time to predict the label [19]. Concretely, this means a k-NN classifier does not explicitly 'learn' a model, instead, to classify a given test instance it 'memorizes' the training data which is subsequently used for prediction [17].

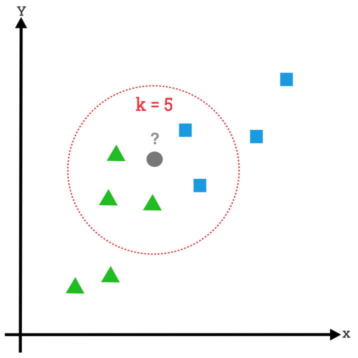


Figure 2.2: KNN example with k=5

This ‘knowledge’ can be modeled as an n-dimensional feature space. The example in fig 3.2 shows the distribution of labeled training examples in a 2 dimensional feature space. A new unknown data point is classified by identifying the labels of the k most ’similar’ training samples in the enclosing region. The prediction for the label of the unknown data point is then determined by a majority vote of the neighbouring training samples [19].

2.4.1 Distance Metric

’Similarity’ is defined according to the distance between a given training example and the unknown data point. There are a range of metrics that can be used to calculate this distance, with one of the most frequently used being the **Euclidean** distance [16]:

$$d(\mathbf{x}_i, \mathbf{x}_k) = \sqrt{\sum_{j=1}^{N} (x_i^{(j)} - x_k^{(j)})} \tag{2.1}$$

The Euclidean distance, d, between points is the length of a line segment that connects them in Euclidean space. For two feature vectors, $\mathbf{x}_i$ and $\mathbf{x}_k$, this difference is the summed difference of each corresponding feature $x^{(j)}$ [16]. Other popular distance metrics include Manhattan, Hamming and Mahalanobis although, for this study, the Euclidean distance was chosen due to its prevalence in the related work [5, 14].

The hyperparameter k is an important factor in the effectiveness of a k-NN classifier, as it dictates the number of training samples used for the vote or, more abstractly, the size of the decision boundary (see Figure 2.3). For binary problems, the value of k should be odd to avoid a tie but this rule is insufficient for multi-class classification problems (e.g we could choose k=5 and still end up with a 2:2:1 split) [17].

When the value of k=1 for example the classifier is less informed about the overall distribution of the training data, resulting in high accuracy but consequently fitting peculiarities in the data leading to poor generalization. A higher k involves more training examples in the majority vote, therefore making predictions more resilient to outliers for a more flexible model[17].

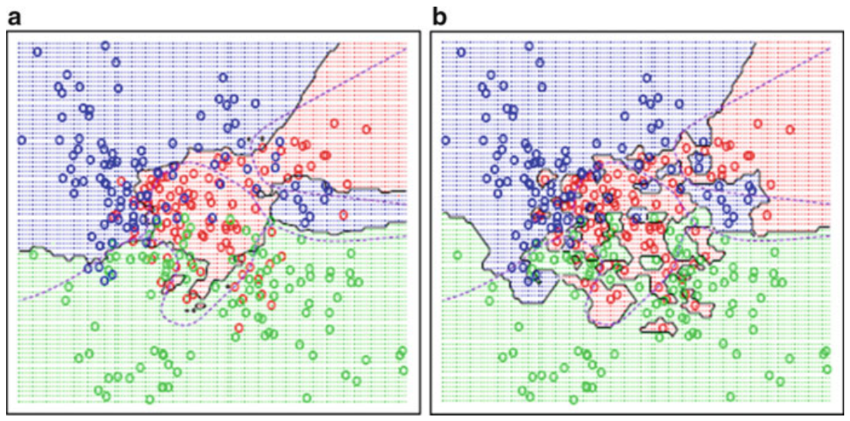


Figure 2.3: Decision boundaries (left) k=13, (right) k=1)[17]

However, picking too large a k will result in a classifier that is too general. Selecting an optimal value for k is thus a trade-off between rigidity (not overfitting) and flexibility.

2.5 Neural Networks

2.5.1 Architecture

Artificial neural networks take their inspiration from the structure and behaviour of the human brain, mathematically modelling the relationship between a set on inputs and outputs [19]. Neural networks are organized into stacked layers where the output of one layer becomes the input to the proceeding layer. This type of structure allows the network to breakdown data in order to extract useful information for problem solving [20].

Each layer of a neural network is made up of artificial neurons, logistical units that mimic the behaviour of biological neurons. As shown figure 2.3 a neuron performs a summation operation with its weights, w_i, and the output of the previous layer, a_i. A bias value b_i is then added which controls the sensitivity of the neuron [20].

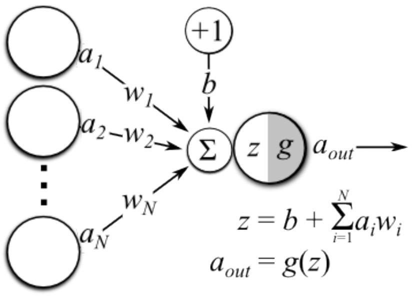


Figure 2.4: A single artificial neuron [21]

The summation of the neuron, z, is subsequently fed into an activation function g to give the final output [17]. The activation function of a neuron plays an important role in adding a non-linearity to result to indicate if the neuron has been triggered or not [19], this final output is then passed to the next layer of the network. There are range of activation functions that can be used in neural networks and the choice of function can have an influence on how a network behaves. The network created for this study will implementing the **Sigmoid** function, described in the following paragraph.

The Sigmoid function is a smoothed-out step function used to compress the output of a neuron between 0 and 1. Due to its asymptotic nature, when the value of z is large and positive the output from a Sigmoid neuron is approximately 1, whereas with negative values this output is closer to 0 [20]. The output of a Sigmoid neuron is useful for classification purposes, as the output value can be interpreted as a simple probability [20].

$$\sigma(z) = \frac{1}{1 + e_{-z}} \tag{2.2}$$

Simple networks have three types of layers. The leftmost layer is the **input later** which receives the raw data to be fed through the network's **hidden layers**, named as such because their inputs and outputs are abstracted away by the structure of the network. As data is fed through the network each layer performs transformations on its input to create a more selective interpretation of the data. Input is fed forward through the network in this fashion until it reaches the final layer; the **output layer** [16]. The output of a network can vary depending on the task being performed; in the context of classification this could be a class prediction.

For a neural network with two layers the output for the first layer can be given by

$$\mathbf{h} = \sigma(\mathbf{w}^{(1)T}\mathbf{x} + b^{(1)}) \tag{2.3}$$

Where $\mathbf{h}$ is the output, σ the Sigmoid activation function, $\mathbf{w}^{(1)}$ is a vector of weights for the layer, $\mathbf{x}$ an input vector and $b^{(1)}$ denotes the bias for the layer. These variables can be depicted in matrix notation, helping to communicate a neural network as series of matrix operations

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_j \end{bmatrix}, \ \mathbf{w}^{(1)} = \begin{bmatrix} w^{(1)}_{1,1} & \cdots & w^{(1)}_{1,k} \\ \vdots & \vdots & \vdots \\ w^{(1)}_{j,1} & \cdots & w^{(1)}_{j,k} \end{bmatrix}, \ b_{(1)} = \begin{bmatrix} b^{(1)}_1 & \cdots & b^{(1}_k \end{bmatrix} \tag{2.4}$$

With the final output layer given as

$$\mathbf{o} = \sigma(\mathbf{w}^{(2)T}\mathbf{h} + b^{(1)}) \tag{2.5}$$

With $\mathbf{o}$ representing the outputs for the network, $\mathbf{h}$ the output from the previous layer and $b^{(2)}$ the bias. In matrix form

$$\mathbf{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_j \end{bmatrix}, \ \mathbf{w}^{(2)} = \begin{bmatrix} w^{(2)}_{1,1} \\ \vdots \\ w^{(2)}_{j,1} \end{bmatrix}, \ b_{(1)} = \begin{bmatrix} b^{(2)}_1 \end{bmatrix} \tag{2.6}$$

The network described is presented in the below figure

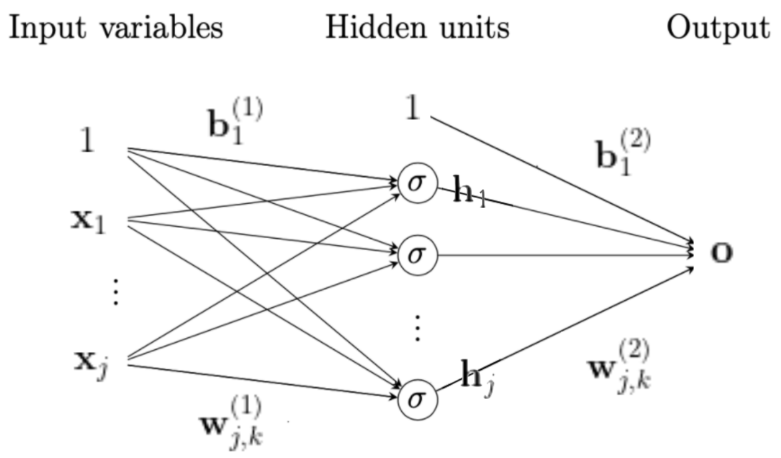


Figure 2.5: Two layer network, with one hidden layer

The example network in Figure 3.4 has a single hidden layer but networks can have multiple hidden layers depending on the complexity of the model.

2.5.2 Training Neural Networks

Training a neural network starts by initializing the weights and biases of the network randomly. From here on reference to a network's current set of weights and biases will be denoted with θ. At first, when given a set of inputs, a network will not be able to make predictions with any suitable accuracy. The objective of training is to incrementally improve the accuracy of these predictions over successive training cycles. More formally, in the context of classification, we want to tune the weights and biases for the network so that the output approximates the correct y_i for all training inputs $\mathbf{x}_i$. To quantify how well a network does this we define a cost function [20], denoted here simply as C.

For example, we could use the **mean square error**

$$\frac{1}{n}\sum_{i=0}(y_i - \hat{y}_i)^2 \tag{2.7}$$

which, given an input, calculates the squared difference between the prediction for the input, $\hat{y}$ and its label y. A large value for C is an indication of poor performance, the objective for each round of training is to reduce its value until $C \approx 0$. Other alternative cost functions exist depending of the type of problem, another popular choice is the **cross entropy** cost, which we will be looking at in more detail in the section describing the projects neural network implementation. Calculating the optimal parameters for a neural network can be framed as a search/optimization problem, normally tackled through the use of an optimization method called **stochastic gradient descent**. The job of the SGD is to explore a space of possible weights that may improve a model's ability to predict more effectively [20], this space is delineated by the network's cost function. With each iteration SGD takes a step, moving down the slope of the cost function towards a minimum.

The magnitude and the direction of each step is determined by the gradient of the cost function. More formally, the gradient can be thought of as a vector of partial derivatives whose elements contain the derivatives of C with respect θ, denoted here as ∇C [20].

Since the gradient points to the direction of the steepest ascent on the cost function the trick for constant minimization is to move in the opposite direction of the gradient [20]. The equations for updating the weights and biases using SGD in order to achieve this, using a single data point, are presented below.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla C(\theta, \mathbf{x}_i, y_i) \tag{2.8a}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla C(\theta, \mathbf{x}_i, y_i) \tag{2.8b}$$

Note that in the above notation C has been expanded to show its parameters. The vectors **w** and **b** refer globally to all the network's weights and biases respectively. The hyperparameter α represents the learning rate which dictates the size of each downwards step. Selecting the optimal learning rate requires some experimentation. A learning rate that is too large risks 'stepping over' the minimum. A learning rate too small will mean the algorithm will take a longer time to converge at the minimum, resulting in a longer training process which is more computationally expensive [20].

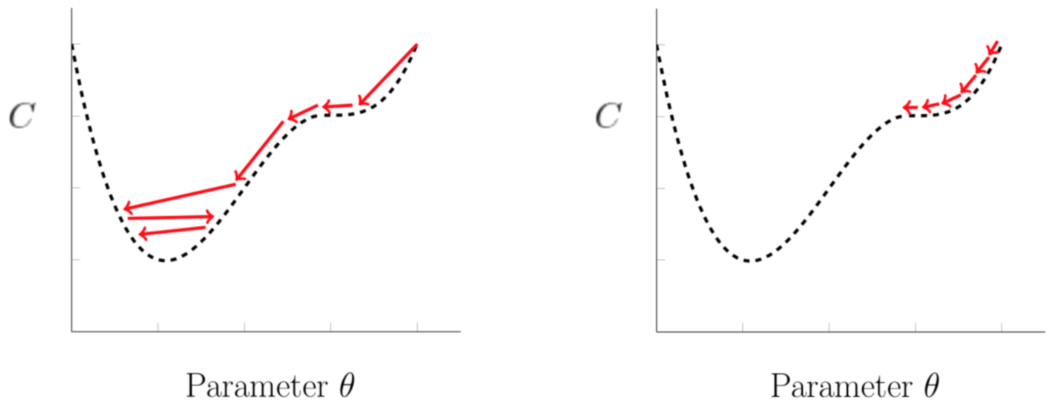


Figure 2.6: How the learning rate affects convergence; (left) too high learning rate, (right) too low learning rate [22]

The gradient vectors for each round of updates are calculated using the **backpropagation** algorithm, which uses the chain rule of differentiation to calculate the partial derivatives of the cost function with respect to each of the network's weights and biases [16]. An error is calculated using the cost function on the network's output, by moving backwards through the network we can understand how error varies with earlier weights and biases by repeatedly applying the chain rule. The reason for this backwards movement is a consequence of the fact that the initial error is a function of all outputs from the network [20]. The resulting gradient vectors are used for the network's updates, applied with some form of gradient descent.

In this section we have explored the approach and the algorithms that this project will implement. In the following section the design decisions and the data will be discussed in more detail.

3 Design

3.1 Tools

In the early stages of the project different existing machine learning libraries were considered for the creation for the project's models. However, such libraries offered high level implementations of the algorithms needed with minimal code use, conflicting with the project's goal of demonstrating further development of programming skills. Thus, it was decided to build all algorithms and supporting software from scratch. In order to provide additional support for the algorithms, further research was undertaken to develop a basic tool-set.

The **`DataPrep`** class was developed in order to help with any further data preprocessing. Its methods were used to manipulate the raw data and format it for use with the learning algorithms.

The **`Matrix`** class provided the matrix operations needed for the neural network implementation. The simple network model step-through in [20] and a linear algebra book by Jason Brownlee [23] provided the required background for the methods and operations that would be needed.

The **`Mertics`** class added simple tools for the analysis of model results; this consisted of some simple graphing methods built with the 'XChart'[24] graphing library.

3.2 Requirements

For organizational purposes a system was designed that would encapsulate the learning algorithms to be implemented, the tools in which they relied on and the experimental models that were to be developed. Furthermore, to get a clearer understanding of the functionality needed for the system, requirements were completed. Note, although these requirements were high-level in nature, they would provide a good guideline for what would be required for the testing of the software.

3.2.1 Functional

- The system must be able to construct a custom dataset using the original .npy category files contained in the data directory.
- The system must be able to load custom datasets for use with chosen learning algorithms.
- The system must, if necessary, allow for the further formatting of the data for use with both of the chosen learning algorithms.
- The system must be able to implement machine learning algorithms in order to build models that can classify the data.
- The system should throw an error if parameters for either learning algorithm are incorrect.
- The system must be able to allow learning algorithm parameters to be adjusted, where appropriate, for experimentation.

- The system must provide additional methods that underpin the functionality of the learning algorithms.
- The system must be able to display results and feedback in a human readable format (graphs, terminal output etc.)

3.2.2 Non-Functional

- The system should not be dependent on any external devices for its operation, it should be able to operate locally on the device it is being run on.
- Model building (the training phase) for the neural should not exceed 1 hour.
- Likewise, the testing phase k-nearest neighbours classifier should not exceed 6 minutes.

3.3 System Overview

Below details the system architecture used to incorporate the project's implementations of algorithms and the tools mentioned so far. The 'models' package contains two classification models created for the project's experiments with the data, discussed more thoroughly in the 'Experiments and Results' section of the report.

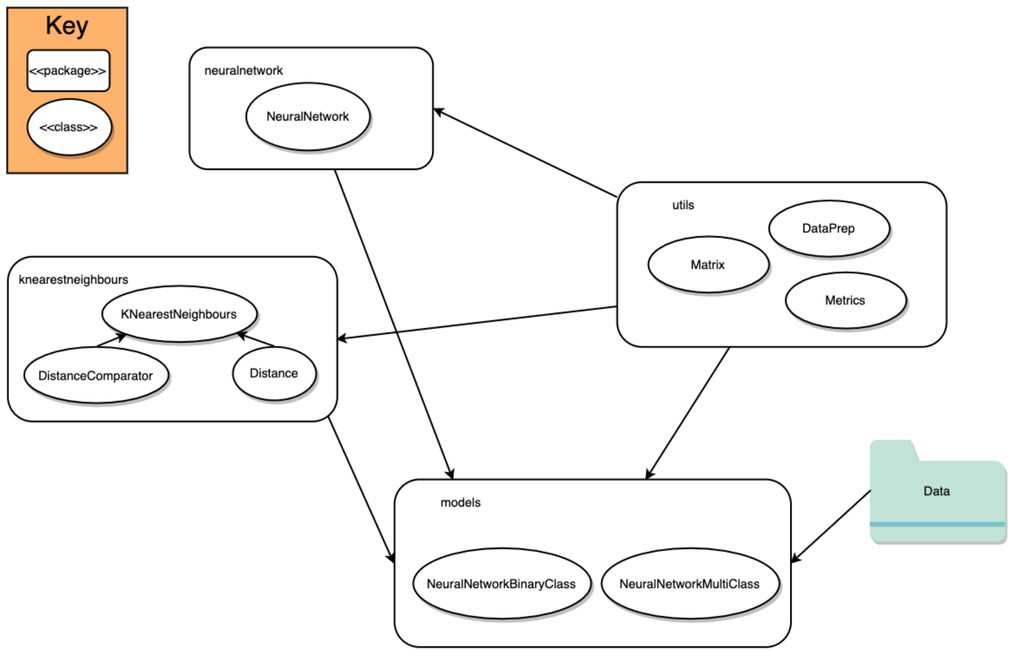


Figure 3.1: High-level system architecture, arrows indicate dependecies

4 Data

4.1 Description

The data was available from the Google *Quick, Draw!* Github page. Each drawing is represented in two distinct formats; either as a time series composed of (x, y) coordinates tracing the pen strokes that make up a sketch, or as a flattened one dimensional array of 784 pixel byte values between -127 and 127. The latter offline format was chosen due to its compatibility with the models considered for the project.

Each of the dataset's 345 categories had its own dedicated .npy file that could be downloaded via Google Cloud, with each category file containing roughly 150,000 samples. Each sketch is represented as a row in the larger .npy matrix. As the drawings were gathered from players in an unsupervised fashion they can be sorted into the following groups, reflecting the variance in each category [3]:

- **Correct** : The player drew the prompted category and it was successfully recognized by the network.
- **Correct but incomplete** : The player started to draw the prompted category but the network recognized the drawing before the player could finish.
- **Correct but not recognized** : The player drew the prompted category but the network failed to classify the drawing, often prompting the player to add more details or start again.
- **Incorrect** : The player could not finish the drawing leading to the image being scribbled out or the image depicted did not match the category.

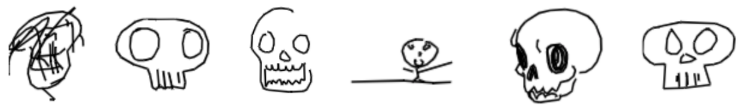


Figure 4.1: Variation in the skull category

4.2 Dataset Construction and Pre-processing

Due to the size of the original dataset a decision was made about how much of the original data should be used. Testing time constraints and computational resources for the project were taken into account and these were balanced against the complexity of the classification task and opportunities for analysis that more varied data would provide. In the end 7 .npy category files were selected from the database, these would be used to create two custom datasets for the project's experiments.

The first and smallest of these data sets, 'setA', would include samples from two category files triangle.npy and moon.npy. This set would be primarily used for integration testing purposes to check the initial performance of the algorithms with a simple binary classification task. The

Dataset	train.dat	validation.dat	test.dat	Categories
setA (8000 samples)	5600	1200	1200	(2) triangle, moon
setB (20,000 samples)	14000	3000	3000	(5) key, cat, axe star, calendar

composition of the second and larger of datasets, 'setB', used for multi-class classification experiments and the focus for the projects results, required further consideration.

As each .npy category file contained in excess of 150,000 samples it was decided to retain only 4000 samples per category for both custom datasets. The resulting sets were both split into training, validation and testing subsets according to the 70:15:15 ratio favoured in the related literature [16].

4.2.1 Selecting the Categories

As Fernadez-Fernadez et al.'s previous analysis on a subset of data had shown [13] categories could be informally grouped according to their complexity. For the report [13] complexity was defined by the following criteria; difficulty of representation (a mountain for example is easier to draw than a calculator) and variation of interpretation (an animal represented as a head/body). The idea for the composition of setB was to do the same, picking categories that could possibly highlight different qualities of the data for further analysis. Adding to this definition of complexity, categories were picked with the additional factors in mind; variation of orientation and level of detail.

"star" was chosen for its relative simplicity whereas "key" and "axe", due their varying orientation from sketch to sketch were considered a more complex categories. "cat" and "calendar" were picked as 'high level' complexity categories; "cat" for its variation of interpretation and calendar for its level of detail.

Note the the similarities in the shapes of 'key' (Figure 4.5) and 'axe' (Figure 4.6); both feature shaft-like shapes in their composition, choosing these two categories was purposeful. In the related work it was already noted [5] that classification performance on sketch categories that shared similar features was poor for multiple models. I wanted to test this rule for my smaller dataset, in order to see how well the classifiers did well with more nuanced details.

Figure 4.2: Two samples of 'cats' from the the project's subset converted into .jpg files of 28 x 28 pixels.

Figure 4.3: Two samples of 'stars' from the the project's subset converted into .jpg files of 28 x 28 pixels.

Figure 4.4: Two samples of 'calendars' from the the project's subset converted into .jpg files of 28 x 28 pixels.

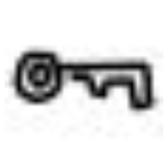


Figure 4.5: Two samples of 'keys' from the the project's subset converted into .jpg files of 28 x 28 pixels.

Figure 4.6: Two samples of 'axes' from the the project's subset converted into .jpg files of 28 x 28 pixels.

4.2.2 Organizing the Data

The same processes were used to construct both setA and setB. The .npy files for each category were stored in the project's data directory in the 'raw' subfolder. In order to retain 4000 samples for each category from each file methods from the **DataPrep** class were used. The packData method was used to iterate through 'raw', retrieving 4000 sketch samples (feature vectors) from each .npy file and converting them into java double array representations.

For simplicity, **normalization** and **labelling** were both handled together. Each individual pixel value in a featue vector was first converted to an unsigned double value between 0 and 255, then, using the following formula [16]

$$\bar{x}^i = \frac{x_i - min^i}{max^i - min^i}, \tag{4.1}$$

it was squashed the interval [0 - 1] using the minimum and maximum values for a pixel (0, 255). Normalization of pixel values in this way would allow for faster, less strenuous computation of the data during the training and testing phases [16].

Next, each sketch sample had to be labeled accordingly. Labels were generated by indexing the .npy files as they appeared in 'raw'. The order of the .npy files in 'raw' was such that each category, and therefore each sample in that category, was assigned a numerical label between 0 - 4, e.g setB: {key : 0, cat : 1, axe : 2, star : 3, calendar : 4}. The final result for each sketch sample was a double array of 785 elements; with element 0 - 784 consisting of normalized pixel values and the final 785th element a label value between 0 - 4, thus the array represented a paired data point $(\mathbf{x}_i, y_i)$, referred to from here on as a **data array**. The combined data arrays were all loaded into a 2D array which was then serialized into a byte stream file (.dat) ready for use. This process was repeated to create training, validation and test sets, sampling from a different place in the original .npy files each time.

In this section we have explored the data used for the project and discussed what qualities of the data might be reflected in resulting models. The next section will go into detail about the more important choices made for the project's implementation of the algorithms as well as how the project's software was tested.

5 Implementation and Testing

5.1 K-Nearest Neighbours

The main decision for the project's k-nearest neighbours implementation was which distance metric to use. It was shown in the previous review of existing work [5, 14], that the Euclidean distance had been used to obtain decent results on the *Quick, Draw!* dataset. Thus it became the metric of choice for the project's baseline implementation. This metric was utilized as follows:

$$(\mathbf{x}_n, \mathbf{x}_m) = \sqrt{\sum_{j=0}^{28*28-1} (x_n^{(j)} - x_m^{(j)})^2} \tag{5.1}$$

where,

$$x_n; n \in \ \textit{training set}$$
$$x_m; m \in \ \textit{testing set}$$

The Euclidean distance above is defined with the format of the data in mind. Every instance $\mathbf{x}$ is the first 784 elements of a data array representing a fixed size 28 x 28 pixel sketch. For each test instance, $\mathbf{x}_m$, we calculate its respective distance to **each** training instance, $\mathbf{x}_n$ in the training set. These distances subsequently sorted in ascending order by size, with the labels of first k distances factoring into to the classifier's prediction for the label of $\mathbf{x}_m$. To get the total accuracy of the model this operation had to be done for each instance in the testing set.

10-Fold Cross Validation

In order to find an optimal value for k, multiple values would have to be tested. To do this efficiently, without using the unseen data from the test set, 10-fold cross validation was used [25]. The training and validation sets would first be combined and then split into 10 folds (subsets) of equal size; the first fold being held back as an informal validation set with the remaining 9 folds used to train the data for one round.

Each round the error would be computed by testing the accuracy of the k-NN classifier on the validation fold. This process is repeated 10 times so that each fold is used as the validation set exactly once. To reduce the complexity of this process a distance table was calculated; a 2 dimensional matrix between testing and training data points i.e [17,000 * 3000] (for setB). The i^{th} row of the distance table stores the distances between the i^{th} test instance and all instances in the training set. The same table could be used to test different values for k instead of having to test different values iteratively, reducing time complexity at the expense of some memory use [25].

5.2 Neural Network

The neural network implementation was split into two classes **`NeuralNetwork`** and **`Layer`**. The former class provided all the underlying algorithms and methods (i.e. backpropagation, cost function and optimization method) with the latter implemented as a set of bias weight matrices that could be stacked to create the network.

To be able to handle the data arrays efficiently, further parsing was required upon initialization. This was done through the **DataPrep** method `vectorize`. A given data array was split into two column vectors; one containing 784 pixel values to be used as input for the network, the other, a binary column Matrix mapping of the label the same size as the outputs for the network created through one-hot encoding [16].

Sketch Category	Array Label	One-hot Vector
"key"	0	$[1,0,0,0,0]^T$
"cat"	1	$[0,1,0,0,0]^T$
...	...	...

Organizing the labels in this way would allow for network output or prediction for a given feature to be computed with its corresponding label; a necessity for the cost function.

Cross Entropy Cost Function

The cross entropy cost function was chosen for the neural network implementation, as it measures the performance of outputs in the [0, 1] interval produced by the Sigmoid function.

$$C(\theta, x, y) = \begin{cases} -ln(\hat{y}), & y = 1 \\ -ln(1-\hat{y}), & y = 0 \end{cases} \tag{5.2}$$

To understand how it works more succinctly consider an example using the first equation and the output of a single neuron, where the label y = 1 and $\hat{y} \approx 1$ for some input **x**. In this case the neuron is doing a good job at classifying the input and consequently the cost tends towards zero. However, in the same case, the closer $\hat{y}$ is to the wrong prediction, 0, the greater the cost with a tendency towards infinity for very bad predictions. This allows bad predictions to be penalised much more harshly, greatly aiding the learning process of the network [20]. This holds true for both equations which are combined into the following form for use in the network

$$C(\theta, x, y) = -\frac{1}{n}\sum[y_i \ln \hat{y} + (1-y)\ln(1-\hat{y})] \tag{5.3}$$

Optimization Method

In the related work stochastic gradient descent was used as an example for how a network learns. SGD is actually part of a larger family of gradient descent optimization methods that differ slightly in their approach. For the project's neural network implementation a different gradient descent variation, **mini-batch gradient descent**, (mBGD) was chosen. The data is split into mini-batches of equal size. When each mini-batch is processed we look at the cost function relative to all data points in the mini-batch and update the parameters for the network accordingly, taking a bigger 'step' towards the minimum. This allows for faster convergence when compared with SGD which uses on data point per update, making training times shorter [26]. The downside to choosing this type of optimizer is that it uses an additional hyperparameter, b, which determines the batch size. Finding the optimal batch size would require some experimentation.

The previous update rules from **Section 2** for gradient descent can be revised as follows

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla C(\theta, \mathbf{x}_{\{i:i+b\}}, y_{\{i:i+b\}}) \tag{5.4a}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla C(\theta, \mathbf{x}_{\{i:i+b\}}, y_{\{i:i+b\}}) \tag{5.4b}$$

where the gradient of the the cost function is now the cost relative to multiple data points in the batch, b.

5.3 Testing

Using the requirements for the project derived in **Section 2**, a test plan was created (see appendix). Some cases from the plan required more extensive unit testing, developing smaller cases for multiple methods that could be tested numerically, for example the case "System can apply numerous matrix arithmetic operations'.

However, at a higher level, larger more complex components (such as gradient descent and backpropagation algorithms) were more difficult to test in this way. To ensure the neural network and k-nearest neighbours classifiers behaved as expected required regular use of the IDE's debugger and manual console readings showing the results of matrix operations and the contents of arrays (see appendix). It was for this purpose that the test dataset, 'setA', was created. Using this dataset I was able to build two binary classification models (one for each classifier) with the objective identifying bugs a gauging the initial performance of the learning algorithms.

6 Experiments and Results

6.1 Evaluation Criteria

Confusion Matrix

After the training process it is important to be able to be able to evaluate and measure the performance of our models. A **confusion matrix** is a table that summarizes how successful a model has been at predicting examples belonging to different classes [16].

		Predicted	
		Positive	Negative
Actual	Positive	TP	FN
	Negative	FP	TN

Figure 6.1: Confusion Matrix Legend

The table in figure 6.1 shows the different label a binary class prediction can have, given the status between an actual value and a predicted value. The matrix indicates the number of instances that were truly classified positively or negatively, referred to as true positives (TP) and true negatives (TP). Likewise, it also indicates the number of cases that were falsely classified as positive or negative; false positives (FP) and false negatives (FN).

For multi-class classification problems each class represented will have its own row and column for a more fine grained breakdown of results. This can is invaluable for assessing a model's performance with individual classes and will help to identify mistake patterns [16].

Accuracy

Total model accuracy, which takes into account predictions over all classes, can be calculated simply dividing the number of correct predictions by the number of total predictions:

$$\frac{correct\ predictions}{all\ predictions} \tag{6.1}$$

Precision and recall give us two important metrics for how well a classifier performs with individual classes. Precision calculates the rate of actual positive classified instances over all positively predicted instances or put more simply, if a positive example is predicted, how often is it correct? [16]:

$$\frac{TP}{TP + FP} \tag{6.2}$$

Recall on the other hand represents the relationship between positively predicted instances and actual results, or how certain a model is of not missing any true positives:

$$\frac{TP}{TP + FN} \tag{6.3}$$

6.2 Experiment 1: k-NN for Multi-class classification

The following describes the optimization of the project's k-nn implementation and its results using the multi-class dataset ('setB') constructed in **section 4**.

The main focus with experiments with the k-NN implementation was to identify which k value would produce the optimal accuracy, without overfitting the training data. This was done before attempting to classify the data from the testing set. For this end the `kFoldCrossValidation` method was used, splitting the data into 10 folds and testing values for k in the range [1, 30] as described in **Section 5**. The results are presented below

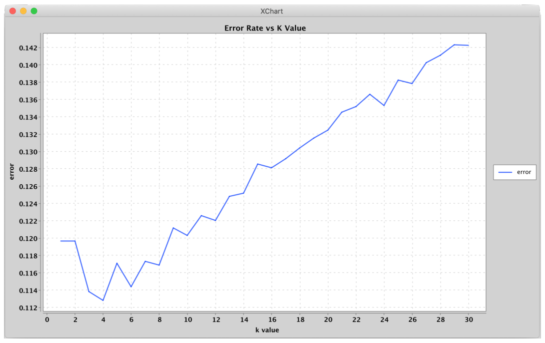


Figure 6.2: Cross validation results testing the values for k from 1 to 30

Figure 6.2 shows that k=4 corresponds to lowest validation error, indicating a sweet spot where the classifier generalises best without losing accuracy. The error creeps up quickly as the value of k becomes larger, showing the effectiveness of the classifier degrading as it becomes to general. This effect can be seen on a larger scale in figure 6.3 below.

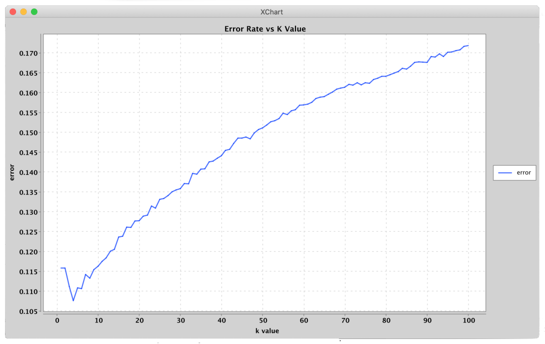


Figure 6.3: Cross validation results testing the values for k from 1 to 100

6.3 Experiment 2: Neural Network for Multi-Class classification

The following describes the experiments and subsequent optimization of the project's neural network implementation using the multi-class dataset ('setB') constructed in **section 4**.

6.3.1 Hyperpararmeter Tuning

So far we have only discussed hyperparameters in the context of the learning algorithms used for the project. More generally, hyperparameters are configuration variables that are external to a model (as opposed to the internal model parameters which are to be optimized), in this sense they can manually selected through an additional process [16] or by hand. For the project's network implementation this the latter approach was chosen. Although this was time consuming it allowed for further analysis that would be relevant to the learning outcomes of the report. As such, a set of rules were considered for how this was to be done, informed by the related reading [20, 16, 27]

- **Learning Rate**: Starting from 1 and then using increasing powers of n in 1 x 10^{-n},.]. This value was important as it would affect how the network's optimization method would navigate the cost function.
- **Batch Size**: Increasing powers of 2 starting with 24 (any smaller would just have the same learning effect as stochastic gradient descent).
- **Epochs**: Defined as one pass through the training data during the overall training phase; the more epochs the longer the training time.
- **Layers and neurons**: Adding more complexity to my network would likely cause longer processing times and overfitting. This would be done in small increments, paying attention to the trade-off between greater accuracy and the negative effects described previously [20].

To keep track of different configurations I used a google doc (see Appendix) to log my choices; this was done to avoid any repetition and zone in on the most effective setup of hyperparamers.

Initial Setup

Nielsen's [20] general advice on choosing network architecture informed the decisions made for project's implementation. Inputs, 784 for each pixel value, and outputs, 5, one for each class ('key', 'cat', 'star', 'calendar') were preset for the network, leaving only the hidden layers to be optimized. The initial setup is shown in Table 1.

Parameter	Specification
No. of neurons in the input layer	**784**
No. of neurons in the hidden layer	**10≥**
No. of neurons in the output layer.	**5**
Activation function	**Sigmoid (for all layers)**
Optimization method	**Mini-batch gradient descent**
Cost function	**Cross entropy**

Table 2: A selection of results for the configuration log showing different choices in architecture. Note that increasing complexity only improved accuracy up to a point; adding larger amounts of neurons and extra layers past a threshold (in blue) had no positive effect on output.

Hidden 1	Hidden 2	Accuracy
10	-	0.8303
50	-	0.8647
100	-	0.8733
300	-	0.8670
90	60	0.8663

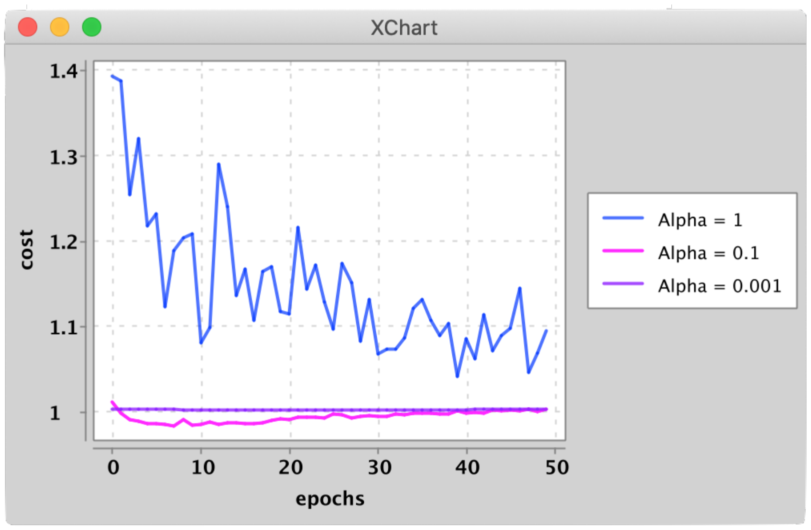


Figure 6.4

Figure 6.2 shows setup monitoring the effect of the learning rate on the behaviour of the cost function using the training data with three different learning rates.

Note the erratic behaviour when the $\alpha = 1$; as stated in related literature, a learning rate too large results in the the optimization method taking overly large steps, overshooting the minimum resulting in the spiking cost shown above. Alternatively, $\alpha = 0.001$ proved too small a value for our learning rate, slowing down the optimization process to produce virtually a imperceptible change in the cost function (at least, on this graph). Table 3: Futher learning rate optimization results using a range of learning rates with an optimized structure of **1 hidden layer with 90 neurons**, the best results are highlighted in blue

Learning rate	Accuracy
0.00001	0.4643
0.00005	0.6420
0.0001	0.6917
0.01	0.8750
0.1	0.8420
1	0.2000

For the largest value the model was not able to converge, however the second a third largest values achieved a reasonable accuracy. Table 4: Shows some experiments with the batch size, note larger batch sizes failed to converge at all having the same effect as a too high learning rate.

Batch Size	Accuracy
24	0.8683
32	0.872
64	0.8677
256	0.8670
4096	0.2000

Table 5: A range of epochs tested for the model with optimized structure and learning rate. Notice the difference in accuracy between 50 and 100 epochs; barley noticeable despite a doubling in training time.

Epochs	Accuracy
5	08117
15	0.8330
30	0.8593
50	0.8747
100	0.8745
200	0.8720

6.3.2 Network Regularization

After trying out different configurations the best accuracy so far was achieved with a network with one hidden layer of 100 neurons. However, after 100 epochs it was clear that the model was overfitting the training data.

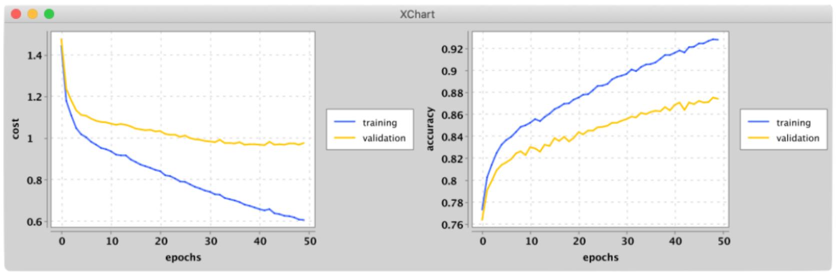


Figure 6.5: For both training and validation sets: (left) the cost function, (right) the accuracy

To combat this a technique called L2 regularization was applied to the network. By using L2 regularization we add an extra term to the cost function

$$C(\theta, x, y) = -\frac{1}{n}\sum[y_i \ln \hat{y} + (1-y)\ln(1-\hat{y})] + \frac{\lambda}{2n}\sum w^2 \tag{6.4}$$

This additional term (in red) is the sum of the squares of all the network's weights, scaled by the factor $\lambda/2n$, where λ is known as the regularization parameter [20]. Intuitively the regularization parameter forces the network to learn smaller weights, whilst penalizing larger weights, therefore reducing the overall complexity of the model and making it less prone to overfitting and peculiarities in the data.[20].

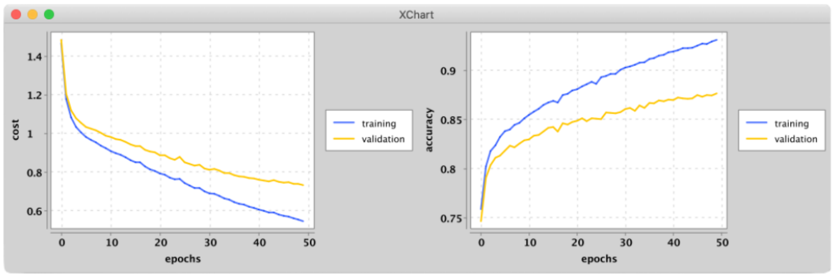
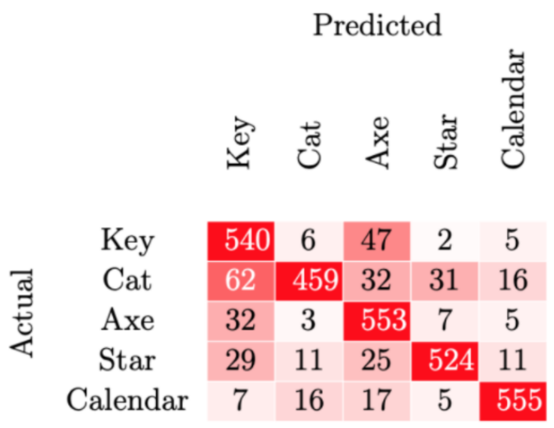


Figure 6.6: Slight reduction in cost function (as shown in figure 6.3) $\lambda = 0.005$

However, even after some experimentation, applying L2 regularization to the optimized network seemed to have little effect on the model results, only slightly reducing the difference between the cost on the training and validation sets but having a virtually negligible effect on accuracy.

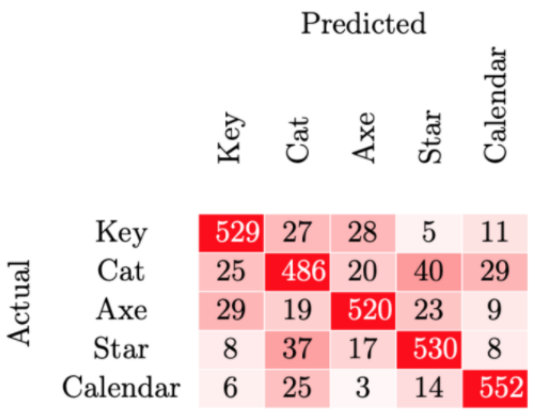
6.4 Results

The two confusion matrices in figure 6.7 and 6.8 break down the final classification results for the optimized k-NN and neural network implementations.



Actual \ Predicted	Key	Cat	Axe	Star	Calendar
Key	540	6	47	2	5
Cat	62	459	32	31	16
Axe	32	3	553	7	5
Star	29	11	25	524	11
Calendar	7	16	17	5	555

(a) **K Nearest Neighbours**



Actual \ Predicted	Key	Cat	Axe	Star	Calendar
Key	529	27	28	5	11
Cat	25	486	20	40	29
Axe	29	19	520	23	9
Star	8	37	17	530	8
Calendar	6	25	3	14	552

(b) **Neural Network**

Figure 6.7: Final results

In the comparison in Figure 6.7, the k-NN implementation predicted categories more consistently, albeit with more prominent mistake patterns. As hypothesized in the original analysis of the data, the similarities in the shape of 'axe' and 'key' categories caused problems for both classifiers, although more so for the k-NN classifier, resulting in its most notable errors. Both classifiers struggled with the 'cat' category which was frequently confused with 'key', 'axe' and 'star', indicating that the variation of interpretation that this category evinced was significantly harder to learn. The neural network had a more varied distribution of false predictions showing that it did not generalize sketch features as well as its counterpart; this could have been due to a lack of effective training or complexity [20].

	Precision	Recall
Key	0.8060	0.9000
Cat	0.9273	0.7650
Axe	0.8205	0.9375
Star	0.9209	0.8733
Calendar	0.9375	0.9250

	Precision	Recall
Key	0.8861	0.8817
Cat	0.8182	0.8182
Axe	0.8844	0.8667
Star	0.8660	0.8833
Calendar	0.9064	0.9200

Measures for the final models: (left) k-NN, (right) neural network

Additional class performance measures showed that both classifiers scored well on precision and recall for most classes with one notable exception being the 'cat' category. The k-NN classifier's ratio of actual correct predictions for 'cat' sketches was considerably lower than other classes leading to a noticeable drop in recall (0.765). Overall both models performed reasonably well with the data with the k-NN classifier just outperforming the neural network.

Classifier	Accuracy
k-nearest neighbours	**0.8770**
neural network	0.8723

7 Discussion and Conclusion

Overall the project was a relative success, as both algorithms had been implemented to successfully classify categories chosen from the dataset, with comparable levels of accuracy to the ones from the related literature concerning the same algorithms. Despite its comparative simplicity the k-NN classifier out-performed the neural network. However, due to the fact that only a small amount of the original data was used, the scalability of this performance wasn't fully taken into account. The neural network took longer to train but once this phase was over it could function as a fast reliable classifier. Prediction times for the k-NN implementation on the other hand were less than ideal, only increasing with the size of the data. Another significant difference between both algorithms was the optimization process; tuning the hyperparameters of the neural network required significant time. The result of training the network was affected by a degree of randomness, a realization I only came to later in the optimization process. If I were to repeat this process I would make sure to test the same configurations multiple times, averaging the results in order to minimize this factor of randomness. Conversely more efficient hyperparameter tuning methods such as random search and grid search could be explored [16].

Predictions made about the qualities of the data were mostly correct although my approach to categorizing the data by level of complexity was highly subjective. This process could have been improved by a more thorough analysis of the data, looking at more existing work on the dataset itself. A particular pain point for the project overall was the dual approach; whilst I certainly learned a lot from implementing the project's algorithms there was an inherent trade off with the development of the software and the experiments with the data as the robustness of my software was a concern during the experiments.

It was because of this that future approaches considered for this study fall into two categories. On one hand, pursuing further development of the software, both of the algorithms implemented here have significant room for improvement, with the network possibly being extended to accommodate different optimization methods and activation functions and the k-NN implementation being tested with different distance metrics. On the other hand a more in depth experimentation with the data, using both online and offline formats and the CNN and RNN architectures currently used in state of the art sketch classification would be a natural extension to the studies undertaken here.

The aim of the work in this project was to develop a pair of supervised learning algorithms that could used to build effective classification models for Google's Quick, Draw! dataset. By implementing these algorithms from scratch this project has satisfied its projected learning outcomes, whilst providing a solid entry into studies regarding sketch recognition and supervised learning.

8 References

[1] P. Xu, T. Hospedales, Q. Yin, Y. Song, T. Xiang, and L. Wang, "Deep learning for free-hand sketch: A survey and a toolbox," *arXiv preprint arXiv:2001.02600*, 2020.

[2] P. Sangkloy, N. Burnell, C. Ham, and J. Hays, "The sketchy database: Learning to retrieve badly drawn bunnies," *ACM Transactions on Graphics (proceedings of SIGGRAPH)*, 2016.

[3] "Google A.I. Blog." "https://ai.googleblog.com/2018/09/introducing-kaggle-quick-draw-doodle.html", 2018.

[4] Y. Yang and T. Hospedales, "Deep Neural Network For Sketch Recognition." arXiv:1501.07873, 1 2015.

[5] K. Guo, J. WoMa, and E. Xu, "'Quick, Draw! Doodle Recognition'." "http://cs229.stanford.edu/proj2018/report/98.pdf", 2018.

[6] M. Eitz, J. Hays, and M. Alexa, "How do humans sketch objects?," *ACM Trans. Graph. (Proc. SIGGRAPH)*, vol. 31, no. 4, pp. 44:1–44:10, 2012.

[7] T. B. M. Eitz, K. Hildebrand and M. Alexa, ""sketch-based image retrieval: Benchmark and bag-of-features descriptors"," *IEEE Transactions on Visualization and Computer Graphics, vol. 17*, no. 11, pp. 1624–1636, 2011.

[8] D. Ha and D. Eck, "A neural representation of sketch drawings," *arXiv preprint arXiv:1704.03477*, 2017.

[9] "Quick, Draw! Doodle Recognition Challenge." "https://www.kaggle.com/c/quickdraw-doodle-recognition/", 2018.

[10] Q. Yu, F. Liu, Y.-Z. Song, T. Xiang, T. M. Hospedales, and C.-C. Loy, "Sketch me that shoe," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 799–807, 2016.

[11] S. Dey, P. Riba, A. Dutta, J. Llados, and Y.-Z. Song, "Doodle to search: Practical zero-shot sketch-based image retrieval," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2179–2188, 2019.

[12] A. T. Kabakus, "A novel sketch recognition model based on convolutional neural networks," in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pp. 1–6, 2020.

[13] R. Fernandez-Fernandez, J. G. Victores, D. Estevez, and C. Balaguer, "Quick, Stat!: A Statistical Analysis of the Quick, Draw! Dataset." arXiv:1907.06417, 7 2019.

[14] D. Kradolfer, "'Quick, Draw!' - Classifying Drawings with Python." "https://www.datacareer.de/blog/quick-draw-classifying-drawings-with-python/", 2018.

[15] M. Alonso, M. Busquets, P. Palau, and C. Pitarque, "2018-dlai-team10: Deep Learning for Artificial Intelligence Project." "https://telecombcn-dl.github.io/2018-dlai-team10/", 2018.

[16] A. Burkov, *The hundred-page machine learning book*, vol. 1. Andriy Burkov Quebec City, Can., 2019.

[17] G. Dougherty, *Pattern Recognition and Classification: An Introduction*, pp. 1–3, 17, 121. Springer, 2012.

[18] B. W. Y. Michael W. Berry, Azlinah Mohamed, *Supervised and Unsupervised Learning for Data Science*, p. 146. Springer, 2019.

[19] P. Dangeti, *"Statistics for Machine Learning: Build supervised, unsupervised and reinforcement models using both Python and R"*, pp. 187, 236, 241. Packt, 2017.

[20] M. A. Nielsen, *"Neural Networks and Deep Learning"*, pp. 5, 9, 16, 36, 42. Determination Press, 2015.

[21] "Everything you need to know about neural networks." https://hackernoon.com/everything-you-need-to-know-about-neural-networks-8988c3ee4491. Accessed: 2020-08-30.

[22] "Deep learning." https://www.math.purdue.edu/~nwinovic/deep_learning_optimization.html. Accessed: 2020-08-30.

[23] J. Brownlee, *Basics of Linear Algebra for Machine Learning: Discover the Mathematical Language of Data in Python*, vol. 1. Machine Learning Mastery., 2018.

[24] "https://knowm.org/open-source/xchart/".

[25] D. Grover and B. Toghi, "'MNIST Dataset Classification Utilizing k-NN Classifier with Modified Sliding-window Metric'." "arXiv:1809.06846v4", 2019.

[26] V. Keselj, "Speech and language processing daniel jurafsky and james h. martin (stanford university and university of colorado at boulder) pearson prentice hall," 2009.

[27] "Deep learning rules of thumb." https://jeffmacaluso.github.io/post/DeepLearningRulesOfThumb/. Accessed: 2020-08-30.

9 Appendix

9.1 Test Plan

Test	
System can extract data from native .npy files and converts them into java double arrays.	Yes
System can load .dat files from a specified directory.	Yes
System can convert .dat files into matrices to use with the neural network classifier.	Yes
System can split and append java double arrays whilst preserving their content.	Yes
System can encode integers into binary column vectors.	Yes
System can apply numerous matrix arithmetic operations.	Yes
System can perform operations that allow for the manipulation of matrices.	Yes
k-nn implementation can compute the Euclidean distance between two data arrays.	Yes
k-nn implementation can sort and compare double values correctly.	Yes
k-nn implementation can compute majority vote given a set of integer labels.	Yes
System's classifiers can compute the accuracy of a model using predictions and labels for the given data.	Yes
System allows for the creation of neural networks of variable length.	Yes
NN layers hold a pair of matrices whose contents can be changed.	Yes
NN implementation can feed-forward inputs using the correct matrix operations.	Yes
Neural network implementation can compute a cost function using its output and a given label.	Yes
NN implementation can updates its parameters using m-BGD.	Yes
m-BGD implementation can split a given dataset into batches of equal size.	Yes
NN implementation can compute the gradient needed for parameter updates using backpropagation algorithm.	Yes
System can display NN metrics for training/validation phase.	Yes
System can formulate results for both classifiers in a confusion matrix.	Yes

9.2 Neural Network Configuration Log

`https://docs.google.com/spreadsheets/d/1faLxu3WKfPRfbngdtjgy5v6gEsJE97l9XQbK6aug4Ik`
`edit?usp=sharing`

9.3 Git Repository

Address of git repository:

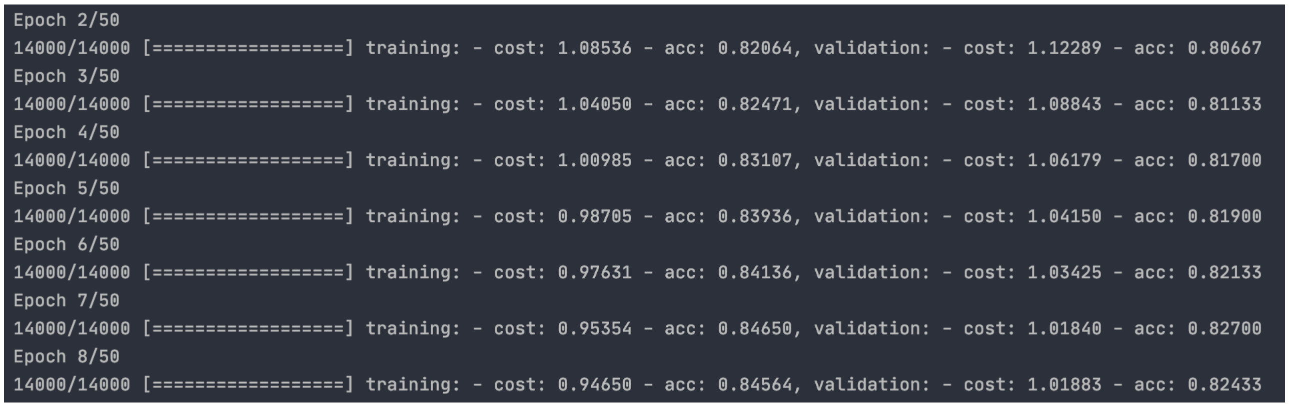
https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2019/jja907

9.4 Running the software

Once downloaded the final program can be run through the commandline with the following steps:

- From the root project directory (jja907) you can compile the models and their dependencies with the following command: `javac -cp "src:lib/*" -d out src/*/*.java`
- From there the pre-optimized neural network can be run with : `java -cp "out:lib/*" models.NeuralNetworkMultiClass`
- and the K-NN model with: `java -cp "out:lib/*" models.KNNMultiClass`

9.5 Network Training Screenshot



```
Epoch 2/50
14000/14000 [==================] training: - cost: 1.08536 - acc: 0.82064, validation: - cost: 1.12289 - acc: 0.80667
Epoch 3/50
14000/14000 [==================] training: - cost: 1.04050 - acc: 0.82471, validation: - cost: 1.08843 - acc: 0.81133
Epoch 4/50
14000/14000 [==================] training: - cost: 1.00985 - acc: 0.83107, validation: - cost: 1.06179 - acc: 0.81700
Epoch 5/50
14000/14000 [==================] training: - cost: 0.98705 - acc: 0.83936, validation: - cost: 1.04150 - acc: 0.81900
Epoch 6/50
14000/14000 [==================] training: - cost: 0.97631 - acc: 0.84136, validation: - cost: 1.03425 - acc: 0.82133
Epoch 7/50
14000/14000 [==================] training: - cost: 0.95354 - acc: 0.84650, validation: - cost: 1.01840 - acc: 0.82700
Epoch 8/50
14000/14000 [==================] training: - cost: 0.94650 - acc: 0.84564, validation: - cost: 1.01883 - acc: 0.82433
```

Figure 9.1: Console Readout