Machine Learning of Fonts

Antanas Kascenas

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2017

Abstract

Font kerning is the adjustment of the spacing between specific pairs of characters that appear next to each other in text. Kerning is important for achieving visually pleasing displayed text. Due to the quadratic growth of possible character pairs, manually kerning a font takes time and effort.

This project explores the possibility of automatic kerning using machine learning. Existing automatic kerning tools were discussed but no implementations using machine learning were found. Kerning specification in font files and relation to other font metrics were researched. A dataset for supervised regression learning of the spacings between character pairs was constructed. A neural network model predicting the proper spacing between the glyph bounding boxes given the whitespace shape between the glyphs and the font tracking was developed.

The neural network model was evaluated on a test set as well as on standard fonts, both quantitatively and visually. The results beat the baseline performance and an existing implementation of automatic kerning in FontForge font toolset.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Iain Murray, for proposing an interesting project topic, helpful feedback, useful thoughts and ideas.

Secondly, I would also like to thank Justas Zemgulys for feedback on the visual quality of the intermediate results during the development of the project.

Table of Contents

Chapter 1

Introduction

One of the main ways to consume information is through reading printed or otherwise displayed text. It is accepted that the layout and other design aspects of the text content can affect engagement, focus, reading speed, eye strain, text comprehension and other characteristics [1, 2, 3].

One of the text design decisions is the choice of a typeface or a font. A typeface is a family of fonts containing different variations of a font such as bold, italic, condensed etc. (for the purposes of this project, no distinction between typeface and font is made). However, the process of designing a font itself has a lot of subtle decisions that can affect the final design quality.

The focus of this project is the kerning and letter spacing aspect of the font design process. Kerning and letter spacing both affect the intraword spacing. More specifically, letter spacing is the assignment of whitespace on the sides of every character. Kerning is the adjustment is spacing between specific pairs of characters.

Together, letter spacing and kerning are used to finely control the space between the letters in the text. A well designed font is more visually pleasing to read and does not attract unnecessary attention to the specifics of the spatial configuration of the letters in a word. There is research indicating that intraword and interword spacings can affect the reading speed and legibility [4, 5] as well as that proportional fonts can be read faster than monospaced fonts [6] (fonts where all characters take up the same amount of horizontal space).

The difficulty with producing well spaced fonts is that due to varied shapes of the glyphs it is not enough to set how much space there should be on either side of a glyph. A glyph is a shape that usually corresponds to a character. A character can be a combination of a few glyphs but in Latin letters (which are used in this project) the correspondence is one to one.

Placing equal amounts of whitespace between the bounding boxes of neighbouring glyphs does not produce text with spacing that is visually consistent. To obtain text where each glyph pair appears to be equally spaced, manual offsets need to be applied to some specific pairs. In other words, some pairs need to be kerned. The number

of possible glyph pairs in a font grows as a square of the glyphs defined in the font. Therefore, it quickly becomes infeasible to kern all the possible pairs manually. In practice, letter spacing (assigning whitespace to each side of every glyph) is used to get as close to a good result as possible and kerning is used to fix the worst spaced pairs. These usually include capital letter pairs such as “AV” or upper-case and lower-case combinations such as “To”.

The aim of this project is to get insight into whether it is possible to leverage already existing manually designed fonts (this project uses fonts collected in the Google Fonts repository [7]) with machine learning methods to create a process that would partly or fully automate the letter spacing and kerning stages of font design.

1.1 Main contributions

- Research was done on how fonts are implemented in the currently widely used TrueType and OpenType font file formats. It was investigated how the relevant glyph spacing information is represented in the mentioned font formats.
- Existing approaches to automatic font letter spacing and kerning were explored and discussed.
- A supervised regression learning problem was set up to predict the proper spacing of glyph pairs given the description of the whitespace between the glyph outlines.
- Tools were sought to handle fonts and allow the extraction of the spacing information. A process of extracting or changing spacing information in font files was implemented using FontForge [8]. Code was written to visually compare the text with original and changed spacings in a font.
- A dataset from popularly used existing fonts was constructed and features describing the whitespace designed.
- Baselines for predicting the glyph pair spacing were set and compared with a variety of basic machine learning regression methods.
- Problem description was modified to model the tracking of fonts (a constant spacing offset common to all the pairs)
- Fully connected and convolutional neural network models were implemented and tuned to achieve reasonable results on the validation set.
- The best model was evaluated on a test set as well as on fonts unseen during training time. The results were also visually compared to the baseline solution and an existing FontForge automatic kerning algorithm. The proposed model beat the baseline and the FontForge implementation both in the quantitative evaluation and visual comparisons in the vast majority of cases.

1.2 Structure of the report

Chapter 2 presents the relevant background information including the main specifications of spacing information in TrueType font file formats and the discussion on the existing automatic kerning and letter spacing solutions.

Chapter 3 describes the regression problem in more detail as well as presents the reasoning, design and the implementation details of constructing the dataset for the regression problem.

Chapter 4 details the initial experiments, baselines, basic methods and their results on the regression problem as well as the problems with the approach.

Chapter 5 presents the reasoning on modifying the problem and the design of the models in the following experiments.

Chapter 6 focuses on the design, implementation and results of fully connected and convolutional neural models as the part of the regression system.

Chapter 7 describes the evaluation done on the best model. Quantitative and visual comparisons are provided and discussed.

Chapter 8 summarises the project and provides possibilities for improvements and future work.

Chapter 9 presents the plan for the second part of the "Machine Learning of Fonts" project.

Chapter 2

Background

Font design is a multipart process and there are multiple software options available for working with various aspects of fonts. Most bigger toolsets such as FontForge or FontLab (both commercial and free) have at least basic options for automatic kerning and letter spacing. Since both processes affect the final horizontal glyph positioning, they are usually applied at the same time.

There are two types of automatic kerning that are usually used in software. Metric automatic kerning uses only the font and glyph metrics specified in the font file (glyph metrics are discussed in the next section) and usually some kind of an iterative algorithm to adjust the letter spacing until it is considered good enough by a metric of visual spacing consistency across all the pairs. The metric of the visual spacing consistency is the hardest part of the algorithm since it is hard to define how exactly the eye perceives the space between different pairs of glyphs. Optical automatic kerning works in a similar way however it takes into account the actual shapes of the glyphs and how they combine in pairs. It uses the glyph shape information to construct more sophisticated metrics of visual spacing consistency. Optical automatic kerning is a more powerful and complex technique. This project heavily relies on the information of glyph shapes so could be categorised as an approach for optical automatic kerning.

In the experience of some font designers, neither of these automatic kerning types are enough to produce a well kerned font [9]. Therefore, some of the existing automatic kerning solutions aim to be transparent as possible or have many parameters so that the designers themselves can tune the process in an understandable and controllable way. The result of the automatic kerning process is usually treated as a starting point for the designer manual kerning work rather than a finished work.

The next section describes the glyph metrics that are specified in TrueType and OpenType format font files and drive existing automatic kerning solutions. Further sections discuss a few existing automated kerning solutions and their working principles.

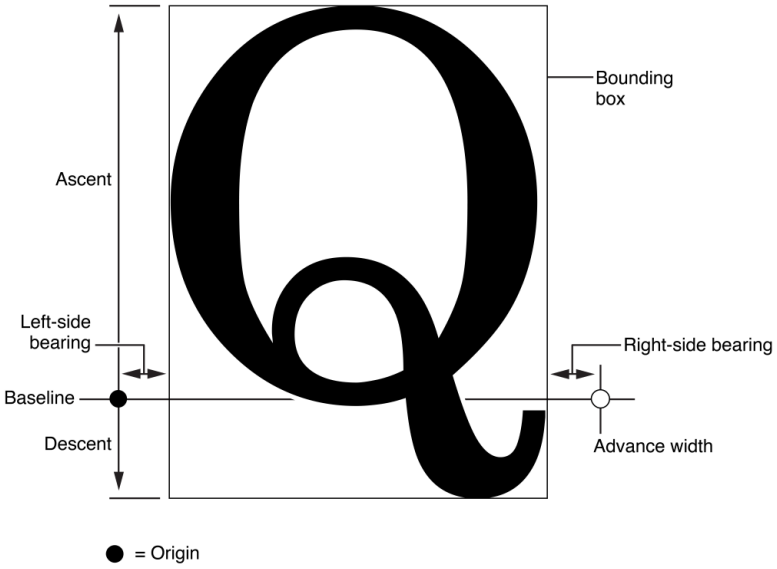

Figure 2.1: The basic glyph metrics. Figure taken from [10].

2.1 Glyph metrics

Glyph metrics contain data that influences how the glyph is rendered in the text in terms of the position of the neighbouring glyphs. The most important horizontal metrics are shown in Figure 2.1. While rendering text, glyphs are drawn sequentially one after another. Each glyph is drawn at the current marker position on the baseline. Glyph advance width defines how much horizontal space the glyph occupies in total and how much the marker position is increased on the baseline for drawing the next glyph. Left-side and right-side bearings define how much additional whitespace there is on either side of the glyph bounding box. Note that these can sometimes be negative to produce tighter fonts. In the case when the previous and current glyph pair has a specified kerning value, the marker position is moved to the right or to the left by the specified number of font units. One font unit is a font-specific fraction (usually 1/1000 or 1/2048) of font ‘em‘ (where ‘em‘’ is the current font size). The kerning value can be either positive or negative (the space between the glyphs is increased if the kerning value is positive and vice versa. Glyph ascent and descent control the vertical positioning of the glyph.

Kerning of a pair of glyphs is the process of finding a suitable marker offset after the first glyph in a pair before starting to draw the second glyph in a pair. Letter-spacing is a process of finding suitable left-side and right-side bearings.

2.2 Existing solutions

2.2.1 iKern

iKern [11] a commercial closed-source service that provides letter-fitting (kerning together with letter-spacing) services for wide variety of scripts (not only Latin). Even though the specifics of the system are not known, the author has explained the theory behind the system publicly [12]. The letter-fitting system is based on a mathematical model of the interaction of glyph shapes. It claims to have a number of parameters that control the automated system that can be adjusted iteratively after getting a fast visual response. The system generally models the whitespace so it works with sidebearings and kerning values at the same time. It claims to be modelling such high-level concepts as text homogeneity, coherence, compenetration, rhythm and distribution of density. These concepts are used to define how pleasing the glyph spacing is to the eye. The author of iKern claims that the system is continually being developed to be more general by taking font designer feedback and has kerned fonts from Latin, Cyrillic, Greek, Arabic and even some ancient scripts. Even though the system is considered to be quite impressive in relation to its competition [13], the author only regards the outputs as test cases that can bring new ideas for the designers. The system has been first publicly announced in 2004 and in development and use since then which testifies to the effectiveness of the service.

2.2.2 TypeFacet Autokern

TypeFacet Autokern [14] is a free and open source tool that tries to help font designers automate the process of letter spacing and kerning. It can perform kerning or letter spacing separately or do both simultaneously. The focus of the project is to make the automated kerning process as transparent and configurable to the font designer as possible and capable of quick iteration. The manual provided suggests guidelines for optimising the process. It involved setting of parameters such as the lower bound on the distance between the contours of two glyphs (by fitting pairs such as “TT”), the upper bound on the distance between the contours (by fitting a pairs such as “MM”), “intrusion” tolerance (how much glyphs can “intrude” into each others space), upper bound on the “overlap” of the horizontal extrema of the glyphs and many others. The process also focuses on heavy logging to improve the understanding of the process. The whole process is less automated (involves a lot of parameter tuning) and requires experience of working with the tool. The website claims the system to be actively developed but no contributions to the source code have been made since 2012.

2.2.3 FontForge

FontForge [8] is a popular and powerful free and open source font work software. It provides both automatic letter spacing and automatic kerning functionality as a part

of its toolkit. Automatic kerning is described here since it can achieve more specific spacing results.

The automatic kerning procedure takes in as a parameter a number that describes the desired optical separation of the glyphs. It uses a glyph separation procedure that guesses the optical distance between the pair of glyphs and calculates the appropriate kerning value to adjust the separation to the desired one. An assumption is made that the optical separation is linear in terms of the actual spacing (so that iteration is not required to find the correct adjustment).

The glyph separation procedure calculates the optical separation of glyphs by taking a weighted average of the horizontal distances between the glyph outlines in their common vertical space. This approach of using horizontal distances between the glyph outlines inspired the design of features describing the whitespace between glyphs in the dataset creation stage of the project presented in Chapter 3.

The details and the pseudo code of the automatic kerning algorithm are described in the deprecated version of the FontForge documentation [15](the functionality itself is not deprecated).

An interesting and powerful feature of this software is that the user can pass a custom routine to calculate the visual separation of the glyphs instead of the default implementation.

FontForge is actively developed since 2001 but its focus is much wider than just automatic kerning and letter spacing.

2.2.4 Other software

There are a few other software solutions to automatic kerning and letter spacing (some commercial and closed-source some open source but long abandoned). Some of them are listed here:

- DTL KernMaster [16] is a commercial font kerning and letter spacing solution. It is provided as a part of FontMaster font toolset developed by Dutch Type Library. The public specifics of the auto kerning algorithm are quite vague. However, it is clear that KernMaster converts glyphs into high resolution bitmaps and performs optical automatic kerning. It boasts very high quality results, however, no samples or other evaluation was found on the website.
- PubLink KernMaster [17] is a commercial automatic kerning solution. It claims to use artificial intelligence and space control algorithms. The essence of the algorithm is described as moving all the glyph pairs as close to each other as possible (so that they barely touch) and then applying tracking to the whole font (tracking is adding a constant spacing value to the spacings of all the pairs).
- FontLab's TypeTool [18]
- Font Combiner [19]

2.3 Discussion

The information on existing solutions is sparse and, in the case of commercial systems, is obscured by marketing promotion rather than actual details. Reviewing the information that is available on existing automatic kerning solutions indicates two different paths taken by most approaches. The first path being implementing a relatively simple algorithm with parameters that govern the execution of the algorithm (TypeFacet and FontForge solutions). The focus in this approach is on transparency and giving as much control as possible to the font designer. The second approach builds on some model of proper glyph spacing. The spacings of a new font are then adjusted to satisfy that spacing model (iKern, DTL KernMaster and PubLink KernMaster solutions).

The approach explored in this project differs from both of these paths since it aims to provide a fully or nearly fully automatic process without defining an explicit model of proper spacing (the “understanding” of proper spacings is learned from manually kerned fonts).

Chapter 3

Problem set up and dataset creation

3.1 The regression problem

The regression explored in the following chapters is that of predicting the normalised spacing (in font units normalised by the number of font units in one font ‘em‘) between the bounding boxes of a pair of glyphs (the spacing can be negative). The spacing is the sum of the right side bearing of the left glyph in the pair, the pair kerning value and the left side bearing of the right glyph in the pair.

The input to the regressors is the set of features describing the whitespace between the outlines of the glyph pair when the bounding boxes of the glyphs are touching. Figure 3.1 displays the glyph placement and the features that are described later in this chapter.

The spacing of each pair is predicted separately and when the spacings of all the required pairs in the font are known, a new font file is generated.

In order to experiment with machine learning regressors, a dataset of well kerned fonts was needed to use for supervised learning. There is currently no official or unofficial font dataset constructed for specifically kerning and letter spacing. A dataset of fonts with open licenses was constructed. The methods, choices, decisions and assumptions made while constructing the dataset are outlined in the sections below.

3.2 Source of the Font Files

There a quite a few websites online that aggregate font files in various formats. However, the quality of the design of such fonts is usually questionable. The variety of font formats would also significantly complicate the task of extracting meaningful kerning and spacing values. In addition to that, it was preferable to work with vector fonts instead of older style bitmap fonts since they are much more widely used and more universal. Google’s open source font files repository was chosen as the source of the font files for the dataset. It has over a thousand font files provided in TrueType and

OpenType font formats (for the purposes of this project, they are compatible) under a few different but open licenses (SIL Open Font License, v1.1 [20], Apache 2 [21] and Ubuntu Font License v1.0 [22]). TrueType is still the most popular font file format used for web and desktop fonts and has a number of software tools available to perform data extraction and font editing.

3.3 Software for Parsing Font Data

Python bindings for FontForge were chosen to perform the data manipulation. There are other lower-level alternatives (such as python's fontTools [23]) but the TrueType format is general enough to allow different ways of encoding the same information in different fonts. That means that it is very hard to write general software that performs actions on batches of fonts. Using lower-level tools proved to be infeasible since it would require writing custom code to extract the needed data from almost each different font. Putting adjusted values back into a font would also require significant manual effort.

FontForge is a powerful set of font software tools. However, despite its popularity and maturity, it was designed with a single font designer editing a single font at a time in mind. It also primarily focuses on providing a GUI interface instead of command line tools or importable modules. The documentation for the python bindings is sparse, outdated and sometimes requires knowledge of how font files work. It took considerable effort to figure out the code and API calls needed to extract the data described in the next section. The obstacles included: getting the kerning values in a consistent way (some fonts have more general positioning descriptions, others just define simple kerning values), some fonts can't be opened by FontForge or parsed, some fonts have glyphs named in non-standard ways that make accessing glyph data more difficult.

3.4 Extraction of the Spacing and Kerning Data

The fonts provided came in a variety of styles, compositions and degrees of exhaustiveness in terms of glyph definition. For consistency and to reduce the complexity of the data, only Latin character data was extracted from fonts.

Features for the training data needed to be chosen. The features needed to reflect the whitespace distribution between a glyph pair in order for machine learning models to be able to produce any kerning/spacing results. The most obvious choice is to provide the images of the glyphs and take the pixel values as the features. However, this would require the complex machine learning models to extract high level information regarding spacing, kerning and the glyph shapes. On the other end of the spectrum, it's possible to just take the side-bearing values, bounding box measurements and other glyph metrics and mostly ignore the relationship between the contours of the two glyphs in a kerning pair (similarly to automatic metric kerning mentioned in Chapter 2). This would work only to some extent because there would be not enough information in

the features to kern pairs with reasonable errors. A middle of the road solution was chosen.

3.5 Feature design

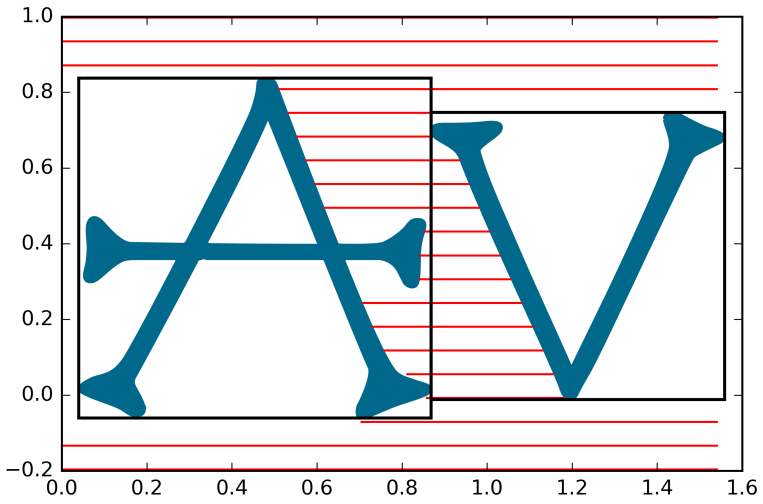

Figure 3.1: Constructing glyph spacing features. The features are the lengths of the red lines which are the distances between glyph outlines. The number of vertical sections (red lines) was reduced for clarity. The black rectangles are the glyph bounding boxes. The glyphs are from a Google font Snowburst One.

For each pair of glyphs, a number of steps were performed to construct the features. The side-bearings between the glyphs were not used as they contain a part of the spacing data (a model predicting just the sum of the left and right side bearings of the respective glyphs in a pair would be correct most of the time since relatively few pairs contain kerning values). Discarding the side bearings resulted in the bounding boxes of the glyph pair touching. The vertical space between the glyphs was divided into sections (variants included 5, 10, 30, 100, 200 sections). Two types of vertical splitting were tried (splitting the whole space between hard-coded upper and lower bounds regardless of the shape of the pair of glyphs and splitting just the common vertical region in each pair). The first type of splitting provided better results in the initial experiments and therefore was chosen for further work. After splitting the vertical space into horizontal sections, a point furthest to the right on the left glyph contour and a point furthers to the left on the right glyph contours were selected in each section and the distance between the points (scaled by the number of font units in font ‘em‘ to be consistent across fonts) was measured. This produced the same number of distinct distances as the number of sections that the vertical space was divided into. The distances together represent the shape of the whitespace between the pair of glyphs. Figure 3.1 displays a subset of features extracted for a glyph pair “AV”.

FontForge alone did not have all the functionality required the perform these calculations. All the glyphs from all the fonts were exported to the SVG files. This allowed the usage of a SVG handling library [24] to approximate the points on the contours of the glyphs. To choose the outermost point in each vertical section the following procedure was performed. Sufficiently many points over the whole glyph contour to reflect the shape of the whole glyph were sampled (2000 points), then the subset of these points that are in the required vertical section was taken and the point which is the most to the right (for the left glyph in a pair) or the most to the left (for the right glyph in a pair) was taken.

The label (target) for each pair of glyphs was calculated to be the sum of right side bearing of the left glyph in a pair, left side bearing of the right glyph in a pair and the kerning value if it exists. The resulting value was divided by the number of font units in font ‘em‘. This is essentially font unit invariant distance between the bounding boxes of the two glyphs.

The process of extracting the glyphs and calculating all the needed distances was relatively computationally expensive since there were over a million being processed. Naive implementation took around 8 hours to process all the glyphs. A faster implementation using bisection algorithms [25] on sorted lists as well as parallelisation on multiple CPU cores brought the time down to about 30 minutes.

3.6 Cleaning, Filtering and Further Processing

Google’s font repository is relatively well maintained and a good resource for fonts. However, not all the fonts provided in the repository are of equal quality. To produce good results through supervised learning it is essential that the training data is of good quality. An assumption was made that the most popularly downloaded fonts in the repository would be of the highest quality in terms of kerning decisions, thoroughness and completeness. Therefore, the training set was chosen to be constructed out of the 100 most popular typefaces (font families).

Access to Google font developer JSON API [26] was required to determine the font popularity order and metadata (such as font type and other fonts in the family). It was required to obtain a Google developer key to get access to the API.

Using the obtained font descriptions, monospaced fonts were skipped when collecting the most popular fonts. Monospaced fonts are specifically designed to have the same width for all the characters. They have no kerning at all and although the letter spacing is still important for a visually pleasing result, it is approached differently than in proportional fonts. Therefore, it does not make sense to include monospaced fonts in the training or testing data.

The final dataset resulted in data from 245 different fonts and 662171 glyph pairs (2704 pairs per font). A small fraction of glyphs that failed to parse was skipped. Each data point contained the target, 200 features and metadata (font and the glyph pair). 25000 pairs were kept for the test set while the rest was shared for training and validation.

Chapter 4

Baselines and Initial Experiments

The creation of the dataset described in the last chapter enables the exploration and experimentation of the data. The first section of this chapter describes some initial exploration of the glyph spacing data. The rest of the sections describe some of the baselines, basic machine learning models trained and the conclusions made.

4.1 Outlier Detection

Some data exploration was needed to check how clean the dataset is. Cleanliness, in this case, means how reasonable all the side-bearing and kerning values in the fonts are. It was needed to make sure that there are no extreme values that are obviously there by mistake and that could impede the training of models later on.

Outlier detection was the method chosen to detect any possible extreme values in the font files. Outlier detection algorithm EllipticEnvelope from scikit-learn [27] machine learning toolset for Python was applied to pairs across all fonts with defined Latin characters. The features used were the right side bearing of the left character in the pair and the left side bearing of the left character as well as the font unit.

The specific EllipticEnvelope algorithm was chosen because it was already readily implemented in scikit-learn and could be applied without a lot of additional set up and it generally works well in low dimensions.

4.1.1 Results

Getting the results required setting an appropriate contamination parameter for EllipticEnvelope (essentially the fraction of the fonts that should be considered outliers). The contamination parameter was set to 0.01 to obtain and visualise the pairs which the model considers to be the biggest outliers.

Random words from a standard English word list through python's NLTK [28] containing the required pairs were generated to visually inspect the results. Figure 4.1

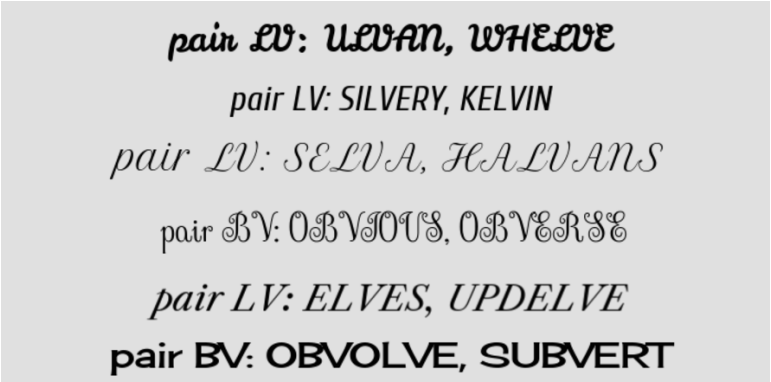

Figure 4.1: The biggest outliers found in glyph pairs “LV” and “BV”.

displays the glyph pairs considered to be outliers for pairs “LV” and “BV”.

The outliers found were not extreme or otherwise unlikely values except for the imperfect spacing in some cases. Therefore, no significant irregularities in the font files were found and it was moved on to training some basic models.

4.2 Regression models

In the second phase of the project, the dataset and its variations described in Chapter 3 were used to try and build a regression model. The targets for the model were the distances between pairs of glyphs (distance from bounding box to bounding box, the sum of both side bearings and kerning scaled by font units in font ‘em‘). The features used were subsets of varying size of the 200 horizontal distances described in Chapter 3.

The results were evaluated looking at the mean squared error (the loss function), mean absolute error (easier to interpret in terms of distances between glyphs and improvement) as well as visual results. Excluding the data set aside for testing (as described at the end of Chapter 3), 50000 data points were used for validation and the rest was used for training.

4.3 Linear Regression

Simple linear regression (minimising the mean squared error between the predicted and target glyph pair spacings) implemented in scikit-learn [29] was first tried on the dataset version with 200 horizontal features. This resulted in the errors shown in the first row of Table 4.1. Linear regression is the simplest model that has been tried and is considered to be the baseline result for this type of regression.

A number of approaches were tried to improve the baseline performance using linear regression.

4.3.1 Principal component analysis

The features are spatially related (sequentially close features are heavily dependent on each other). It means that there is at least some redundancy in the information provided by the features. One way to avoid the redundancy is to make the features linearly uncorrelated using a transformation such as PCA [30]. However, the results of running PCA with linear regression only provided very small improvements. It was also tried to reduce the number of principal components used to reduce the number of features overall. The feature reduction did not have a significant impact. Only when reduced down to 10 components it increased the errors slightly. The results of these experiments are shown in rows 2 to 4 in Table 4.1.

The reason for the lack of improvements is most likely the fact that there is a lot of training data. PCA in this case (especially when reducing the dimensionality of the features) acts as a regulariser. Regularisation is a technique to reduce overfitting. Overfitting is the model describing the noise in the data rather than the actual relationship between the targets and the features. Since there is enough data, the linear regression model generalises well and does not overfit. Therefore, PCA does not improve the performance. Training with other regularised linear regression variants (ridge and lasso regression) also did not improve performance which confirms the hypothesis that overfitting is not a problem (training and validation errors being very similar indicates that too).

4.3.2 Increasing the complexity of the model

Since the previous point proved that overfitting is not a problem, it makes sense to try and increase the complexity of the model.

The goal of increasing the complexity of the model is to try to extract more information from the features which the model can use to make better predictions and in that way improve the performance.

One way of making the linear regression model more complex is adding polynomial features. That is, adding new features that are multiplications of existing features. The problem with this is that the number of features increases exponentially with respect to the polynomial order. This explosion means that to use polynomial features we first need to reduce the dimensionality of existing features (it is not feasible to add polynomial features if we already have 200 features due to exponential explosion). Two approaches for reducing the dimensionality were taken. The first one approach involves reducing the number of features by uniformly sampling a subset of existing features. However, this means losing some information. The second approach involves adding polynomial features to the PCA components with the biggest explained variance. In theory, using the components with the biggest explained variance should means that less information should be lost.

For the first approach, the existing features were reduced to 20 features and quadratic features were added using scikit-learn's PolynomialFeatures class [31] resulting in

231 features in total. Adding polynomial features resulted in noticeable improvement shown in the row 5 of Table 4.1. Slightly different variantions (such as reducing feature number to 10 and adding cubic polynomial features instead) were tried as well and yielded similar or slightly worse results.

For the second approach, after applying PCA to the original features, 20 first components were taken. Quadratic features were added similarly to the first approach. Using PCA in this approach yielded significant improvements (presumably because less information was lost during the original feature reduction). Another variation of taking 10 PCA components and adding cubic features yielded similar results. The improvements are shown in rows 6 and 7 of Table 4.1.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
Linear regression 200 features	$1.9*10^{-3}$	$3.4*10^{-2}$	$1.9*10^{-3}$	$3.4*10^{-2}$	Baseline result
Linear regression 200 PCA features	$1.9*10^{-3}$	$3.4*10^{-3}$	$1.9*10^{-3}$	$3.4*10^{-2}$	
Linear regression 100 PCA features	$1.9*10^{-3}$	$3.4*10^{-2}$	$2.0*10^{-3}$	$3.4*10^{-2}$	
Linear regression 10 PCA features	$2.1*10^{-3}$	$3.5*10^{-2}$	$2.2*10^{-3}$	$3.5*10^{-2}$	
Polynomial regression 20 original features 231 features in total after adding quadratic features	$1.6*10^{-3}$	$3.1*10^{-2}$	$1.6*10^{-3}$	$3.1*10^{-2}$	8% validation MAE improvement over the baseline
Polynomial regression 20 PCA features 231 features in total after adding quadratic features	$1.4*10^{-3}$	$2.9*10^{-2}$	$1.4*10^{-3}$	$2.9*10^{-2}$	
Polynomial regression 10 PCA features 286 features in total after adding cubic features	$1.4*10^{-3}$	$2.9*10^{-2}$	$1.4*10^{-3}$	$2.9*10^{-2}$	15% validation MAE improvement over the baseline

Table 4.1: Initial results for linear and polynomial regression

4.3.3 Discussion

The baseline results provide a good starting point to measure improvements. The closeness of validation and training error show that the linear regression models might be underfitting (the models can't capture the underlying relationships between the features and the targets). Therefore, it is understandable why dimensionality reduction by PCA did not help. Underfitting also means that it is worth trying models that can extract more information from the features. The improved results from polynomial regression indicate that significant improvements could be achieved by combining features. It is very likely that more feature engineering would result in further improvements. A few of such approaches could be combining only spatially close features (would allow using higher order polynomials) or splitting the features into sections and adding

additional features that describe those sections (such as minimum and maximum in a section). However, feature engineering takes considerable effort and it is easier to switch and try more complex models to solve the problem of underfitting.

4.4 Decision trees and random forests

Decision trees and random forests are popular models for generic regression and classification. Decision trees and forests produce non-linear decision boundaries. Non-linear decision boundaries can be beneficial in the glyph pair spacing regression problem since the previous experiments indicated the need for more complexity in the model. Single decision trees are usually prone to overfitting (subjectively more than linear regression) unless some variance reduction techniques are used. Two ways of dealing with overfitting were tried in the experiments in this section. Firstly, the maximum depth of the decision tree was limited. Secondly, random forests were tried. Random forests average the predictions of multiple trees each of which is trained on a subset of the training data. This makes the overall predictions more robust.

4.4.1 Experiments

A single decision tree models with maximum depths of 5, 10 and 15 were tried. Random forest of 30 and 50 trees with each having maximum depth of 15 was tried. It is likely that increasing the number of trees in the random forest would increase the performance slightly but the improvements in performance had diminishing returns. The improvement likely would not be big so no bigger forests were trained. The resulting model performance is displayed in Table 4.2.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
Decision tree 5 max depth	$1.6*10^{-3}$	$3.2*10^{-2}$	$1.8*10^{-3}$	$3.1*10^{-2}$	7% validation MAE improvement over the linear regression baseline
Decision tree 10 max depth	$1.1*10^{-3}$	$2.4*10^{-2}$	$1.1*10^{-3}$	$2.4*10^{-2}$	
Decision tree 15 max depth	$6.1*10^{-4}$	$1.7*10^{-2}$	$7.4*10^{-4}$	$1.9*10^{-2}$	
Random forest 30 trees 15 max depth	$4.7*10^{-4}$	$1.5*10^{-2}$	$5.5*10^{-4}$	$1.6*10^{-2}$	
Random forest 50 trees 15 max depth	$4.6*10^{-4}$	$1.5*10^{-2}$	$5.5*10^{-4}$	$1.6*10^{-2}$	52% validation MAE improvement over the linear regression baseline

Table 4.2: Initial results of decision tree experiments

4.4.2 Discussion

Bigger decision trees and forests proved to perform significantly better than linear regression models. Similarly to more complex linear regression models, the improvements come from more complexity. More specifically, the ability of the trees to fit more complex decision boundaries and having enough data to avoid overfitting. However, since decision trees (and by extension, forests) split the data only on one feature at a time (on a single node), they combine features rather inefficiently (since the max depth is at most 15, each prediction is made only relying on 15 features at most). To get even better results, it would probably be useful to choose a slightly different feature representation or hand-craft some manually combined features. This would allow extracting the information of feature combinations easier. In general, both linear regression and decision tree experiments show that feature combination can bring significant improvements, therefore rather than trying to hand-craft features for decision trees, it was decided to try other models that deal with features combinations better.

4.5 Nearest Neighbours

The K Nearest neighbours is a powerful but fairly simple type of model. To predict a spacing for a glyph pair, k closest glyph pairs in the training set (in terms of some distance measure on the features) are found and their targets averaged. There are two main reasons for experimenting with KNN. Firstly, KNN models can be effective given enough data and the previous experiments indicate that we do have a lot. Secondly, KNN models do not make assumptions about the characteristics of the concepts behind the predicted relationship but can learn complex concepts by local approximation. In addition, it requires almost no additional set up so it is easy to plug the data into the model and see if it is possible to draw any meaningful conclusions from the results.

4.5.1 Experiments

The KNN implementation from scikit-learn [32] was used with the default parameters (most importantly, Euclidean distance measure) was used. KNNs with k = 1, 3 and 5 were tried. In addition, it was also tried to use KNNs with PCA features to reduce the effects of the curse of dimensionality [33] (in a high-dimensionality space, Euclidean distance might not be helpful). The results or the experiments are shown in the Table 4.3.

4.5.2 Discussion

The results from the k-NN experiments show k-NN models performing better than the best random forest models from the previous section despite high feature dimensionality. The relatively good results indicate that the features lie in some manifolds of the whole feature space (otherwise, the curse of dimensionality of the 200 features should

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
k-NN k = 1	0	0	$5.3*10^{-4}$	$1.3*10^{-2}$	
k-NN k = 3	$2.2*10^{-4}$	$9.0*10^{-3}$	$4.9*10^{-4}$	$1.4*10^{-2}$	
k-NN k=5	$3.3*10^{-4}$	$1.2*10^{-2}$	$5.3*10^{-4}$	$1.6*10^{-2}$	
k-NN k = 3 20 PCA features	$2.3*10^{-4}$	$9.4*10^{-3}$	$5.3*10^{-4}$	$1.5*10^{-2}$	
k-NN k = 3 30 PCA features	$2.2*10^{-4}$	$9.1*10^{-3}$	$5.0*10^{-4}$	$1.4*10^{-2}$	
k-NN k = 5 30 PCA features	$3.4*10^{-4}$	$1.3*10^{-2}$	$5.3*10^{-4}$	$1.2*10^{-2}$	64% validation MAE improvement over the linear regression baseline

Table 4.3: Initial results of k-nearest neighbours experiments

affect the results more). It is likely that it is possible to increase the performance of the model by customising the feature representation and the distance measure used. However, there is a number of issues with the k-NN approach to glyph spacing regression. The good model performance is likely a result of using the same fonts in both training and validation sets (the model might not generalise well on unseen fonts). Ultimately, predicting on unknown fonts is the goal and it is particularly hard to extend the model to provide some font information that glyph pairs share (discussed in more detail in the next chapter). In addition, initial experiments with neural networks (described in the next section) provided comparable results and have a clear path of improvement. Therefore, despite promising results, k-NNs were not pursued any further.

4.6 Basic neural networks

4.6.1 Introduction

Neural networks are graphical models consisting of small computational units "neurons") that compute a non-linear function of their inputs. Units are usually grouped in sequential layers. The first layer is called the input layer, the last layer is the output (the prediction) layer and the layers in-between are the hidden layers. At first, fully-connected hidden layers are explored which means that all units from a previous layer (input layer if it is the first hidden layer) are connected to all the units in the current layer.

Having multiple layers makes neural network models have many weights that are learned (each connection between two units has a weight). Having many weights allows the deeper neural networks to learn complex relationships between the fea-

tures and the targets. Neural network training is usually done by backpropagation [34] which, in practice, allows to train networks of sufficient depth to achieve high performance in a lot of classification and regression tasks.

4.6.2 Experiments

Scikit-learn's implementation of multi-layer perceptron [35] (another name for a neural network) was used to run experiments. Neural networks with 1, 2, 3 and 4 hidden layers were tried. All hidden layers were set to have 300 units. ReLU activation functions [36] were used in hidden units since it's the most popularly used non-linearity in recent networks and in practice works well across a wide range of learning problems. Mean squared regression error was minimised using the Adam optimiser [37]. Training was done using batches of 200 samples until the error improvement after each of two consecutive epochs (complete passes through the training data) was less than 0.000001. No regularisation techniques were used. The results of the neural network experiments are displayed in Table 4.4.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
NN 1 hidden layer of size 300	$9.5*10^{-4}$	$2.3*10^{-2}$	$9.6*10^{-4}$	$2.4*10^{-2}$	
NN 2 hidden layers of size 300	$7.2*10^{-4}$	$2.0*10^{-2}$	$7.4*10^{-4}$	$2.1*10^{-2}$	
NN 3 hidden layers of size 300	$4.7*10^{-4}$	$1.6*10^{-2}$	$5.2*10^{-4}$	$1.7*10^{-2}$	
NN 4 hidden layers of size 300	$3.9*10^{-4}$	$1.5*10^{-2}$	$4.6*10^{-4}$	$1.6*10^{-2}$	53% validation MAE improvement over the linear regression baseline

Table 4.4: Initial results of neural network experiments

4.6.3 Discussion

Neural networks managed to achieve the lowest validation mean squared error from all the models tried (KNN achieved lower MAE but that will also be beaten by the further experiment with neural networks). The performance is increasing as more layers are added which suggests a clear path for further improvements. Together with increased number of layers, the networks also are getting progressively slower to train. In general, this agrees with the hypotheses and experiments made earlier which indicated that more complex models are needed to extract the required information from the features and learn the concepts governing the spacings. However, most experiments in this chapter only scratched the surface of what might be possible as few hyperparameters

Original font	Predicted font
Sphinx of black quartz, judge my vow	Sphinx of black quartz, judge my vow
The quick brown fox jumps over the lazy dog	The quick brown fox jumps over the lazy dog
Thief give back my prized wax jonquils	Thief give back my prized wax jonquils
PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS	PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS
SIXTY ZIPPERS WERE QUICKLY PICKED	SIXTY ZIPPERS WERE QUICKLY PICKED

Figure 4.2: Text samples with the original Times New Roman (regular) font (left) and the same font with letter spacings predicted by the 4 hidden layer neural network model. Note that the Times New Roman font was not part of the training or validation sets. Particularly, pay attention to the words “Sphinx”, “Thief”, “JUGS”, “QUICKLY”.

were optimised (such as changing the distance function for k-nearest neighbours models and adjusting the hidden layer width for neural networks). Since neural networks achieved the best mean squared error as well having an obvious way to increase the model complexity further (more hidden layers), they will be explored in more detail in Chapter 6.

4.7 Discussion

An important general trend to notice is that the models did not display performance in different orders of magnitude. Visualising the predictions of the best neural network model (4 hidden layer network, best in terms of MSE) displays an obvious issue with the problem set up (Figure 4.2). In some cases, the model predicted spacings are quite different than the original ones (even for the pairs that did not actually need any changes). In addition, individually the predicted spacings between pairs might look fine but consistency in spacing might be more important than the preciseness of the spacings. These problems point to the fact that the model does not have any information about the whole of the font itself. The issues of the consistency of predictions across a font and providing models with font information will be addressed in the next chapter.

Chapter 5

Encoding font information

The results from the previous chapter turned out to be not good enough to achieve nice and consistent glyph spacing. One of the most important causes of that is the fact that none of the models from the previous chapter had any information about the font from which a pair distance was being predicted. This limitation restricts the models to predicting the same distance for all the fonts in the training set that have a specific type of glyph pair contour interaction (which our features describe).

An example of the problem described above would be two very similar fonts in the training set with different tracking values. A tracking value is a scalar value that determines the general "spaciness" of the font (the tracking value gets added to the spacing between all the pairs of glyphs in a font and can be both positive and negative) [38]. Since in the current learning set up, a model does not see the difference in predicting the spacing of the same pair of glyphs in differently tracked fonts, it is forced to predict the average of the targets for such a pair of glyphs to minimise the error.

The hypothesis for the experiments in this chapter is that if the models had information about the overall features of the font for which a pair spacing is being predicted, they could learn to tune the predictions to particular fonts and thus achieve lower errors.

Two ways of providing the models with some font specific information were tried and are described in detail in the sections below.

5.1 One-hot encoding of fonts

The most basic way of providing the font information together with the original features is to one-hot encode the font in the training set. One-hot encoding means having an additional feature for each font used in the training set (the feature is set to 1 if the pair described with the features is from that specific font and 0 otherwise). Since about 100 font families totalling to about 245 fonts were used in the dataset construction, this adds 245 additional features to the data.

In the case of the linear regression models, this translates directly to each font having its

specific tracking value (the weight of the specific encoded font feature) that is learned (to minimise the training error) and added to the spacing predicted by the rest of the model. The learning of tracking is a desirable trait for reasons that will be explained in the next section.

In the case of neural networks (and to a lesser extent, other non-linear models), adding one-hot encoded fonts is more powerful. It allows the neural network to produce different values in the hidden layers for different fonts. Since hidden layers build on top of the previous hidden layers and the input layer, in theory, using one-hot encoded features enables the model to use the one-hot encoded features to predict different glyph pairs in different ways. For contrast, in linear regression, all the glyph pairs from a specific font are affected in the same way (a weight of the font feature is added to the spacing).

More generally, providing the model with information about the exact font that each pair is from adds more complexity to the models (more features in total). Therefore, it is expected to improve the training performance of all the models. Since we have a lot of data and validation set uses subsets of pairs from the same fonts, it should improve validation performance as well (particularly because the experiments in the last chapter favoured model complexity).

5.1.1 Experiments

One-hot encoded font features were added to the baseline linear regression model to get an idea of how much adding font information could improve the performance. In addition, one-hot encoded font features were added to the input of the same best performing (in terms of validation MSE) 4 hidden layer neural network from the previous chapter.

A new "same space" baseline was constructed. The new baseline model predicts a single spacing value for each font and applies it to all the glyph pairs in that font. This baseline allows us to compare how much the models learns beyond just putting glyphs same distance from each other (while it is a lower baseline than the one set in the previous chapter, the comparisons to it are more interpretable).

5.1.2 Results

Table 5.1 displays a few interesting results. Firstly, the "same space" (which used only one-hot encoded fonts but no shape describing features) baseline achieves only slightly worse results than the previous linear regression baseline (which used only the shape describing features). The models with added one-hot encoded fonts show big improvements in performance. Including both shape describing and one-hot encoded font features in a linear regression model (3 row in the table) achieved the best result for linear regression (even without polynomial features) by a significant margin (17% validation MAE improvement over the previous best polynomial regression model seen in Chapter 4). These improvements indicate that font information is important and is

likely required to achieve lower errors. Secondly, adding one-hot encoded features to a neural network model reduced the validation MAE by 49% (compared to the same 4 hidden layer model trained only on shape describing features in Chapter 4). As mentioned before, adding one-hot encoded font features gave more power to the neural network model than the simple linear regression models because in neural networks the relation between the one-hot encoded features and the predictions is non-linear. Even though the results from the neural network models look promising, there are significant problems with the approach of using one-hot font encoding for neural networks. These problems are described in detail in the next section.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
Linear regression only 200 features	$1.9*10^{-3}$	$3.4*10^{-2}$	$1.9*10^{-3}$	$3.4*10^{-2}$	Previous baseline (no one-hot encoding)
Linear regression only one-hot encoded fonts	$2.5*10^{-3}$	$3.6*10^{-2}$	$2.5*10^{-3}$	$3.6*10^{-2}$	"Same space" baseline result
Linear regression 200 features + one-hot encoded fonts	$1.1*10^{-3}$	$2.4*10^{-2}$	$1.1*10^{-3}$	$2.4*10^{-2}$	
NN 4 hidden layers 200 features + one-hot encoded fonts	$1.2*10^{-4}$	$7.7*10^{-3}$	$1.5*10^{-4}$	$8.2*10^{-3}$	77% validation MAE improvement over the "same space" baseline

Table 5.1: Results and comparisons of experiments with one-hot encoded fonts

5.1.3 Problems with one-hot encoding

The goal of the project is to use these models for font work which means that we probably want to predict the spacings for a font that is not in the training data or validation data (in the current set up, training and validation data contain glyph pair from the same set of fonts). However, the predictions of spacings for an unseen font are not possible (or at least do not make much sense) by the fact that the unseen font does not have its corresponding one-hot encoded feature. This problem affects linear models and neural networks in different ways.

Linear regression models (using both shape describing features and one-hot encoded fonts) during training take in as input a list of glyph pairs that are described by their in-between whitespace shape and one-hot encoded font features. The outputs are the predicted spacings for each pair of glyphs. Predicting unseen fonts raises the problem that any unseen font will not have its one-hot encoded feature and the font specific weight will not be added to the predicted spacing value. In essence, for an unseen font during test time we have a weight missing because it was not learned during train time.

In linear regression models, the missing weight is not a huge problem. Since the font specific weight gets added to all the pairs in that font, it acts as a tracking value that

controls the overall "spaciness" of the font. Therefore, it is possible to leave the font tracking decision to the designer. After all, it is just one value to adjust and it affects the output linearly. In addition, for an existing that has all its pairs run through the model and spacings adjusted, it is always possible to find such weight that the whole model changes the font in a minimal way. In other words, for an existing font, it is possible to find the font specific weight by minimising the error between the inputs (the original font spacings) and the outputs (the predicted spacings). Since the font specific weight does not affect individual glyph pair spacings (only all of the spacings at once), the model is forced to keep the overall "spaciness" of the original font and the predicted font similar while applying the spacing adjustments for individual pairs that it learned from the training data.

The same approach of solving the problem of the missing weight for an unseen font cannot be applied to neural networks. Features in neural networks interact with each other by design (so that the network is able to construct complex features from the inputs) and therefore, one-hot encoded font features can affect the output of the network in ways that are hard to predict even after the network is trained. The non-linear relationship between the output and the one-hot encoded font means that one-hot encoding of fonts for neural networks is not useful in practice since it is not possible to meaningfully apply the model to unseen fonts. A way to use neural networks for individual glyph pair spacing decisions but also make use of font information in a linear way is needed.

5.2 Separating kerning and tracking

As explained in the last section, the problem with using neural networks with unseen fonts stems from the non-separation of the adjustments to the spacing between glyphs because of how the shapes of the glyphs interact and because of the general "spaciness" of the whole font. The approach chosen was to leave the neural network to make decisions only about the specific glyph pair spacing (kerning) and build a linear model on top of that which adjusts the predicted spacing to be consistent across the font (tracking).

In more detail, the inputs to the neural network are only the features describing the shape between the glyphs. The output of the neural network is one of the inputs to a linear model which also uses the one-hot encoded fonts to predict the final spacing output.

Two different linear models on top of the neural network were tried. The first one (call it model A) only adds a font specific weight (shifts) to the neural network output (in effect, very similar to the linear regression with one-hot encoded fonts).

The second model (call it model B) uses 2 weights. The output of the neural network is multiplied by the first weight and the second weight is added. In general terms, the model A just shifts the neural network output, while model B scales and shifts it.

5.2.1 Experiments

Both linear models were tried on top of the previously used 4 hidden layer neural network and the results compared to the previously best performing neural network (which used one-hot encoded fonts as inputs). The resulting errors are shown in Table 5.2.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
NN 4 hidden layers 200 features + one-hot encoded fonts	$1.2*10^{-4}$	$7.7*10^{-3}$	$1.5*10^{-4}$	$8.2*10^{-3}$	Best model from Chapter 4
NN + shift 4 hidden layers	$1.5*10^{-4}$	$8.6*10^{-3}$	$1.7*10^{-4}$	$8.9*10^{-3}$	Model A
NN + shift and scale 4 hidden layers	$1.2*10^{-4}$	$7.7*10^{-3}$	$1.6*10^{-4}$	$8.4*10^{-3}$	Model B

Table 5.2: Results of providing the 4 hidden layer neural network with the font information in different ways.

5.2.2 Discussion

Rather surprisingly, the errors of the three models compared in Table 5.2 are relatively close. The neural network + shift model (model A) is the easiest to use in practice since it only requires one font specific weight to predict the spacings for an unseen font. In fact, in practice, tracking is done by adding a single constant to all the spacings (corresponding to the model A approach) [39]. Therefore the model A approach was chosen to be pursued in all the following experiments. In all further experiments, the tracking parameter (shift) was learned for each font and added to the final predictions whether it was explicitly mentioned or not (at train and test time for fonts included in the training set). At test time for unseen fonts, the tracking parameter would need to be supplied as another input to the whole model.

Chapter 6

Deep neural networks

Neural networks in the previous chapters were used simply as high complexity models. In this chapter, the focus is more on optimisation of neural networks as well as exploring techniques that can extract as much information as possible from the shape describing features to lower the regression error. The preliminary numeric evaluation is given in the tables in this chapter but more detailed and application-specific evaluation of the best model is done in Chapter 7.

To allow for more flexibility in implementations and training speed, the neural network implementation was switched from scikit-learn to the Keras framework [40] with TensorFlow [41] backend. Keras is a high-level neural network API that features automatic computational graph differentiation and the possibility to train on GPUs. Training on GPUs is a key feature because it allows experimenting with bigger models while keeping the training timeframes reasonable (scikit-learn models only run on CPUs). All training in this and the following chapters was done on Nvidea GeForce GTX970 video card.

Keras was already used to implement the neural networks with linear tracking models on top from Chapter 5 since feeding the output from one model to another and training them both at the same time is not possible in scikit-learn.

6.1 Deep feed-forward neural networks

The results of neural network experiments in Chapter 4 showed that increasing the number of hidden layers increased the network performance. In this section, we experiment with the neural network hidden layer number and their width. In addition, a lot of other hyperparameters influencing the network performance were tuned and although quantitative justification for all of them is not provided for conciseness, the reasons for the choices made are given.

6.1.1 Experiment set-up

Since the framework was switched to Keras, some training options are different than with scikit-learn. All of them are outlined below. Most of the choices described in this section are fairly standard and are included for completeness and reproducibility.

All neural networks were trained using Adamax optimiser (a variant of previously used Adam). It was found to achieve slightly better results than Adam in some cases. Adamax was not available in the scikit-learn implementation of neural networks.

Same as previously, mean squared error was optimised while monitoring mean absolute error as well.

Training was done in batches of 500 (network weights updated every 500 data points) for 50 to 150 epochs depending on how the training was going. The batch size of 500 was found to make the networks converge the fastest (lower batch size increased the time per epoch while higher batch size did not improve the error as fast). Early stopping [42] was used stop training when the lowest error on the validation set was reached.

10% of the training data was randomly picked for validation.

As previously, ReLU activations were used although different options (tanh and sigmoid) were briefly explored. ReLU performed the best in all the vast majority of cases.

Xavier initialisation [43] was used for all the weights. Biases were initialised to zeros. Other options were not explored but Xavier initialisation should be good enough.

6.1.2 Experiments and results

Neural networks with 5 to 11 hidden layers were tried. Hidden layer width was also varied from 300 to 1000 units. The exact configurations used and the results are shown in Table 6.1.

6.1.3 Discussion

Increasing the number of layers as well as the number of units in each hidden layer significantly decreased the errors. The decrease could probably be explained by the fact that having more layers and more units allows the neural network to combine and construct new features and thus extract information easier. Therefore, it might be a good idea to experiment with more techniques that help networks to combine features and extract patterns. One such technique is convolutional layers in neural networks.

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
NN 5 hidden layers 300 layer width	$9.8*10^{-5}$	$6.9*10^{-3}$	$1.3*10^{-4}$	$7.5*10^{-3}$	
NN 7 hidden layers 300 layer width	$7.7*10^{-5}$	$6.0*10^{-3}$	$1.2*10^{-4}$	$7.0*10^{-3}$	
NN 9 hidden layers 300 layer width	$6.3*10^{-5}$	$5.3*10^{-3}$	$1.1*10^{-4}$	$6.5*10^{-3}$	
NN 9 hidden layers 500 layer width	$4.8*10^{-5}$	$4.4*10^{-3}$	$9.9*10^{-5}$	$5.8*10^{-3}$	
NN 9 hidden layers 1000 layer width	$3.7*10^{-5}$	$3.7*10^{-3}$	$9.0*10^{-5}$	$5.2*10^{-3}$	
NN 11 hidden layers 1000 layer width	$3.8*10^{-5}$	$3.6*10^{-3}$	$8.8*10^{-5}$	$5.1*10^{-3}$	43% validation MAE improvement over the 4 hidden layer model B from Chapter 5 86% validation MAE improvement over the "same space" baseline

Table 6.1: Results of experiments on hidden layer number and width

6.2 Convolutional neural networks

Convolutional neural networks are a different type of neural network structure that makes use of convolutional, pooling and fully-connected layers.

Convolutional layers contain units that have local receptive fields instead of being fully-connected (a specific unit would only have connections to a few spatially close units from the previous layer). The local receptive fields of the units usually overlap. Some units share their weights with other units. The units that share the weights form filters being applied all throughout the inputs. The application of filters results in the network being able to detect patterns invariant of their spatial location in the inputs.

Pooling layers are usually included after every one or a few convolutional layers to reduce the dimensionality of the intermediate activations and concentrate the important information that was extracted. Most popularly, Max pooling is used, which splits the input (to the pooling layer) into groups of spatially close units (same number of units in each group) and outputs the maximum value from each group.

Fully connected layers are usually included after all the convolutional and pooling blocks just before the output layer. They function in the same way as in fully-connected networks except for the fact that there is usually very few of them and the inputs are

flattened (to 1 dimension) outputs from the convolutional layers.

The motivation for using convolutional neural networks is the fact that an assumption is made that input features form a one dimensional image (of the whitespace between the pair of glyphs). This assumption allows to greatly reduce the number of weights in the network as well as focus the network on learning location invariant local features (in this case, the local features could be narrowing or widening of the whitespace). Because of the weight sharing, the local features can be learned in any place among the features (as opposed to learning a local feature only in one specific place in the features by fully-connected networks). The location invariance could mean, for example, that handling glyph serifs at the baseline could be easily generalised to handling serifs the same way at the top of the glyphs.

6.2.1 Experiments

Training convolutional neural networks involves quite a few more decisions than training simple fully-connected networks. Among others, there are choices in network architecture, size of filters in every convolutional layer, number of filters in every convolutional layer, pooling type and size.

6.2.1.1 Architectures

$$INPUT \rightarrow [[CONV \rightarrow ReLU] * N \rightarrow POOL] * M \rightarrow FC * K \rightarrow OUTPUT$$

Figure 6.1: Parametrised architecture of convolutional neural networks

Architecture advice from [44] was taken to design most of the experiments. More specifically, the reference mentions the architecture construction pattern displayed in Figure 6.1

CONV, ReLU, POOL and FC here refer to convolutional layers, non-linearity layers with ReLU activation function, pooling layers and fully connected layers respectively. The architecture parameters are N, M and K.

The experiments were designed to try the most popularly used N, M and K values for building CNNs. However, the advice is more tailored to visual (from images which are 2 dimensional data) classification (as opposed to the current task which is regression from 1 dimensional data). Therefore, some ad-hoc architectures that were based on intuition about the problem at hand were also tried.

6.2.1.2 Other hyperparameters

The filter size used in the convolutional layers was 3 (5 was tried briefly but produce worse results). 3x3 filters are most commonly used even for high dimensional image

data.

The pool size for max pooling was 2 and applied so that they do not overlap (stride of also 2). Almost all applications use non-overlapping small pools since they already reduce the dimensionality aggressively. The input dimensionality in this problem is relatively low to begin with so there is even more reason to use small pooling layers.

The number of filters in the convolutional layers was selected depending on the rest of the architecture choices as to allow the model to finish training (via early stopping) in a couple of hours. In practice, the number of filters did not affect the performance of the trained models a lot as long as it was not too low (usually at least 10 filters). The number of filters in the convolutional layers was doubled after each pooling layer to keep the number of total activations per layer approximately consistent (a common practice in most CNNs).

6.2.2 Results

The most successful or interesting model structures and hyperparameters are provided below.

1. Basic CNN

   An architecture of N = 1, M = 2, K = 2 given by Figure 6.1. Filter numbers of 16 and 32 for the two convolutional layers respectively. Fully connected layers had width of 1000 units each.

2. Deeper Basic CNN

   An architecture of N = 1, M = 4, K = 2 given by Figure 6.1. Filter numbers were 16, 32, 64, 128 for the four convolutional layers respectively. Fully connected layers had width of 1000 units each.

   This is essentially the same architecture as the “Basic CNN” but made deeper. Since “Basic CNN” worked relatively well already with 2 convolutional layers, it was hoped that adding more convolutional and pooling layers might provide significant improvements.

3. 4C1P

   An architecture of N = 4, M = 1, K = 2 given by Figure 6.1. All convolution layers had 16 filters. Fully connected layers had width of 1000 units each.

   During training of “Deeper Basic CNN” and similar models, it was noticed that adding more pooling layers makes the models harder to train (in terms of convergence speed) and worse (in terms of achieved validation MSE) in general. Therefore, this experiment contains more convolutional layers than the “Basic CNN” but less pooling layers.

4. 4C

$$\begin{array}{rl} \text{INPUT} \rightarrow [\text{CONV} \rightarrow \text{ReLU}] * 4 & \rightarrow \\ \text{INPUT} & \rightarrow \end{array} \quad \text{FC} * 9 \rightarrow \text{OUTPUT}$$

Table 6.2: The architecture of network ”4CNN”. Feeding the inputs again in the middle of the network.

An architecture of 4 convolutional layers and two fully connected layers after that (No pooling layers). The filter number for all convolutional layers is 16. Fully-connected layers have a width of 1000.

Since the input features are not high dimensional, even the smallest pooling layers might be too aggressive in reducing the dimensionality. The idea of checking the necessity of pooling layers at all comes from [45].

5. 4CFC

   A custom architecture of combining the convolutional fully-connected network approaches. The intuition was that even though CNNs and deep fully-connected layers both achieve comparable errors, it might be the case that they learn differently. Therefore, a custom architecture was designed to first feed the inputs to 4 convolutional layers, then concatenate the resulting flattened activations with the inputs again and input the resulting tensor to 9 fully-connected layer network (of width 1000 each). The resulting network structure is laid out visually in Figure 6.2.

The performance achieved by these models is displayed in Table 6.3

	Training		Validation		Notes
	MSE	MAE	MSE	MAE	
Basic CNN	$2.3*10^{-5}$	$3.0*10^{-3}$	$7.8*10^{-5}$	$5.1*10^{-3}$	
Deeper Basic CNN	$3.5*10^{-5}$	$4.0*10^{-3}$	$1.0*10^{-4}$	$6.1*10^{-3}$	
4C1P	$1.8*10^{-5}$	$2.6*10^{-3}$	$7.3*10^{-5}$	$4.8*10^{-3}$	
4C	$1.6*10^{-5}$	$2.5*10^{-3}$	$7.2*10^{-5}$	$4.9*10^{-3}$	Best validation MSE achieved
4CFC	$1.7*10^{-5}$	$2.2*10^{-3}$	$7.9*10^{-5}$	$4.5*10^{-3}$	Best validation MAE achieved

Table 6.3: Results of CNN experiments

6.2.3 Discussion

Basic CNN achieved similar performance to the best performing 11 hidden layer fully connected layer from the previous section. Basic CNN model has an order of magnitude fewer weights so it achieving the same performance shows the effectiveness and higher efficiency (in terms of the number of weights needed) of convolutional networks.

However, going deeper with CNNs did not work out. The most likely reason is that the “Deeper Basic CNN” network had too many pooling layers which discarded too much information.

Stacking convolutional layers with little to no pooling improved the results. Both ”4C1P” and ”4C” models have a relatively small number of weights but achieve the same or better results than models with more parameters. Stacking more convolutional layers without pooling was tried as well, but proved to be too difficult to train (networks would get stuck or train extremely slowly). Increasing the number of filters also didn't improve the performance.

Combining the convolutional and fully-connected networks in ”4CFC” produced an impressive mean absolute error. However, the mean square error is what was optimised and is more important. Intuitively, square error penalises extreme errors more. In visualised text, a lot of small errors in spacing would look better than a few big spacing errors.

The ”4C” model was selected as the best model and used for all the evaluation in the next chapter.

Chapter 7

Evaluation

The evaluation of the best model from the last chapter was done in a few ways. Firstly, quantitative evaluation with mean squared errors and mean absolute errors was done. Secondly, the best performing model was compared to the auto-kerning functionality of FontForge by measuring the errors as well as visually inspecting the results on some standard fonts. Finally, a section is dedicated to more thoroughly look at the visualisations of applying the best model to a few different fonts.

7.1 Quantitive evaluation

7.1.1 Test error on held out glyph pairs

25000 random glyph pairs were held out from the training and validation sets to be used for the final evaluation. The best model recorded the mean squared error of $8.09*10^{-5}$ and mean absolute error of $4.91*10^{-3}$. The errors are not too far from the validation errors. The dataset is big and the number of experiments that would need to be run to notice any overfitting on the validation set would be high. Note, however, that these pairs are from the same fonts that were used for training and validation. Therefore the test set only shows how well the model generalises on unseen pairs but not necessarily on unseen fonts. Generalisation on unseen fonts is more interesting, therefore, the test errors of this kind are not particularly meaningful or interesting.

7.1.2 Errors on standard fonts

A different way to measure how well the model is performing was devised. A few widely used and generally well-regarded fonts were chosen (the standard fonts used are displayed in Figure 7.1).

As mentioned in Chapter 4, producing spacings for fonts that were not in the training set required additional input of a tracking parameter (which corresponds to the weight

Helvetica	The quick brown fox jumps over the lazy dog
Times New Roman	The quick brown fox jumps over the lazy dog
Baskerville	The quick brown fox jumps over the lazy dog
Akzidenz Grotesk	The quick brown fox jumps over the lazy dog
Franklin Gothic	**The quick brown fox jumps over the lazy dog**
Didot	The quick brown fox jumps over the lazy dog
Gotham	The quick brown fox jumps over the lazy dog
Rockwell	The quick brown fox jumps over the lazy dog
Minion	The quick brown fox jumps over the lazy dog

Figure 7.1: Standard fonts (left) and their samples (right) that were used for evaluating the quality of model predictions.

of the one-hot encoded font feature for fonts in the training set). The tracking parameters were learned during training for seen fonts but need to be provided to obtain predictions for unseen fonts.

An assumption was made that the standard fonts mentioned previously are well-spaced. Therefore, the tracking parameter was chosen to minimise the mean squared error between the original font spacings and the full model predictions. In other words, the tracking parameter is chosen so that the average space between two glyphs as predicted by the full model is as close to the original as possible.

The minimisation of the error was implemented by linear regression of no features with targets being the differences between the original spacing and the predicted spacing without tracking (only by the neural network without the shift model on top) for each pair of glyphs in a specific font. The linear regression learns no weights but a single bias that minimises the MSE of full model predictions and original font targets. The bias is used as the best tracking parameter for the font.

In general terms, obtaining the tracking parameter for an unseen font this way means keeping the global spacing of the font as close to the original as possible while still making adjustments to the spacings of individual pairs. All the pairs have almost the same spacing on average but the specific pair spacings differ in ways that the model thinks improves the spacing quality.

The resulting errors for the standard fonts comparing them to the “same space” baseline are displayed in Table 7.1. The comparison of the mean absolute errors with the width of the letter “H” in that specific font is also given to make the errors more interpretable and comparable across the fonts.

Font	"Same space" baseline mean squared error	"Same space" baseline mean absolute error	Prediction mean squared error	Prediction mean absolute error	Prediction mean absolute error as a percentage of letter H width
Helvetica	$1.6*10^{-3}$	$3.2*10^{-2}$	$4.1*10^{-4}$	$1.5*10^{-2}$	2.8%
Times New Roman	$1.1*10^{-3}$	$2.3*10^{-2}$	$8.9*10^{-4}$	$2.3*10^{-2}$	3.4%
Baskerville	$2.9*10^{-3}$	$4.2*10^{-2}$	$9.2*10^{-4}$	$2.3*10^{-2}$	2.9%
Akzidenz Grotesk	$1.4*10^{-3}$	$3.0*10^{-2}$	$4.7*10^{-4}$	$1.5*10^{-2}$	2.6%
Franklin Gothic	$1.5*10^{-3}$	$3.0*10^{-2}$	$4.5*10^{-4}$	$1.5*10^{-2}$	2.8%
Didot	$2.4*10^{-3}$	$3.9*10^{-2}$	$1.2*10^{-3}$	$2.7*10^{-2}$	3.8%
Gotham	$1.8*10^{-3}$	$3.3*10^{-2}$	$3.0*10^{-4}$	$1.2*10^{-2}$	2.1%
Rockwell	$7.7*10^{-4}$	$2.1*10^{-2}$	$8.9*10^{-4}$	$2.3*10^{-2}$	3.5%
Minion	$7.7*10^{-4}$	$2.1*10^{-2}$	$7.4*10^{-4}$	$2.0*10^{-2}$	2.9%

Table 7.1: Errors obtained by predicting standard fonts

7.1.3 Discussion

The first thing to notice is that all the errors on the standard fonts are significantly worse that the errors on the test set. The cause of this difference is most likely the model overfitting on the set of fonts (as opposed to the glyph pairs) that were used in the training, validation and test sets. Even though the neural network itself did not have any information regarding the font of each pair, it managed to learn representations that are more suitable for the whole set of fonts in training/validation/test sets rather than unseen fonts.

The model prediction quality was not uniform across all the standard fonts. Sans serif fonts (Helvetica, Akzidenz Grotesk, Franklin Gothic and Gotham) were predicted significantly better on average than serif fonts (Times New Roman, Baskerville, Didot, Rockwell and Minion). Serif fonts are usually more tricky to space manually as well. The serifs might touch each other and not reduce legibility (e.g. in the pair "kn") and vice versa (e.g. in the pair "nm"). Sans serif glyphs do not touch each other in the vast majority of cases (doing so would almost always reduce legibility). The contours of sans serif fonts are also usually straight lines or smooth curves and might be easier for a neural network to learn and represent. Evidence of this hypothesis is the Rockwell font which was predicted worse than the baseline solution most likely due to unusually strongly pronounced serifs (slab serifs).

Relatively high errors do not necessarily mean that the predicted spacings are bad or that the model did not generalise well (in terms of the visual quality of its predictions). The visualised sampled text from standard fonts (predicted and original) is showcased in Section 7.3. The relationship between the errors and the goodness of the letter spacings is further discussed in the next chapter.

7.2 Comparison to FontForge automatic kerning functionality

The predicted standard fonts from the previous section were compared to the results produced by FontForge automatic kerning functionality mentioned in Chapter 2.

7.2.1 FontForge set-up

FontForge sets the kerning for each pair according to an optical separation parameter that is passed in as an argument to the autokern function. The algorithm then adjusts the kerning of each pair so that it is as close to the optical separation (as measured by an optical separation measure, default or custom user-provided) as possible. The default optical separation measure was used in the comparisons. The optical separation parameter was optimised to minimise the MSE between the autokerned font and the original font (similarly to how the tracking parameter was found in the neural network models). The minimisation was done using a golden section search algorithm [46]. The same solution of applying linear regression to find the optimal parameter could not be applied because the relation of optical separation parameter and the produced spacings in FontForge automatic kerning functionality is non-linear.

7.2.2 Results and discussion

Font	FontForge autokerned	"Same space" baseline mean squared error	Prediction mean squared error
Helvetica	$6.2*10^{-3}$	$1.6*10^{-3}$	$4.1*10^{-4}$
Times New Roman	$7.3*10^{-3}$	$1.1*10^{-3}$	$8.9*10^{-4}$
Baskerville	$1.1*10^{-2}$	$2.9*10^{-3}$	$9.2*10^{-4}$
Akzidenz Grotesk	$5.7*10^{-3}$	$1.4*10^{-3}$	$4.7*10^{-4}$
Franklin Gothic	$4.4*10^{-3}$	$1.5*10^{-3}$	$4.5*10^{-4}$
Didot	$1.0*10^{-2}$	$2.4*10^{-3}$	$1.2*10^{-3}$
Gotham	$6.6*10^{-3}$	$1.8*10^{-3}$	$3.0*10^{-4}$
Rockwell	$5.5*10^{-3}$	$7.7*10^{-4}$	$8.9*10^{-4}$
Minion	$5.7*10^{-3}$	$7.7*10^{-4}$	$7.4*10^{-4}$

Table 7.2: Results of the "same space" baseline, best model predictions and FontForge automatic kerning functionality.

The resulting errors are compared to the results from the previous section in Table 7.2.

The errors produced by FontForge automatic kerning are significantly higher than both the prediction errors and the baseline errors for all the standard fonts. The errors distribution is also non-uniform. Serif fonts seem be harder to space well for automatic kerning in FontForge as well.

The big errors from FontForge autokerned fonts seem to be due to a few grave errors rather than accumulated over lots of pairs. This is showcased further in the next section.

Although the autokerned errors being bigger than the baselines is surprising, it is explained by the fact that the FontForge automatic kerning algorithm is recommended as just a starting point for manual spacing work and it is not optimising for the best MSE.

7.3 Visual comparisons

The MSE of produced spacings is not always a good measure of the quality of the spacing. In this section, some of the fonts were visualised and compared in terms of their spacings in same sampled text. A few of the more interesting comparisons with comments are given here, but more can be found in Appendix A.

7.3.1 Implementation details

This section describes the implementation of the font visualisation process and does not have any evaluation results.

The most of the software capable of visualising text with the user choice of fonts either needs the font to be installed on the operating to function or does not make use of full kerning information available in the fonts if the font is not installed. Specifically, ImageMagick [47] and Matplotlib [48] libraries were tried but could not produce correctly spaced images.

A roundabout way to visualise a large number of fonts without having to install them was developed. HTML templates were generated that use CSS to load and use the specific fonts as well as to enable font kerning capabilities. PhantomJS [49] implementation of a headless browser was used to render the HTML templates and take screenshots of the resulting web pages to obtain the images for visual comparison of differently spaced fonts.

7.3.2 Standard fonts

The results of predicting “Times New Roman” is showcased in Figure 7.2.

In general, although the visual comparison is highly subjective, the predicted text samples are more consistent in terms of spacing than the two other solutions. Both the FontForge automatic kerning algorithm and the “same space” baseline solution do not produce consistent results for both lowercase letters and capital letters in most cases. Looking more closely, it is always possible to find a few pairs that stick out and are highly noticeable in both solutions while the predicted text looks much more similar to the original and the disparities can sometimes be disputed (such as the spacing of the triplet “PAC” in word “PACK” in Figure 7.2).

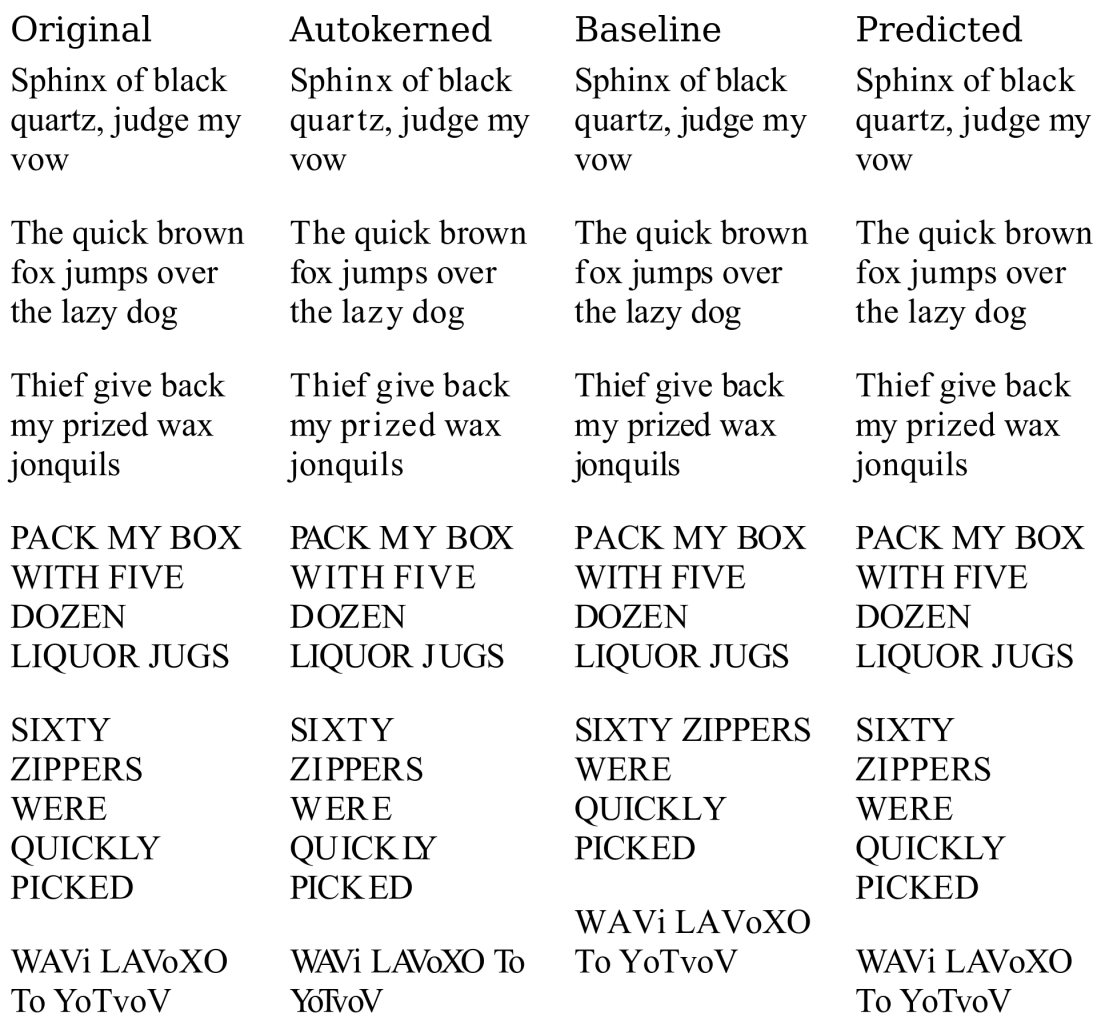

Figure 7.2: Model used to predict the spacings for the Times New Roman font. The "Original", "Autokerned", "Baseline" and "Predicted" columns refer to the original font, font autokerned using FontForge, font spaced with the "same space" baseline solution and font with the model predicted spacings respectively.

7.3.3 Predicting from a set of glyphs

Another perspective of looking at the results produced by the model is to see the outputs as the "fixed" version rather than it trying to mimic the original. Using fonts with subpar spacing, it is possible to use the model predicted spacings to improve the font and compare the visual results.

7.3.3.1 Monospaced fonts

Monospaced fonts are designed so that all the characters take up the same amount of horizontal space. Therefore, it does not make sense to try and predict spacings of monospaced fonts. However, we can take the glyphs from a monospaced font and treat

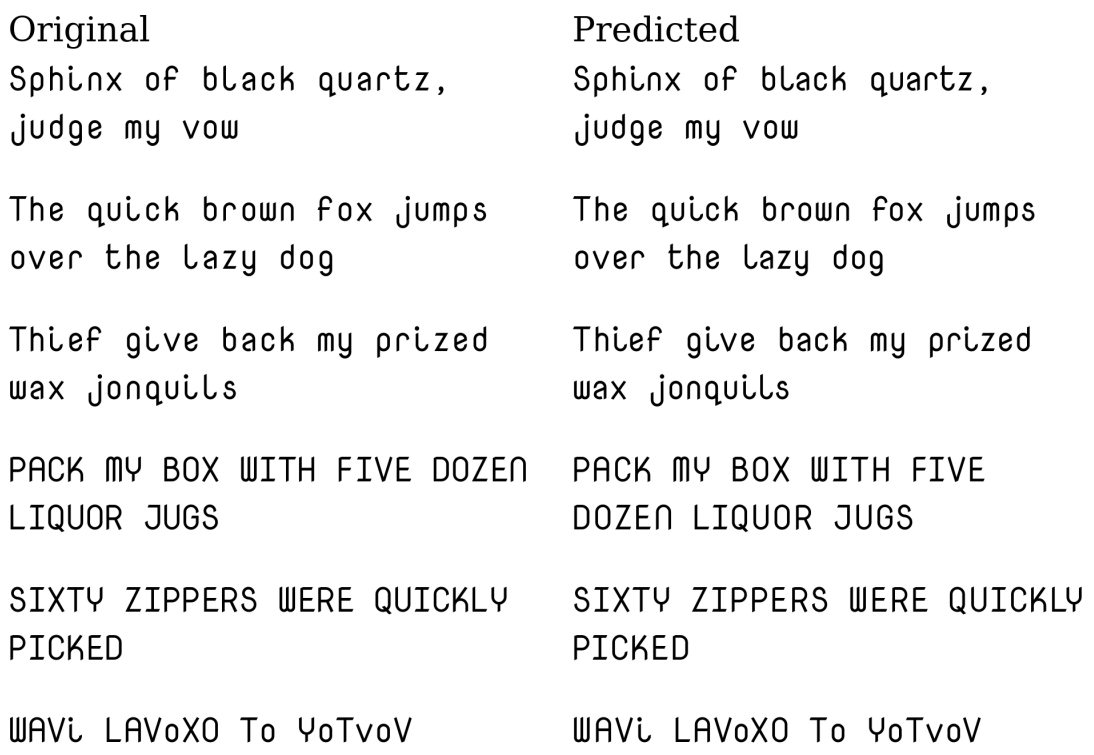

Figure 7.3: Model used to predict the spacings for a monospaced font Nova Mono. The spacings are made to be more balanced around each letter while the overall width of the font is held to be approximately the same. Particularly, note the spacings around letter “i” and the word “fox”.

it as a font without any spacing information. The model can then predict the spacings, in essence properly letter-spacing and kerning a font. The glyphs in a monospaced font are not designed to be used in a proportional font so the result is still not very visually pleasing. However, the resulting visualisation in Figure 7.3 helps to understand what the model is doing and how the spacings are changed according to glyph shape interactions.

7.3.3.2 Unfinished fonts

During the development of a font, the letter spacing and kerning stage sometimes might be delayed or skipped (more so for amateur font designers). Generating the correct spacing and kerning information might be considered tedious and not as interesting as designing the glyph shapes but is nonetheless important for a visually pleasing result. Figure 7.4 showcases how the model predicted spacings could be used to assist the process of development of a font.

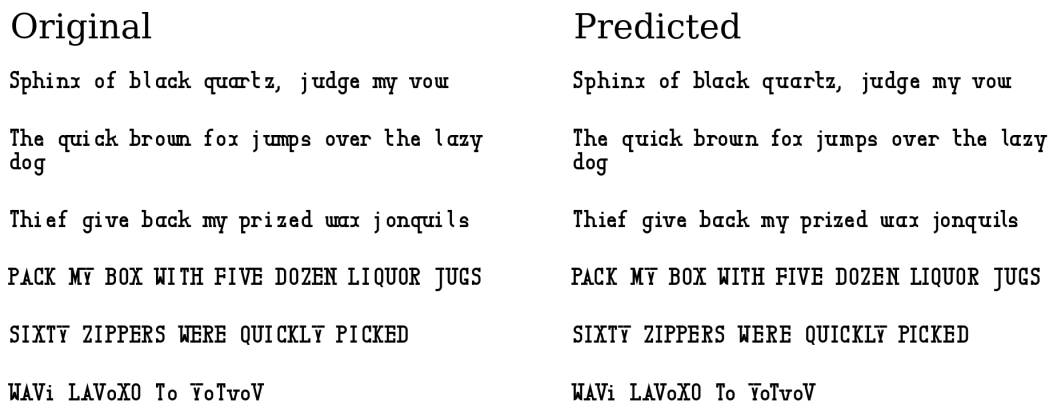

Figure 7.4: Model used to predict the spacings for a mid-development font Iont Slab. The models produces spacings that are a good start for the designer to fine tune.

Chapter 8

Conclusion

The goal of this project was to investigate the possibility of using machine learning for fully or partially automating the kerning and letter spacing part of font development. The minimal objective was to learn to predict some hold out kerning information in existing fonts and quantitatively evaluate the predictions.

The existing tools were reviewed. Most of the existing tools for automatic kerning and spacing are either experimental, not good enough or commercial and closed source. No open-source implementations explicitly using machine learning were found.

A dataset for supervised learning was created using the most popular fonts in the Google fonts repository. A process for extracting glyph information from fonts and constructing features describing the spacing between glyphs was developed.

Linear regression was used to satisfy the minimal objective by predicting the spacings between glyphs and evaluating the predictions by calculating errors and comparing them to baselines.

A system modelling both the kerning (via a neural network) of a pair and the tracking (via a per font constant) of a font was developed to improve the quality of the predictions and generalisation capability.

The final system was quantitatively evaluated on a test set and on unseen standard fonts in terms of the similarity of its predictions to the original standard font spacings. Additionally, the system was compared to an existing open source tool for automatic font kerning. Visual evaluation was done by providing figures comparing different spacing solutions applied to the same text.

The developed model produced better errors than both the existing FontForge automatic kerning solution and the baseline solution in all tested cases of standard fonts except for font Rockwell where the baseline provided a slightly better solution.

There are a lot of possible further research paths to improving the results, the approach to the problem in general and practicality of the tool. Some of the possible paths are outlined in the next section.

8.1 Future work

8.1.1 Dataset quality

What the model learns is highly dependent on the training set. Since the training did not face any overfitting on the pairs, it might be beneficial to train on less but higher quality data (less but more popular fonts).

In addition, the fonts could be separated into groups of different types and different models for the different groups trained separately. The results already indicated that the model sometimes struggles with serif fonts. Italic and more calligraphic fonts could also be separated.

Finally, it could be beneficial to make sure that a higher percentage of the pairs in the training set were actually kerned in the original fonts. Kerned pairs are almost sure to be spaced properly while unkerned pairs might be not because they are unlikely to appear in English text, the kerning data is incomplete or for other reasons.

8.1.2 Glyph shape representation

The approach taken in this project was measuring the horizontal distances between the glyph outlines at various heights for every pair of glyphs. While this approach of representing the whitespace between two glyphs worked and was inspired by the existing design of an algorithm in FontForge automatic kerning functionality, the usage and design of it were not fully justified. Other representations could be investigated. A more direct approach could be to convert a pair of glyphs to one bitmap and feed it directly to the neural network. The network would probably need more layers to learn its own internal representation but it might be more expressive and meaningful that the one enforced in this project.

8.1.3 Different approaches

In this project, glyph pairs were spaced. Another approach that some font designers take is to take triplets of glyphs and adjust the spacings of the two pairs so that the middle glyph is visually the same distance away from both of the side glyphs. Going even further, newer font formats support contextual kerning which allows specific spacing adjustments for strings of more than two glyphs.

There was an implicit assumption in this project that there is a set of kerning values for a font that are correct. However, kerning can also be a design choice. Investigation on clustering fonts with different styles of kerning and whether factoring that in could help improve the model.

Chapter 9

Plan for next year

The overall topic of the project ("Machine learning of fonts") is broad. Although, as described in the previous section, there are still a lot of different ways to keep attacking the same problem of automatic font spacing and kerning, next year the focus will be on a different problem in typography.

9.1 Problem description

Not all fonts are created equal. In the wild, especially in fonts where not as much design effort was put in, a variable number of characters is defined and have glyphs assigned to them. For example, a certain font might be missing diacritics (such as ö, š, ø), while another might not have the Euro symbol (€) or the inverted question mark punctuation symbol (¿).

The absence of some characters, punctuation or symbols might make some fonts significantly less desirable or unusable in some contexts even though the look of the font is good otherwise. The simplest solution is to insert the characters from other fonts but they might appear out of context and ruin the visual text style and consistency.

The goal is to attempt to solve this problem by designing a machine learning approach that is capable of learning a font "style" and using that style to generate the required missing characters.

9.2 Existing tools and approaches

Not a lot of font toolsets explicitly offer the functionality of generating missing glyphs. No complete tool that would perform the required functionality using machine learning was found at all.

Some cases of missing characters can be handled more easily. Some font formats, as well as typesetting software, have methods to combine glyphs [50] and thus produce

a lot of more rarely used diacritic characters. FontForge offers some basic automatic accented character creation [51] although the exact method is not explained. The implementation of it will need to be inspected directly. Both of these methods do not handle the harder cases of missing symbols that are not slight modifications of the existing ones.

There has been some work done on interpolating whole fonts (changing some representation of the font style to obtain new fonts). Two such approaches are described in [52] and [53]. The former uses Gaussian processes to construct a generative manifold of fonts using which it is possible to interpolate and extrapolate some standard fonts. The latter uses deep learning to train a neural network which can then create a "font vector" which can then be manipulated to visualise some interpolated fonts or characters.

9.3 Proposed approach

One approach to try is to train deep neural networks to learn to represent the style of fonts. Then by using the same, or a different network, generate some missing characters by using the data of how they should look in other fonts.

Quite a lot of work is needed to set up the learning problem. Similarly to the spacing project, a dataset will need to be made. Google fonts will likely be used again. It would probably be easier to develop networks that learn from bitmaps so all the vector glyphs would need to be converted. The output of the learning models will also be in bitmap images. To achieve anything close to being practical, bitmaps will need to be converted back to vector graphics. Some raster-to-vector software (such as [54]) can be used for vectorising bitmaps. It might be the case that a lot of tinkering with post-processing and vectorisation parameters will be needed to obtain anything useful.

The completion objective will be to predict some held out characters from standard fonts and quantitatively evaluate the results by calculating errors between the original and predicted bitmaps. The resulting errors will be compared to some simple baselines (taking the character from the "closest" font or taking the "averaged" character over other fonts).

Bibliography

[1] Maria dos Santos Lonsdale, Mary C. Dyson, and Linda Reynolds. Reading in examination-type situations: the effects of text layout on performance. *Journal of Research in Reading*, 29(4):433–453, 2006.

[2] Leyla Akhmadeeva, Ilnar Tukhvatullin, and Boris Veytsman. Do serifs help in comprehension of printed text? An experiment with Cyrillic readers. *Vision Research*, 65:21 – 24, 2012.

[3] Michael L. Bernard, Barbara S. Chaparro, Melissa M. Mills, and Charles G. Halcomb. Comparing the effects of text size and format on the readibility of computer-displayed Times New Roman and Arial text . *International Journal of Human-Computer Studies*, 59(6):823 – 835, 2003.

[4] Keith Rayner, Timothy J. Slattery, and Nathalie N. Bélanger. Eye movements, the perceptual span, and reading speed. *Psychonomic Bulletin & Review*, 17(6):834–839, 2010.

[5] Yu-Chi Tai, James Sheedy, and John Hayes. Effect of letter spacing on legibility, eye movements, and reading speed. *Journal of Vision*, 6(6):994+, 2006.

[6] Donald E Payne. Readability of typewritten material: proportional versus standard spacing. *Visible Language*, 1(2):125–136, 1967.

[7] Google. google/fonts: Font files available from Google Fonts. `https://github.com/google/fonts`. (Accessed on 25/1/2017).

[8] FontForge Open Source Font Editor. `https://fontforge.github.io/en-US/`. (Accessed on 25/01/2017).

[9] Automated Kerning With iKern Typographica. `http://typographica.org/on-typography/automated-kerning-with-ikern/`. (Accessed on 05/04/2017).

[10] Typographical Concepts. `https://developer.apple.com/library/content/documentation/StringsTextFonts/Conceptual/TextAndWebiPhoneOS/TypoFeatures/TextSystemFeatures.html`. (Accessed on 27/03/2017).

[11] Igino Marini. Manifesto iKern: type metrics and engineering spacing (autospacing) and kerning (autokerning) for type designers. `http://ikern.com/k1/`. (Accessed on 25/01/2017).

[12] Igino Marini. Introduction to iKern. http://ikern.com/k1/wp-content/uploads/iKern_Intro_2016_02_07.pdf. (Accessed on 25/01/2017).

[13] New Fell Types and new iKern site. — Typophile. http://www.typophile.com/node/46301. (Accessed on 05/04/2017).

[14] Matthew Chen. TypeFacet Autokern. http://charlesmchen.github.io/typefacet/topics/autokern/index.html. (Accessed on 25/01/2017).

[15] Auto Width and Auto Kern. http://fontforge.github.io/en-US/documentation/reference/autowidth/. (Accessed on 25/01/2017).

[16] Dutch Type Library. DTL FontTools. http://www.fontmaster.nl/. (Accessed on 25/01/2017).

[17] KernMaster — PublinkPublink. http://pub-link.com/products/kernmaster/. (Accessed on 25/01/2017).

[18] FontLab Typographic Tools - font editors and converters - TypeTool. http://old.fontlab.com/font-editor/typetool/. (Accessed on 25/01/2017).

[19] Font Combiner — Custom web fonts and web icons. http://fontcombiner.com/create. (Accessed on 27/01/2017).

[20] SIL Open Font License (OFL). http://scripts.sil.org/OFL. (Accessed on 25/01/2017).

[21] Apache License, Version 2.0. https://www.apache.org/licenses/LICENSE-2.0. (Accessed on 25/01/2017).

[22] Ubuntu Font License v1.0. http://font.ubuntu.com/ufl/ubuntu-font-licence-1.0.txt. (Accessed on 25/01/2017).

[23] fonttools/fonttools: A library to manipulate font files from Python. https://github.com/fonttools/fonttools. (Accessed on 25/01/2017).

[24] svg.path 2.2 : Python Package Index. https://pypi.python.org/pypi/svg.path. (Accessed on 27/01/2017).

[25] 8.6. bisect Array bisection algorithm Python 3.6.1 documentation. https://docs.python.org/3/library/bisect.html. (Accessed on 05/04/2017).

[26] Developer API — Google Fonts — Google Developers. https://developers.google.com/fonts/docs/developer_api. (Accessed on 05/04/2017).

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] Natural Language Toolkit NLTK 3.0 documentation. http://www.nltk.org/. (Accessed on 24/02/2017).

[29] sklearn.linear_model.LinearRegression scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. (Accessed on 02/03/2017).

[30] sklearn.decomposition.PCA scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. (Accessed on 06/04/2017).

[31] sklearn.preprocessing.PolynomialFeatures scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html. (Accessed on 03/02/2017).

[32] sklearn.neighbors.KNeighborsRegressor scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html. (Accessed on 26/03/2017).

[33] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. *When Is "Nearest Neighbor" Meaningful?*, pages 217–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[34] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[35] sklearn.neural_network.MLPRegressor scikit-learn 0.18.1 documentation. http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html. (Accessed on 26/03/2017).

[36] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In Johannes Frnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010.

[37] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014.

[38] Tracking Your Type - Fonts.com. https://www.fonts.com/content/learning/fontology/level-2/text-typography/tracking-your-type. (Accessed on 06/04/2017).

[39] CSS letter-spacing. http://www.quackit.com/css/properties/css_letter-spacing.cfm. (Accessed on 28/03/2017).

[40] Keras Documentation. https://keras.io/. (Accessed on 28/03/2017).

[41] TensorFlow. https://www.tensorflow.org/. (Accessed on 28/03/2017).

[42] Lutz Prechelt. *Early Stopping — But When?*, pages 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[43] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[44] CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks/#layerpat. (Accessed on 29/03/2017).

[45] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for Simplicity: The All Convolutional Net. *CoRR*, abs/1412.6806, 2014.

[46] minimize_scalar(method=golden) SciPy v0.19.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/optimize.minimize_scalar-golden.html#optimize-minimize-scalar-golden. (Accessed on 01/04/2017).

[47] Convert, Edit, Or Compose Bitmap Images @ ImageMagick. https://www.imagemagick.org/script/index.php. (Accessed on 06/04/2017).

[48] Matplotlib: Python plotting Matplotlib 2.0.0 documentation. http://matplotlib.org/. (Accessed on 06/04/2017).

[49] PhantomJS — PhantomJS. http://phantomjs.org/. (Accessed on 06/04/2017).

[50] LaTeX/Special Characters - Wikibooks, open books for an open world. https://en.wikibooks.org/wiki/LaTeX/Special_Characters#Escaped_codes. (Accessed on 29/03/2017).

[51] Design With FontForge: Diacritics and Accents. http://designwithfontforge.com/en-US/Diacritics_and_Accents.html#FontForge's_basic. (Accessed on 29/03/2017).

[52] Neill D. F. Campbell and Jan Kautz. Learning a Manifold of Fonts. *ACM Transactions on Graphics (SIGGRAPH)*, 33(4), 2014.

[53] Analyzing 50k fonts using deep neural networks Erik Bernhardsson. https://erikbern.com/2016/01/21/analyzing-50k-fonts-using-deep-neural-networks/. (Accessed on 29/03/2017).

[54] Peter Selinger: Potrace. http://potrace.sourceforge.net/. (Accessed on 29/03/2017).

Appendices

Appendix A

Visual comparisons

A.1 Standard fonts

Original	Predicted
Sphinx of black quartz, judge my vow	Sphinx of black quartz, judge my vow
The quick brown fox jumps over the lazy dog	The quick brown fox jumps over the lazy dog
Thief give back my prized wax jonquils	Thief give back my prized wax jonquils
PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS	PACK MY BOX WITH FIVE DOZEN LIQUOR JUGS
SIXTY ZIPPERS WERE QUICKLY PICKED	SIXTY ZIPPERS WERE QUICKLY PICKED
WAVi LAVoXO To YoTvoV	WAVi LAVoXO To YoTvoV

Figure A.1: The original and predicted Helvetica font. The model mimics the spacings quite well as it takes a while to find the differences. To see one difference, note the pair “rt” in word “quartz”. Also, note the slight kerning on the pair “oT” in the last row in the predicted font. The original font did not kern the pair most likely due to it not being common in normal usage. However, the model kerns the pair slightly and is penalised for it in terms of MSE. It could be argued that the pair should be kerned and the error is “undeserved”.

Figure A.2: A harder case of kerning the font Baskerville. The FontForge automatic kerning functionality kerns the capital letters that fit together too closely while the baseline solution makes the mistake of letting some letters touch each other which reduces legibility (pairs “jo” and “DO”). The predicted model avoids both issues although makes a mistake in pair “AV” by not kerning it enough. The triplet “XTY” in word “SIXTY” could be argued to be spaced better in the predicted version.

Figure A.3: Font Rockwell. The only font that was predicted better by the baseline solution than the model in terms of MSE. The most obvious mistakes made by the model are the spacings in the triplet “PAC” (in word “PACK”) and pair “Vo”. Presumably, the slab (blocky) nature of the serifs resulted in the baseline solution being able to produce a reasonable spacing save for some specific pairs (such as “WA” and “AV”).

A.2 Improved fonts

Some unkerned fonts found in the Google font repository were predicted and compared below.

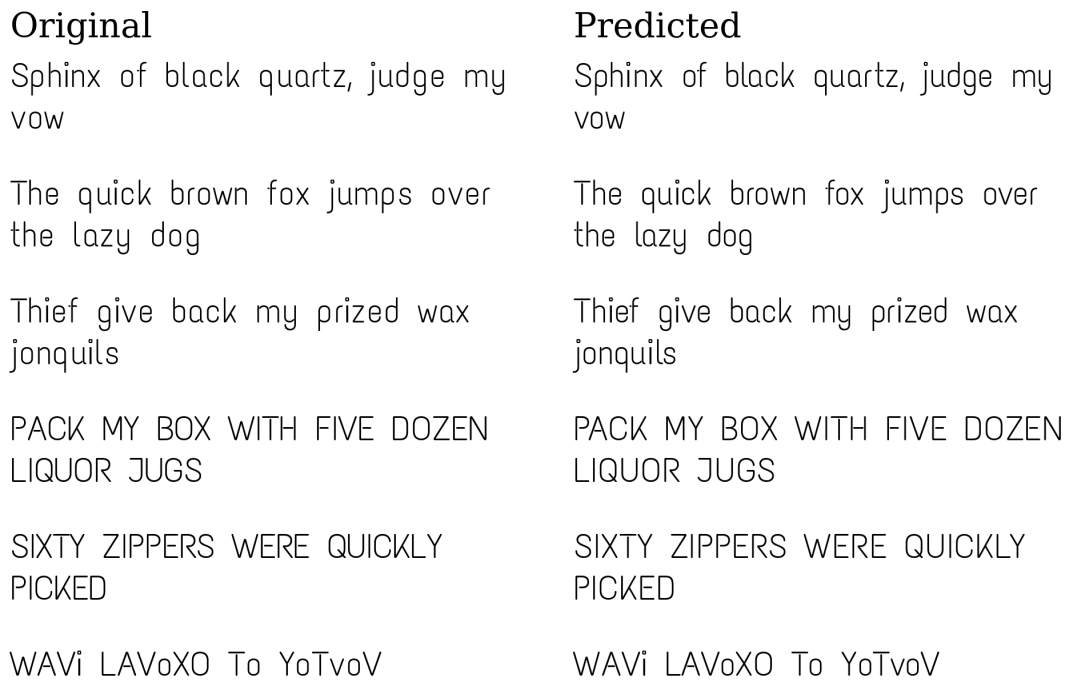

Figure A.4: Font TextMeOne from the Google font repository. The predicted version has the spacings consistency increased in most cases. Particularly note the words “quick”, “fox” and “PACK”. The spacing is not perfect (The difficult cases in the last row are still not great) but in a better state than in the original font.

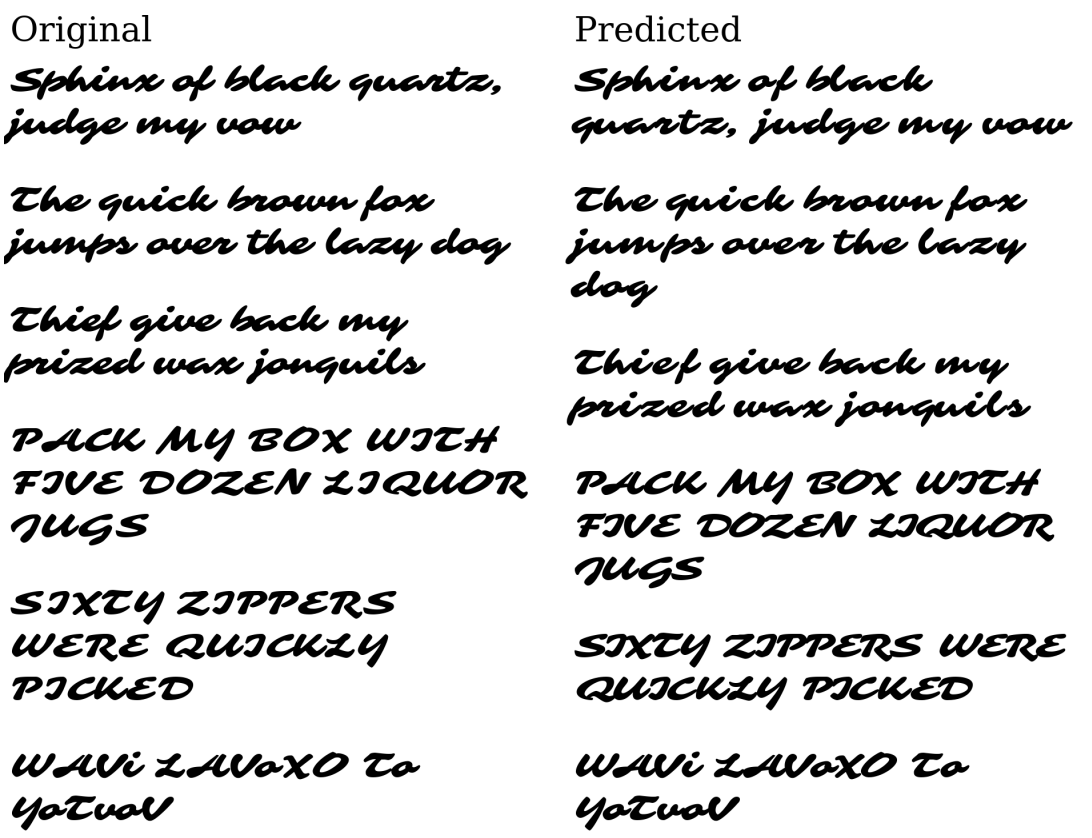

Figure A.5: Font Sarina from the Google font repository. It shows that the model does not generalise well to more calligraphic fonts. Since the same tracking value is applied to both lowercase and uppercase letters, the model cannot distinguish which letters should be connected and which should be not.