Font Style Transfer Using Deep Learning

Antanas Kascenas

MInf Project (Part 2) Report

Master of Informatics
School of Informatics
University of Edinburgh

2018

Abstract

Font style transfer is a task of inferring a font style from a set of characters and generating a different set of characters exhibiting the inferred style. Font style transfer is a generalisation of a problem of filling in fonts that do not define certain characters.

This projects explores the applicability of deep neural networks for inferring style and generating character images. Existing approaches are reviewed and discussed. A general system for font style transfer is designed and evaluated both quantitatively and qualitatively. The results show significant improvement over a previous method.

Two improvements to the designed system are proposed in generative adversarial training and image function representations. Both approaches show qualitative improvements and possible future work directions.

Acknowledgements

I would like to thank my supervisor, Dr. Iain Murray, for being an encouraging supervisor and supporting the development of the project by valuable feedback and advice.

I would also like to thank James Sloan for introducing me to the idea of unconditional order discriminators for adversarial training that ended up being an important part 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. The choice of a font or a typeface is one of the main characteristics of the style and feel of a text piece. It is accepted among designers that different fonts and their combinations are suitable for different purposes. Some example guidance on font choice is provided in [2, 3].

Hundreds of thousands of fonts have been created by font designers for purposes such as newspapers, books, advertisements, displayed text and video games. However, font design is a demanding task and it is often the case that fonts are created only specifically for the purpose that they are needed. This can result in some fonts being close but not quite usable for different purposes. For example, fonts might be missing certain symbols because only certain characters were needed or a font might only be suitable to be used in certain size.

This project is motivated by the problem of missing glyphs in fonts. Glyphs are symbols in a font that are used as parts or whole characters. In practice, it is common to use fallback fonts for characters that are not defined in some specific font. An example of such behaviour in Microsoft Word is explained in [4]. An example of a font fallback can be seen in Figure 1.1. Using fallback fonts often produces visually subpar results.

Figure 1.1: Font “Josefin Slab” does not define latin characters with diacritics that are used in the Lithuanian script. Therefore, a fallback font is used for one of the characters. Example produced using Google Docs.

The handling of missing characters in fonts is a set of problems that can involve automatically obtaining some form of substitute glyphs, glyph metrics, positioning information and other font specific metadata. Therefore, the creation of a tool that

obtains and inserts missing characters into existing fonts is an extensive task. Some work on handling font metadata was done in part 1 of the MInf project. This project only focuses on the generation of raster images of font characters. More generally, the aim of this project was to explore the feasibility of using machine learning and deep neural networks in particular to generate missing characters in fonts given their existing characters.

1.1 Main contributions

- Literature review on existing approaches to style modelling on font data.
- Data preprocessing code for generating an image dataset for training from raw font files.
- Design and implementation of neural networks for one-to-one character image translation.
- Design and implementation of generalised neural networks to handle multiple input and output images.
- Adaptation and implementation of generative adversarial network framework to improve the generated image outputs.
- Adaptation and implementation of image as a function representation for high resolution image generation.

1.2 Work done in the MInf1 project

The first part of the MInf project focused on using machine learning methods to try and predict the correct kerning - spacing between letters in printed text, given some font features. The MInf2 project focuses on an entirely different topic. Therefore, besides slight knowledge of handling font files, no work was reused in this project.

Chapter 2

Background

2.1 Character representation

The majority of computer fonts used today are specified in a vector format. Vector formats store the outlines of the characters and their relative size. Vector characters need to be rasterised, that is, converted into a pixel representation to be displayed or printed. The conversion between vector and pixel representations is lossy. Vector representation cannot be reliably and fully restored given only the pixel representation. Pixel image tracing is a process of extracting lines, curves as well as shape fill colours from an image in pixel representation to generate a closely matching image in vector representation. Rasterising (converting to a pixel representation) an image and then tracing it to obtain a vector representation again is a lossy process. However, given a pixel representation at high enough resolution, it is possible to trace it for a vector representation that closely approximates the original vector image.

The choice of a representation affects the overall task of generating missing characters in two major ways. Firstly, whatever representation is chosen must be convertible to a specific vector format that is used for storing glyphs in font files. The conversion must preserve enough detail to be usable in the desired context. For example, if characters are generated in pixel representation they must be of high enough resolution for tracing to produce useful vector images. Secondly, the machine learning aspect of the task is heavily affected by the choice of representation. Supervised machine learning algorithms differ in their capability to handle features and targets of different types (e.g. continuous, categorical, sequences, optional features etc.). In addition, the types of features and targets affect the learning performance of algorithms. Therefore, the choice of character representation is a key choice in this project that is closely related to the choice of the machine learning method. Some of the possible methods and character representations are described in Section 2.5.

2.2 Example system

An example system performing font style transfer on character images could work as follows. Each font is considered separately. For each font, some characters are available and some are missing. It is known which characters are available and which are missing. That is, for each available character image, it is known which character it is. Likewise, for each missing character image, it is known what character it should be. The character classes (e.g 'A', 'c', 'T') are what can be referred to as the character content information.

Furthermore, since the available characters for each font come from the same single font, they share some attributes among themselves and consistently differ in those attributes from other fonts. Examples of such attributes could be letter thickness, slant, presence of serifs, letter width etc. These attributes are what can be referred to as the style of the font.

An example system performing font style transfer could take one of the available character images (e.g. character 'T') and estimate the style attributes exhibited in that image. For example, estimate the thickness of the stem of 'T' as well as measure the slant. A subsystem could then take the estimates of these attributes (the style information) and the class of one of the missing characters (the content information, e.g 'd') and output the image of character 'd' exhibiting the style attributes estimated on character 'T'. The task is known as a style transfer task since the style was transferred from content 'T' to content 'd'.

These ideas of style and content are used extensively in explaining the previous related work and the ideas developed in this project.

2.3 Modelling of style and content

A frequent theme in previous related literature and style transfer literature in general [5] is the explicit and separate modelling of style and content.

The advantage of modelling style and content separately for the process of generating new font characters is intuitive.

To generate a specific character given other characters from the same font, both the content information and the style information are needed but are obtained in different ways. The content information is usually known since character images extracted from raw font files always have the character classes defined. The style information needs to be estimated from the character images that are available.

Therefore, the font style transfer task can usually be split into two parts. Firstly, estimating the style information from available information. Secondly, generating the character of required content and the estimated style.

2.4 Neural networks and deep learning

This section provides general information on neural networks and their training to provide context for understanding the building blocks of the deep learning approaches described in the previous related work and the developments in this project.

In general, neural networks are models that take input and produce output and can be trained to approximate a function mapping the input to the output. A classical example of a neural network is a classifier network taking an image of a handwritten digit as the input and producing the classification of the digit as the output (e.g. '1', '2', ...).

2.4.1 Fully-connected neural networks

Feedforward neural fully-connected networks or FCNs consist of layers. Each layer consists of neural network units. The units in the first layer are the inputs to the neural network and the units in the last layer are the neural network outputs. The layers between the first and the last are known as the hidden layers. The units in the hidden layers have values that are weighted sums of the unit values in the previous layer passed through a non-linear activation function. That is, $h_i^l = f(\sum_{k=0}^{n} w_{k,i}^l h_k^{l-1})$ where $h_{i,l}$ is the ith unit value in the lth layer of the network, $w_{k,i}^l$ is the weight of the kth unit value in $l-1$th layer in the calculation of the weighted sum for the ith unit value in the lth layer, n is the number of units in the $l-1$th layer and $f()$ is the non-linear activation function. All the weights determining the values of weighted sums are the parameters of a neural network model. A neural network is trained by changing the parameter values.

2.4.2 Convolutional neural networks

Convolutional neural networks or CNNs are fully-connected networks constrained in two ways. Firstly, each hidden unit value is determined by only a fixed number of neighbouring units in the previous layer where in a fully-connected network all units from the previous layer contributed to every value in the next layer. Secondly, each hidden unit value in a layer uses the same weights (but different units values from the previous layer) in calculating their values. The limited connectivity and weight sharing are designed for image processing and enforce translation invariance as well as a certain hierarchical image composition prior. Convolutional neural networks are usually preferred for working with image data.

2.4.3 Neural network training

Neural networks are trained by defining a differentiable loss function between the neural network output and the targets corresponding to the neural network input. Once the loss function is calculated, the gradients for all the network parameters are obtained by backpropagation - an algorithm that assigns "credit" to each neural network parameter

by calculating its contribution to the loss. Once the gradients for every neural network parameter are known, they are updated using a gradient based optimiser such as gradient descent.

2.4.4 Neural network implementation

The only requirement for training by gradient based optimisers is that the gradients with respect to the loss function for every network parameter can be obtained. Backpropagation allows to have any connectivity between neural network units therefore, the neural network architectures are not limited by the fully-connected and convolutional structures described above.

Neural networks are mostly discussed as building blocks in the related work and the design of the systems in this project. In the descriptions of the networks used in this project the exact implementation details are often omitted in favour of explaining the concepts being modelled. However, the low-level details are essential to functional models and have a large impact on the performance of neural networks. While knowledge of how neural networks work is not necessary to understand what the models in the previous related work and this project are doing, it is required to implement the models.

2.4.5 Generative adversarial networks and adversarial training

Generative adversarial networks [6] or GANs in their basic setup are two different neural networks that are trained to compete with each other. In an unsupervised setting, one of the networks, the generator, is generating an image that is likely to be among the distribution of images in the training set. The other network, the discriminator, is trying to predict whether the generated image came from the training set or was generated by the generator. The generator is improved to generate images that fool the discriminator while the discriminator conversely is improved to achieve better classification of true or generated images. In a game theory sense, the Nash equilibrium or the point where neither of networks can make a good change is the point where the generator is generating images that the discriminator cannot distinguish from the real images in the training set.

Conditional adversarial networks [7] take GANs to the superviser setting. Instead of having a set of images, the training set consists of image pairs. Both the generator and the discriminator take the first image of the pair as their input. The generator tries to generate an image that fools the discriminator to predict that the generated image is the second image of the pair. Conditional adversarial networks have been used for image-to-image translation tasks such as translating a greyscale photo to a colorised photo or a photo to an edge map.

Gradient reversal layer [8] is a different way to enable adversarial training. Contrary to the first setup, the generator and discriminator networks share a common subnetwork, a feature extractor. The feature extractor and the generator are connected normally while the values going to the discriminator go through a gradient reversal layer. The gradient

reversal layer negates the gradients passing through it during backpropagation. As a result of using the gradient reversal layer, it is possible to train the the feature extractor to avoid picking up on certain features. It has been shown useful for domain adaptation or in the case of [1] forcing the latent style vector not to contain any information about the content.

2.5 Related work

There have been attempts to use machine learning models to extrapolate fonts from single or multiple characters. However, the attempts vary in terms of the exact problem as well as the dataset used. There is no established problem specification for font style transfer. Most likely due to copyright reasons, all attempts use different sets of fonts for their experiments. This section mentions some previous attempts and highlights the similarities and differences to this project.

2.5.1 Methods using non-pixel representations

2.5.1.1 Separating Style and Content [9]

Representation used: warps of black particles from a reference shape into the specific letter shape. No further details are specified.

Data used: 5 fonts with 5 letters each for training. 1 font for testing.

The general problem of factoring out style and content from an observation is attempted. Bilinear models explicitly represent the two factor structure and thus are used to model the two factors of style and content. Two variants are proposed. A symmetric model of $y^{sc} = a^s \cdot W \cdot b^c$ where y^{sc} is the observation of content c and style s and a^s, b^c and W are parameters that need to be estimated. An asymmetric model is $y^{sc} = A^s \cdot b^c$ where A^s is the parameter matrix specific to a style s. The asymmetric model is used to fit the font data since it has more parameters to model the style and thus is more flexible. A closed form procedure exists to fit the asymmetric model using SVD factorisation.

Once the style and content parameters are estimated they can be used to generate observations with new style given known content or vice versa.

Since the bilinear factorisation approach is a relatively simple model, the representation is the key to good results. It is difficult to compare the method to the other approaches in this chapter without more details on the representation that was used and considering the tiny amount of data used for training.

2.5.1.2 Learning a Manifold of Fonts [10]

Representation used: universal polyline representation where the charactor topology is consistent across fonts. For example, the top of the letter ‘O’ in two different fonts

should be at the same point in the polyline representation of both characters.

Data used: 46 standard fonts with 512 points sampled along every character outline.

The method is based on the observation that given the set of all possible closed curves, the curves representing possible character outlines lie on a low dimensional manifold within it. The approach consists of two stages.

Firstly, the universal representation of all characters is achieved using character outline matching. For each character, points are densely sampled along the outline of a character using its vector representation. The point sequences of characters of the same class across fonts are then normalised and aligned using an energy model. The energy model is based on preserving consistent curvature and location information as well as elasticity regularisation. The energy minimisation procedure results in aligned polyline representations for all characters where points on the polyline correspond to the same semantic locations on the same characters across fonts. The polylines are represented by a fixed-size sequence of point locations along an interval of $[0, 1]$.

Secondly, the manifold of possible character outlines is found using dimensionality reduction. Gaussian Process - Latent Variable Model (GP-LVM) is used to find latent variables that represent the manifold space. Once the model is fitted, the latent variables can be considered as the style parametrisation of fonts. The style parametrisation can be manipulated to interpolate or extrapolate new fonts.

The goal of this approach is an unsupervised task of manifold learning but it still allows the inference of missing characters of the same style. The projection to the latent style parametrisation can be done with only having a subset of all font characters which then allows to generate the missing characters of the same style.

The method produces sharp and realistic characters since the character representation used allows to recover vector-like character outlines. In addition, the projection to latent space can make use of any number of observed characters which is an advantage most other approaches do not have. However, the method relies heavily on the success of the energy model based outline matching, which does fail on non-standard fonts as admitted by the authors.

2.5.1.3 FlexyFont: Learning Transferring Rules for Flexible Typeface Synthesis [11]

Representation used: vector representation.

Data used: 88 fonts for upper case letters, 57 fonts for lower case letters.

The method is motivated by the fact that font designers often keep important character parts for re-use. Therefore, style can be often considered to be the consistent usage of some standard character parts such as serifs.

The approach consists of multiple stages. Firstly, each character vector representation is converted to a set of simple polygons. The polygon represented glyphs are then decomposed into parts and character skeleton using a skeletonisation technique. Some

manual input is required to mark some boundaries of semantic character parts. A character is then represented as a collection of semantic pats that includes the skeleton of the character. The style of a font is represented by the similarities among all the parts used by all the characters in a font. For example, the similarity of the bars of characters 'A' and 'H' would be one entry of the style vector for the font.

Secondly, a Bayesian Gaussian Process Latent Variable Model (BGPLVM) is used to model the font part similarity vector. The model is fit during training and the unobserved entries of the vector are inferred using the fitted model during testing.

Finally, the generation of a new character given the inferred style vector is done by arranging the semantic parts onto a skeleton that is obtained by template deformation.

This is the only approach that is designed specifically to handle decorative fonts. The generated characters are of high quality since vector representation is used for all the parts. In addition, the method has the potential for professional stylistic consistency since the same parts are re-used. However, the approach comes with a few downsides as well. Some manual input is required which is not the case for all the other methods described in this section. In addition, the obtained semantic parts might not always align smoothly in the generated characters which might require some additional post-processing.

2.5.2 Deep learning methods using pixel representations

2.5.2.1 From A to Z: Supervised Transfer of Style and Content Using Deep Neural Network Generators [1]

Representation used: 40×40 greyscale pixel images of each character.

Data used: 1,839 fonts including both standard and decorative fonts.

The method is essentially an enconder and decoder pair inspired by the variational autoencoder [12]. The encoder maps the image input to a multivariate Gaussian latent space. The latent space is forced to represent only the style but not the content via adversarial training. The method makes an assumption that a separate latent style is needed for every image class in the output. Since the encoder generates the style only for the input subset of all characters, the style for all the missing characters is generated by a learned linear combination of the styles of the observed characters. Thus, a latent multivariate Gaussian distribution representing the latent style is inferred for every character of the specific font. The required latent variable is then sampled from the corresponding multivariate Gaussian distribution and used together with a content switch as an input to the decoder subnetwork which generates the desired output image. The content switch can then be changed to generate the remaining missing characters.

The visual image output quality is additionally improved by using an additional imposter detector implemented by an adversarial network that helps to prevent the generation of unrealistic images. Both cases of adversarial training are done via gradient reversal.

All model sub-networks are fully-connected neural networks and use structured image similarity (SSIM) [13] as the optimisation objective. The whole model is trained in two stages. Firstly, the encoder and the decoder are trained. Secondly, the whole network, meaning the encoder, decoder and the adversary subnetworks, are trained.

While the method was evaluated on the task of generating all Latin font characters given only the character 'A' as the input, it can be adapted to take any number of observed characters as the input. However, the number of input images must be fixed and thus cannot be changed during test time. The approach applies no pre or postprocessing to the raw pixel image representations as is common with deep learning approaches. As a result, some outputs are blurry or contain unrealistic artefacts.

Furthermore, the model was quantitatively evaluated using SSIM scores which allows for objective comparisons to be made.

2.5.2.2 Multi-Content GAN for Few-Shot Font Style Transfer [14]

Representation used: 64×64 greyscale pixel images. Randomly generated font-specific colour texture is applied to every character.

Data used: 10,000 fonts of different styles with only capital letters. Randomly generated colour gradients are added as a texture for the fonts, augmenting the training set to 20,000 fonts.

The method uses a conditional adversarial network [7] conditioning on the character that is generated. The modelling consists of two stages. Firstly, modelling the overall character shape given the input. Secondly, modelling the ornamentation, that is, the colour and texture to apply on the previously generated character shape, given the input. Both stages are optimised at the same time.

Both the glyph generation network and ornamentation network are of standard conditional GAN architecture with modified inputs. The input to the glyph shape generation subnetwork consists of all the font character images stacked together. The missing images (unobserved characters from a specific font) are substituted with pure black images. The input to the glyph ornamentation subnetwork consists of glyph shape mask generated by the glyph shape generation subnetwork.

Least squares GAN training (LSGAN) [15], L1 loss and other regularisation losses are combined using a weighted sum and used as the optimisation objective. The model is trained end-to-end and evaluated qualitatively.

The approach is attractive because it is capable of making use of any number of observed characters during test time. However, most of the focus is given to improving the ornamentation part of the method which prioritises highly stylised fonts. Stylised fonts are usually decorative in shape as well which makes the method face similar problems as in [1], namely blurry output and artefacts for more special glyph shapes. The problems are not as noticeable because of a larger dataset but still present.

2.5.2.3 Other deep learning methods

A simpler task is explored in [16]. A neural network model is designed to take the 36×36 images of characters 'B', 'A', 'S', 'Q' as inputs and generate the rest of the capital letters. Some evidence of performance gains when generating all characters with the same model as opposed to separate models for different characters is provided. The neural networks used do not contain any domain specific modifications and training is done on simple pixel wise mean square loss. Training was performed on 10,000 fonts. A small subset of the results was evaluated by a single human rater.

An approach in [17] explores style transfer in Chinese and Japanese fonts. The project combines a few innovations in generative adversarial networks to construct a model optimising 4 different losses. Due to high count of Chinese and Japanese characters in fonts, the model only learns one-to-one translations between characters. However, the model is trained to be able to take character class pair as input and output. 27 fonts are used with 1000 to 2000 characters taken from each font.

A project in [18] describes generating Hindi characters. The model uses 3 character images as input and a single character image as the output. Data augmentation was applied to character images from 141 fonts due to shortage of training data. The neural network model learns 3 different latent representations from the input images which then are concatenated to generate the output image. Thus, the approach is not flexible in terms of the input character subset.

2.5.3 Discussion

The previous work on the topic of generating characters is distributed on a wide range of problem variations. In addition, the training datasets used differ greatly both in size and type of fonts used. Despite the differences, some common trends can be noticed.

Firstly, deep neural networks are increasingly commonly applied to the problem of character generation despite of the shortcomings of low resolution pixel output. However, the approaches that use larger datasets do manage to produce relatively high quality results as well as generalise to more unique fonts. With more data becoming available and deep learning methods still improving, neural networks are a promising approach.

Secondly, all neural network approaches described used pixel representations. Pixel representations are easier to work with in machine learning and especially with neural networks. However, pixel representations are high dimensional and therefore are usually only used only with deep learning methods. Simpler models often make use of some kind of preprocessing to convert data to a more constrained representation. While simpler representations such as in [9] and [10] might not generalise to more original and decorative fonts, they allow to recover the vector representations to some extent. Recovering vector representations solves issues such as blurriness and low-resolution artefacts like jagged lines which are a problem in deep learning approaches that produce low resolution pixel outputs.

Thirdly, most described neural networks approaches lack flexibility in terms of inputs and outputs. Ideally, a model would be capable of making use of observation sets of varying size at test time while generating either a specified character or all of them at once. Only the approach in [14] satisfies these requirements. However, the model [14] implements the functionality in a parameter inefficient way of simply zeroing out the unobserved character inputs. Flexibility of neural network approaches is a possible area of improvement.

Fourthly, separation of style and content either explicitly as in [1] or implicitly as in [14] is a common theme in most of the mentioned approaches. Modelling style explicitly allows to go further beyond generating missing characters into exploring the latent style space and possibly combining or interpolating new fonts.

Finally, the evaluation and comparison of the results is difficult since percepted visual quality of the results is not reflected well by pixel-wise losses such as mean squared error or the structured image similarity metric [13]. There is no agreement on a suitable evaluation procedure besides simple visual observation. The training dataset sizes vary wildly as well, making it hard to distinguish which models are better ideas and which only work better due to more data.

Chapter 3

Initial experiments and baselines

Baselines were required to enable comparisons that can guide further experiments. This chapter evaluates the errors of simple baseline solutions as well as basic neural network models on a basic subtask of the project problem.

3.1 Data preparation

Fonts were required to create a dataset for training and evaluation of models. Already preprocessed image data were provided by the authors of [1] upon request. The image data received could be directly used in training neural network models without any preprocessing. However, before the image data was received, image generation procedure was reproduced as much as possible following the preprocessing descriptions in [1] in the case of having to use a self made dataset.

3.1.1 Converting font files to character images

Fonts with open licenses were obtain from the Google font repository [19]. The fonts were filtered to use only one font per font family. A font family contains stylistically identical fonts that usually differ only in thickness or slant. The restriction prevents same style fonts from appearing in train and test data splits together. The resulting 858 fonts were split into 600, 128 and 130 fonts for training, validation and test respectively.

Each of the 52 Latin characters for every font is then rasterised into 40×40 images using the FreeType font engine [20]. 15 pixels from the bottom is selected as the baseline for the characters to allow the bottom of descenders to be seen. Fonts vary greatly in the size of each character, therefore the following procedure is used. For each font, the optimal font size is found by linear search on the biggest character width and height. The font size is increased until any character height or width exceeds 39, then the previous font size is selected. Once the optimal font size for each font is found, all the characters are horizontally centred and saved into 40×40 images. Images of randomly selected fonts are shown in Figure 3.1.

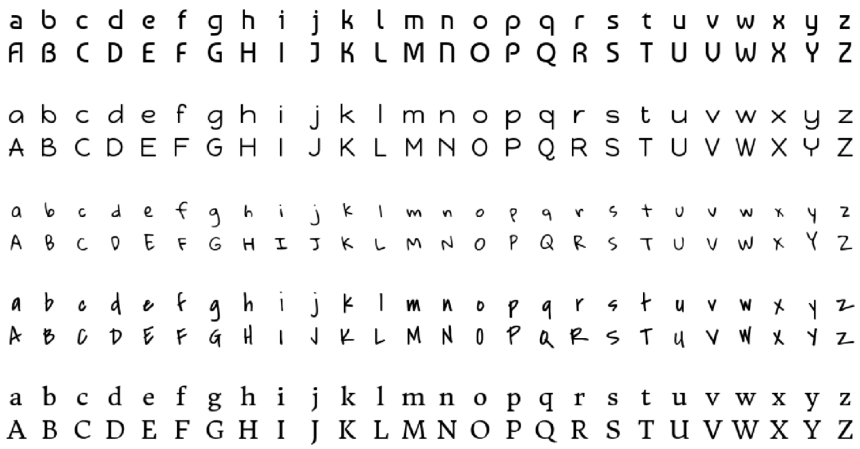

Figure 3.1: Some preprocessed font samples showing both lowercase and uppercase characters.

3.2 Problem set up

The formal definition of the problem could be formulated as follows. Given images of any non-empty subset of characters from any font, generate the images for all the characters in the font. Trivially, the characters that are given do not need to be generated and can be taken as is.

The problem of transferring the style of the given characters onto to the generated characters could be approached in multiple ways, as shown in the previous chapter. For example, characters can be deconstructed and reconstructed from their semantic parts or ornamentation separately applied onto character shape templates. An approach of more interest is the direct extraction and separation of style in some representation.

3.2.1 Advantages of explicit representation of style

Modelling the latent style variable allows interesting applications of the style representations:

- Style representation space interpolation. Interpolating the style space can allow exploration and possibly generation of completely new fonts. For example, interpolating between bold and thin style fonts could yield images of a regular thickness characters.
- Style combination. Having explicit style representation might allow to combine styles for multiple fonts in a similar way that vector arithmetic con be done on word embeddings. For example, combining bold, italics and serif fonts might allow to generate a character set with a combination of these features not present in the existing data.

- Dimensionality reduction. Compressing the style representations might allow to visualise the relationships between styles of different fonts. Visualisation could help to understand how well the model learns various style attributes
- Clustering. Clustering on style representations can be used to group fonts into stylistic classes. As an example, clustering could be used to improve font browsing and selection user interfaces.

Considering the advantages, the approaches explored in this project attempt to model font style in a vector representation.

3.2.2 Problem subtasks

Considering that a latent style vector will be modelled, the problem can naturally be divided into two subtasks. Firstly, the style needs to be extracted from the subset of given character images. Secondly, the extracted style needs to be used to generate the character images. The subtasks can largely be explored and optimised separately and are discussed throughout the experiments in this and the following chapters.

3.3 Basic task

The most basic subtask that is representative of the overall project problem would be to train a model which can perform one-to-one character image translation. That is, given a single character from a font, generate a different character from the same font. For simplicity, a neural network model is trained to take the image of character ‘A’ as the input and produce the image of character ‘B’ as the output.

3.4 Trivial solutions

Two trivial solutions were selected to set the baselines:

- Taking the per-pixel average of all ‘B’ characters in the training set. The average is used to measure the error between each generated character and the average to set a baseline. This method produces blurry images that could not be used in practice but is good enough to measure if models in further experiments are learning anything useful.
- Using the nearest neighbour approach of picking the the character ‘B’ in the training set that corresponds to the character ‘A’ that is closest to a given input. The picked character image is used to measure the error.

3.5 Basic architectures

The basic task of generating character ‘B’ given character ‘A’ involves the 1,600 input values representing the image pixels of character ‘A’ and 1,600 output values representing the image pixels of the output character ‘B’. The simplest neural network architecture is to have a fully-connected network with 1,600 units in the input layer, 1,600 units in the output layer and some hidden layers in-between.

The slightly more advanced architectures can be split into two modules. One that encodes the input image to a style vector and another that uses the style vector to generate the output image. The first module is referred to as the style extractor and the second module is referred to as the image generator.

The style vector is a bottleneck connecting the style extractor and the generator. In a working system, it must contain all the information about the style that could be inferred from the input image. Thus, the bottleneck style vector can be considered as the style representation learnt by the neural network.

The explicit modelling of the latent style is not strictly necessary for the basic task of one-to-one image translation. However, it will be needed as the inputs and outputs are generalised to variable numbers of images. Therefore, some baselines were constructed with the latent style bottleneck although the full effect and specifics of the bottleneck are explored later in the project.

The main architectural choice explored with the baseline experiments is whether to use fully-connected neural networks (FCNs) or convolutional neural networks (CNNs). The biggest difference between CCNs and FCNs is the image translation invariance enforced by CNNs by sharing of weights, thus reducing the overall number of parameters needed when compared to a similar FCN. CNNs have been successfully applied on image data achieving state-of-the-art results in most computer vision tasks [21]. However, CNNs are usually applied to photography data while the image data used in this project exhibits different features. The character images are horizontally and vertically aligned and therefore translation invariance might not be as important as in natural images. The experiments in the chapter guide the choice of the architecture for further work in the project.

4 different baseline architectures were tried:

- Architecture 1. A baseline zero hidden layer neural network. Similar to a simple linear regression.
- Architecture 2. A FCN with two hidden layers.
- Architecture 3. A CNN style extractor module and FCN generator module.
- Architecture 4. A CNN style extractor module and CNN generator module.

3.6 Error measurement

There is no clear metric that evaluates the quality of image reconstruction. A common loss function to train with is the mean squared loss or L2 loss. The conditional adversarial networks paper used mean absolute loss or L1 error. The outcomes of training on both these losses are compared. The structured image similarity [13] (SSIM) or more specifically the dissimilarity (DSSIM, $DSSIM = (1 - SSIM)/2$) is also monitored for context with the results in [1]. SSIM loss implementation for the PyTorch framework was taken from [22].

3.7 Implementation details

3.7.1 Data scaling

All the image pixel data was stored in integer range of 0 to 256. The images were scaled to a range of -1 to 1 of real values. The scaling allows neural networks have hyperbolic tangent ($tanh()$) activations in the last layer. The *tanh* activation helps since the vast majority of pixels in the scaled data have values close to either -1 or 1 corresponding to pure black or pure white. Differently to regressing the original integer pixel colours, neural networks cannot predict and get penalised for predicting values that are darker than pure black or lighter than pure white. Advice taken from [23].

3.7.2 General training details

Details in this subsection are mentioned for reproduction purposes and are choices made based on popularity in neural network implementations and experience. The choices do not have any significant meaning in terms of the project direction or outcomes. Unless mentioned otherwise, these choices were applied throughout all the experiments in the project.

- The style bottleneck vector where applicable is chosen to be 512-dimensional, similar to the value of 500 that was used in [1].
- Weights for all networks were initialised depending on layer type as described in [24].
- Batch normalisation [25] was used between convolution or transposed convolution and activation layers.
- *LeakyReLU* [26] activations were used for all activations except for the last layer where *tanh* was used and fully connected baseline networks were simple *ReLU* activations were used.
- All neural networks were implemented using the PyTorch [27] framework.
- All training was done on a single GeForce NVidia GTX970 GPU.

- All errors were measured on the scaled data.
- Training was done in batches of 32 images for 300 epochs. 300 epochs were enough for all the baseline models to converge. Best resulting errors from any epoch were reported.
- Adam [28] was used as the optimiser for all training.

3.7.3 Architecture specifications

This subsection provides the exact specifications of the baseline architectures. The following notation is used to specify the layers in the neural networks. IN represents the network input. OUT represents the network output. $FC(x)$ represents a fully-connected layer with x neurons. $CL(x,y)$ and $TC(x,y)$ represent a convolutional layer and transposed convolutional layers respectively with x filters and kernel size of $y \times y$. LR, R, $TANH$ represent the $LeakyReLU$, $ReLU$ and $tanh$ activations respectively.

- Architecture 1. $IN \Rightarrow FC(1600) \Rightarrow TANH \Rightarrow OUT$
- Architecture 2. $IN \Rightarrow FC(1600) \Rightarrow R \Rightarrow FC(1600) \Rightarrow R \Rightarrow FC(1600) \Rightarrow TANH \Rightarrow OUT$
- Architecture 3. $IN \Rightarrow CL(256,3) \Rightarrow LR \Rightarrow CL(256,4) \Rightarrow LR \Rightarrow CL(256,4) \Rightarrow LR \Rightarrow CL(512,4) \Rightarrow LR \Rightarrow CL(512,5) \Rightarrow FC(1600) \Rightarrow R \Rightarrow FC(1600) \Rightarrow R \Rightarrow FC(1600) \Rightarrow TANH \Rightarrow OUT$
- Architecture 4. $IN \Rightarrow CL(256,3) \Rightarrow LR \Rightarrow CL(256,4) \Rightarrow LR \Rightarrow CL(256,4) \Rightarrow LR \Rightarrow CL(512,4) \Rightarrow LR \Rightarrow CL(512,5) \Rightarrow CT(256,4) \Rightarrow LR \Rightarrow CT(256,4) \Rightarrow LR \Rightarrow CT(256,4) \Rightarrow LR \Rightarrow CT(256,4) \Rightarrow LR \Rightarrow C(1,3) \Rightarrow TANH \Rightarrow OUT$

These specifications are still missing some information such as padding or strides. The exact specifications can be found in the project code.

Note that architectures 3 and 4 have the style bottleneck and can be separated into the style extractor and generator modules as described previously. The bottleneck is hard to notice in the specifications above since it involves the image downsizing after every strided convolution reaching the resolution of 1×1 and thus compressing the information to a single vector.

3.8 Baseline results

3.8.1 Quantitative results

The quantitative results can be seen in Table 3.1. Firstly, both the average character and nearest neighbour baselines are beaten in terms of all measured losses by even the most basic neural networks tried. Secondly, in the architectures 2, 3 and 4 that are

more powerful, the achieved results are quite similar. In particular, the results of purely fully-connected architecture 2 and fully-convolutional architecture 4 are close. The small difference between the results is in agreement with one of the conclusions of [16] that fully-connected networks and convolutional networks perform equally well in a similar task. Finally, the difference between optimising L1 error and L2 error is not significant in terms of DSSIM error. Quantitatively, it would be hard to decide which loss function is better to use for future experiments.

Table 3.1: Validation results of baselines and initial experiments on the task of generating character 'B' given character 'A' from the same font.

Architecture	**Optimised loss**	**Average validation error**		
		DSSIM	**L1**	**L2**
Average baseline	N/A	0.1722	329.5	307.3
Nearest neighbour baseline	L1	0.1101	168.5	239.6
Nearest neighbour baseline	L2	0.1120	171.5	242.2
Nearest neighbour baseline	DSSIM	0.1062	164.4	229.3
Architecture 1	L1	0.1014	147.7	181.9
Architecture 1	L2	0.1020	172.0	150.5
Architecture 2	L1	0.0933	136.9	169.7
Architecture 2	L2	0.0889	142.3	132.6
Architecture 3	L1	0.0900	135.6	155.3
Architecture 3	L2	0.0896	141.9	142.0
Architecture 4	L1	0.0868	131.0	152.5
Architecture 4	L2	0.0868	136.5	141.5

3.8.2 Qualitative results

Figure 3.2 displays how the generated images visually different among the tried architectures, baselines and optimised errors. A few trends can be noticed. Firstly, optimising L2 error produces blurrier images. The blurriness is especially noticeable on worse performing architectures but remains significant even in the case of architecture 4 on harder cases than displayed here. Secondly, the results produced by fully convolutional architecture 4 are noticeably better looking even though fully-connected network architecture 2 achieved similar errors in the quantitative evaluation.

Figure 3.3 displays the range of the quality of the images generated by the architecture 4 model optimised with L1 error. While there is still improvement to be made for the 4 bottom cases of standard fonts, the worst predictions come from the decorative font cases in the top 3 rows. The discrepancy could be attributed to the training set being

biased more towards standard fonts. However, even intuitively, predicting the decorative cases seems to be a much harder task since it involves more complex font stylistic attributes such as decoration or deformation of characters as opposed to the simpler cases which can mostly be described by a simpler stylistic attribute such as character thickness.

3.9 Discussion

The experiments in this chapter allow to make choices that narrow and direct the experiments in further chapters.

The fully-convolutional architecture 4 is chosen to be the base for further work since it achieves one of the best results quantitatively and produces the best looking results. In addition, the reduction in intermediate representation dimensionality as a consequence of using convolutional layers fits in conveniently with the goal of having a low dimensional bottleneck in the network. Having the bottleneck will allow to easier set up the transfer of the style from multiple inputs rather than a single one.

Optimising the L1 error is chosen as the pixel-wise optimisation objective. The qualitative results show that images produced by models that optimised L1 error have more contrasting outlines as opposed to blurry edges present in images generated by L2 error trained models.

Some trends in the produced image quality can be noticed. In particular, there is a big discrepancy in the quality of results for standard and decorative fonts. The problem with decorative fonts is often the decorations that have not been observed in the training set. Since similar decorations have not been observed in the training set, it is hard to expect the model to generate them during a test prediction. The problem of failing on decorative fonts is inherent to all the models that try learn a model of the shape of the characters. As an example, the approach in [11] did not try to fit a model for character shapes directly and thus can handle some font decorations.

Another important thing to recognise in Figure 3.3 is the fact that for the worse cases of more decorative fonts the models optimised on L1 error produce blurry, garbled and unrealistic results. The problem is with pixel-wise losses in general - similar results can be observed optimising the L2 error. A more graceful and practical mode of failure would be to produce stylistically simpler but still realistic characters if the style transfer could not be done well enough. The problem of realistic outputs will be explored later in the project.

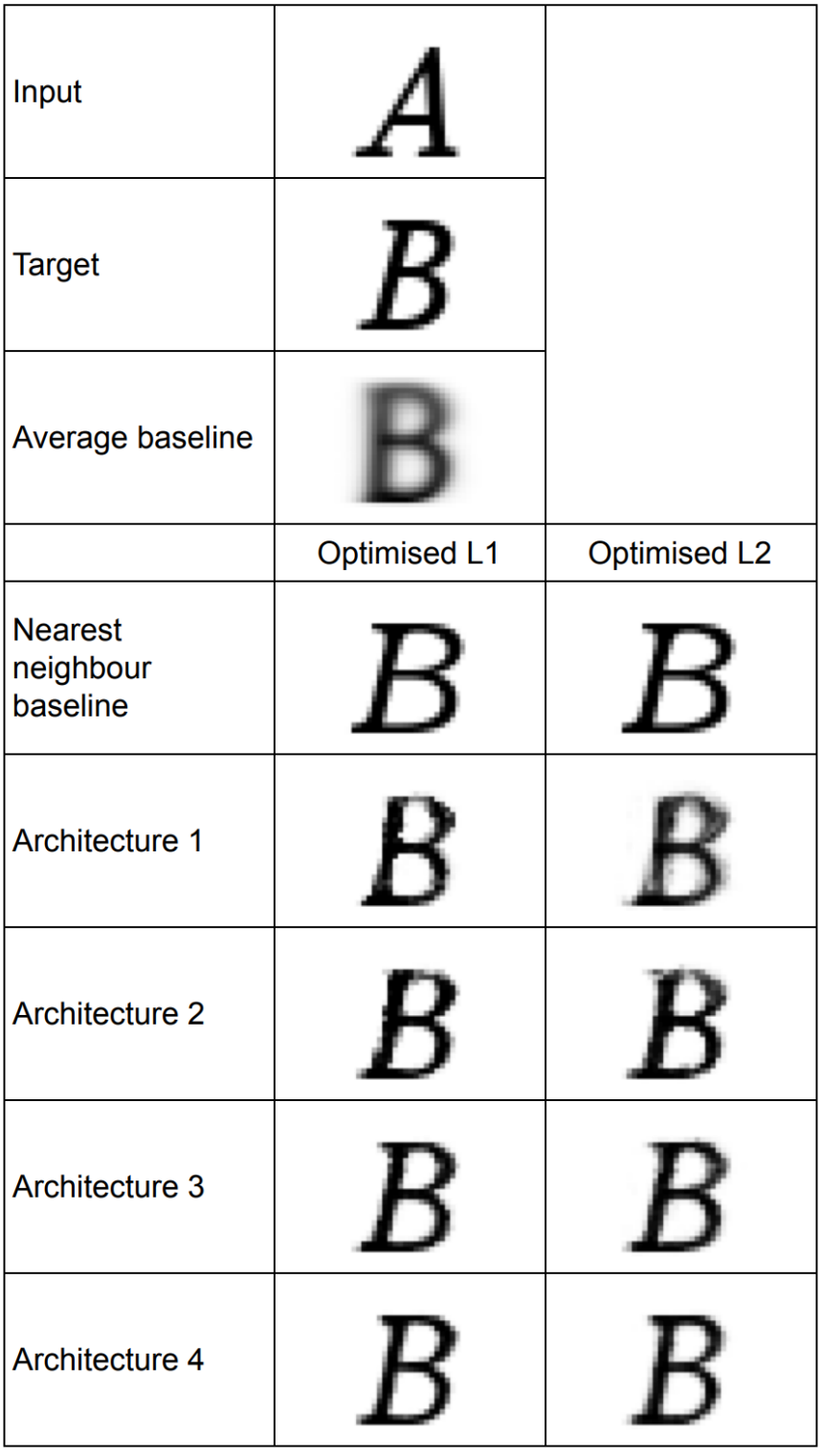

Figure 3.2: A single validation case showcasing the differences between the outputs of different architectures and optimised losses.

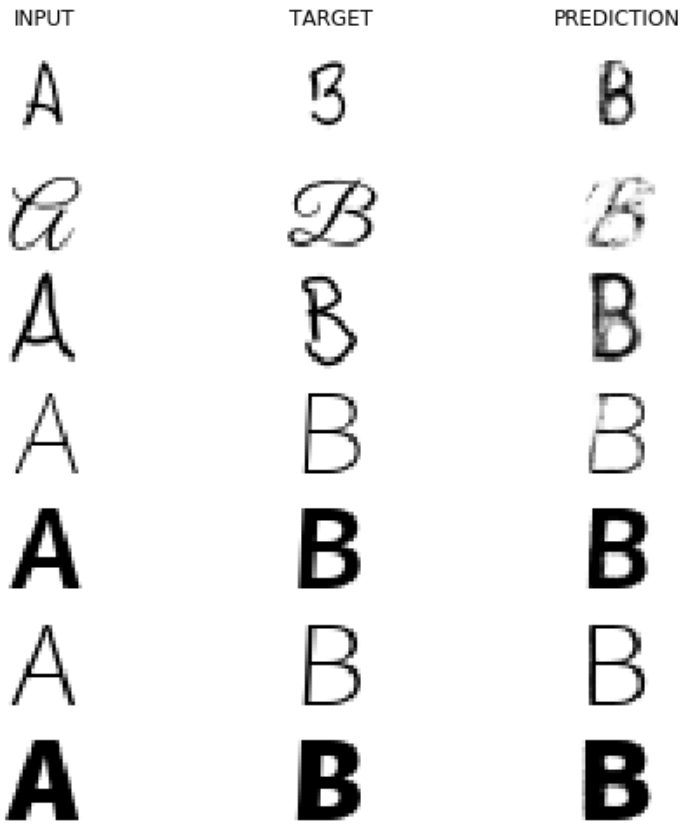

Figure 3.3: A few randomly selected validation cases generated with architecture 4 optimised on L1 error.

Chapter 4

Generalising to multiple inputs and outputs

The model developed in the previous chapter was designed to take an image of one character and generate an image of another character as the output. Sections in this chapter describe the modifications to obtain a model capable of taking any number of characters as an input and producing any number of characters.

4.1 Multiple outputs

The case for multiple outputs is trivial to achieve. The simplest thing that can be done is to train 52 different models of the same structure as described in the previous chapter for generating each character (26 letters of the alphabet, uppercase and lowercase). At test time, only the models corresponding to the characters needed could be used.

An optimisation can be made given that it is possible to split our architecture into two parts, the style extractor and the generator. Since the architecture chosen in the previous chapter includes a fixed dimensionality bottleneck, the network part that generates the bottleneck activation values can be considered the style extractor and the network part that takes the bottleneck activation values as the input and generates the output image can be considered as the generator.

In the case of multiple outputs but a single input, the style extractor can only be used once - to obtain the bottleneck activation values, that is, the style vector representation. The style representation then can be used with multiple generators that generate different characters. The reuse of style representation allows to train one style extractor instead of 52.

A similar optimisation can be applied to the generator as well. The idea is to have a generator that can generate any needed character. To allow this, a generator is modified to take an additional content switch input, exactly like it is done in [1]. The content switch can then be set to generate a character that is desired.

Having a shared style extractors and generators handling multiple characters should bring the advantage of multi-task learning [29]. Multi-task learning is a result in machine learning that training a model to do multiple tasks at once, in this case generating multiple characters, can result in better performance than training different multiple models for the tasks separately. This hypothesis is tested with experiments described in the following subsection.

4.1.1 Multi-task learning experiments

The original task of generating character 'B' given character 'A' is extended to generating characters 'B', 'G', 'e' and 'm'. The cases are tested:

- 4 different instantiations of the model chosen in the previous chapter each for handling the generation of characters 'B', 'G', 'e' and 'm' given 'A' respectively.
- A single model modified with a content switch in the generator to be able to generate any of the four characters.

Architecture	**Validation error**	
	DSSIM	**L1**
'A' to 'B' pair model	0.0863	129.3
'A' to 'G' pair model	0.1049	149.2
'A' to 'e' pair model	0.0732	104.5
'A' to 'm' pair model	0.1101	168.1
Average of pair models	0.0936	137.7
Single model with a content switch	0.0940	137.4

Table 4.1: Comparison of training 4 models for different task and a single model with 4 tasks.

The Table 4.1 shows the results of multi-task learning experiments. There are only insignificant differences between the results of the single multi-task model and the average results of the four single-task models. The results point towards multi-task learning not helping or the experiment set-up being not big enough. It might be the case that the effect of multi-task learning would be noticed if the experiment was performed with more tasks - more possible output character classes. The results are in slight disagreement with one of the conclusions of [16] that found small 5%-6% error improvements when transitioning to multi-task training. Practically, there is also no significant difference in training time between the two cases. Nevertheless, it is more convenient to have a single model since it simpler to load, save, deploy and has fewer total parameters. The single model with a content switch is used for all further experiments.

4.1.2 Comparing to [1]

The model constructed in the previous section is capable of performing the task of generating any set of characters given a single character. The multiple-output capability allows to set up a task measured in [1], generating 52 letters and 10 digits given the character 'A'. The authors of [1] provided their source code for preprocessing and training. While training their model would require setting up a completely different environment, it allows to check how exactly their evaluation of DSSIM scores was done since that was not mentioned in the paper. The key difference is that in this project the DSSIM scores are measured on images scaled to a range $[-1, 1]$ while [1] measure their DSSIM scores on images scaled to a range of $[0, 1]$. To have appropriate comparisons, the DSSIM scores in this subsection are of images scaled to a range of $[0, 1]$.

4.1.2.1 Quantitative comparison

The Table 4.2 shows the results of applying the model to the task used in [1]. There is a large improvement in terms of DSSIM scores. Note that the model described in this section optimised L1 error as opposed to directly optimising DSSIM score as was done in [1].

Method	**Validation errors**		**Test errors**	
generating all characters given 'A'	**L1**	**DSSIM**	**L1**	**DSSIM**
Base model	126.0	0.0812	135.7	0.0895
Modified VAE with adversaries [1]	-	0.0892	-	0.0990

Table 4.2: Error values of comparison between [1] and the model described in this section. Base model refers to the model described in this section.

4.1.2.2 Qualitative comparison

Figures 4.1 and 4.2 show a few sets of character images generated by [1] and the model described in this section respectively. While the comparison is quite subjective since the specific fonts used for Figure 4.1 were not known, the base model approach exhibits images with noticeably more contrast and less blur.

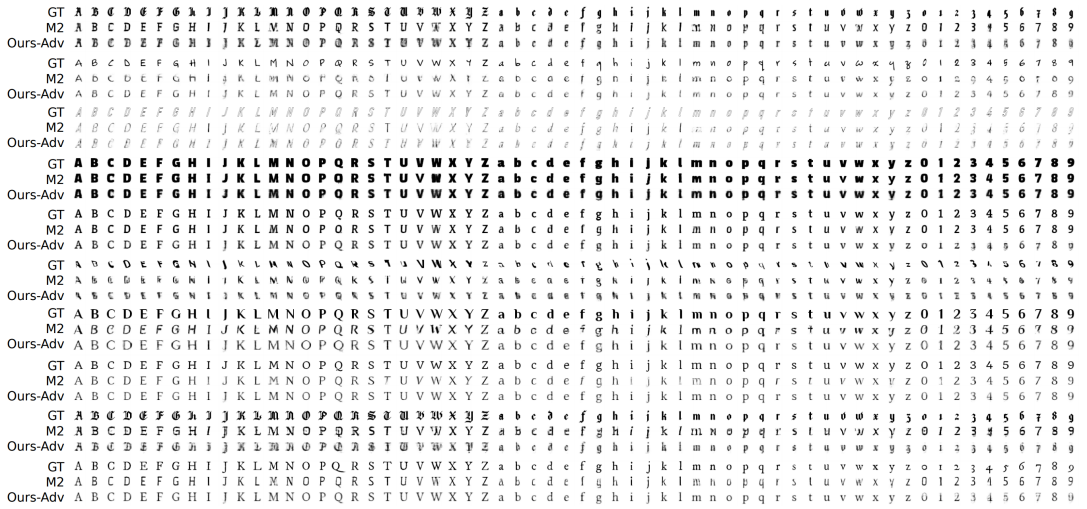

Figure 4.1: Figure taken from [1]. GT refers to the ground truth - true targets. M2 refers to an approach from [30]. Ours-Adv refers to the best performing approach of [1]. A random sample of fonts from the test set.

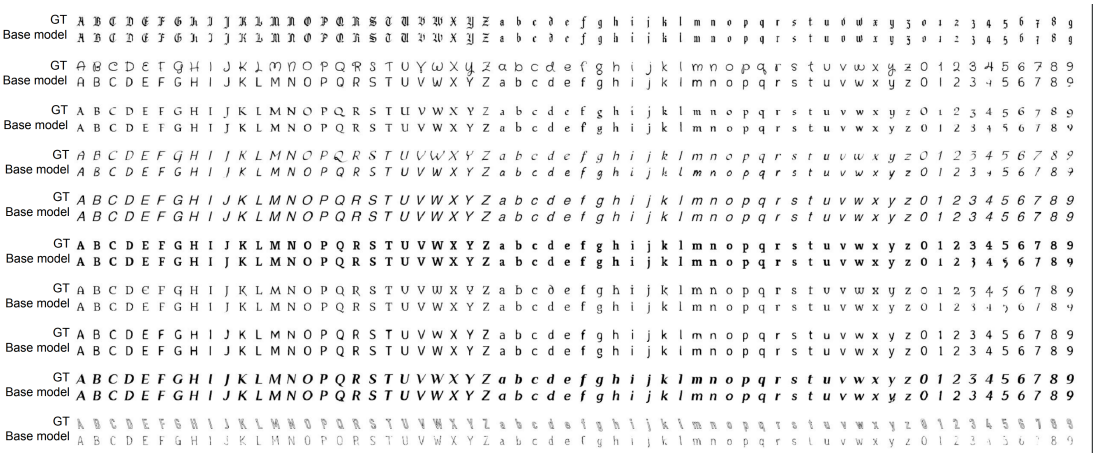

Figure 4.2: Base model refers to the model described in this section. GT refers to ground truth. A random sample of fonts from the test set.

4.1.2.3 Key improvements

The result improvements seem to be significant both quantitatively and qualitatively. There are two key contributions that seem to have caused to the performance improvement.

- Using convolutional neural networks instead of fully-connected networks. CNNs are more suited to image generation. While this was slightly noticeable in the experiments of the previous chapter, the difference is exaggerated when the task is generalised to many possible outputs.
- A more powerful model. The neural network described in this chapter uses

1-2 orders of magnitude more weights than the approach in [1]. The large increase together with the fact that CNNs use weights more "efficiently" for image generation result in a vastly more powerful model. A powerful model is needed because we require the generation of 62 different characters. While the characters are similar to a human, they might be quite different tasks for a neural network, especially a fully-connected one.

It is worth to note that due to the mentioned advantages, the base model performs well with a relatively simple architecture. Applying the contributions of [1] could improve the model further.

4.2 Multiple inputs

The architecture capable of generating any character described in the previous section needs to be modified to be able to handle multiple inputs. The task of multiple inputs is harder than the task of multiple outputs since it is not possible to just have multiple style extractors. Whatever the style extractor architecture looks like, it needs to produce a single style vector that then the generator can use to produce the outputs. The current style extractor architecture can be run on each input separately and produce n of style vectors where n is the number of inputs in the given case. Therefore, the task becomes combining the multiple style vectors into one. This section explores a few ways of doing it.

The simplest way of combining multiple style vectors into one is to take the average of the multiple vectors. A few other ways to achieve the single style vector could also be to take the sum of the vectors, take the per-dimension maximum of all the vectors. A more advanced method would be to train a separate reducer network that can combine two style vectors into one. Given the reducer network, it would be possible to combine any number of style vectors into one similarly to how the sum operator can be used to combined multiple numbers into one sum.

4.2.1 Multiple input experiments

The experiments in this subsection try to determine the best method of combination by comparing the results on a modified task. The modified task is the generation of characters 'G' and 'm' given any number of characters from the set of 3 characters 'A', 'e' and 'n'. That is, in any particular instance the model might need to generate 'G' or 'm' given any 1, 2 or 3 characters from the input set in any order.

To accelerate training, the weights from the previous experiment generating all characters were used as a starting point for training. In addition, the model was trained only on a subset of all possible input-output combinations.

The Table 4.3 shows the results of the multiple input experiments. The difference among the proposed choices for the style accumulation seems to be small. Qualitatively, there was no noticeable difference among the results. However, the accumulation by

averaging did perform the best and is faster than the next best accumulation method of learned neural network. Therefore, averaging is used as the choice of accumulation in the proposed best model.

Accumulation method	Validation error		
	DSSIM	L1	L2
Sum	0.1024	148.4	191.9
Average	0.1001	145.1	187.3
Per-dimension maximum	0.1030	148.5	193.1
NN learned accumulation	0.1018	146.5	190.6

Table 4.3: Results of multiple input experiments for the task of generating characters ‘G’ and ‘m’ given any subset of characters ‘A’, ‘e’, ‘n’.

The Figure 4.4 shows the comparison of results obtained by evaluating different datasets on the model trained with averaging style accumulation. The results show that adding additional inputs does not always improve the performance. However, without have any prior knowledge of there is no way other than guessing to determine which inputs will work the best and thus it is beneficial to use as many inputs as possible.

Inputs	Validation errors	
generating ‘G’, ‘m‘	L1	DSSIM
‘A’	158.4	0.1083
‘e’	164.7	0.1123
‘n’	157.4	0.1082
‘A’, ‘e’	160.2	0.1083
‘A’, ‘n’	157.2	0.1075
‘e’, ‘n’	161.7	0.1115
‘A’, ‘e’, ‘n’	157.2	0.1089

Table 4.4: Results from evaluating the previously trained model on datasets with different fixed inputs. The results are slightly worse overall compared to Table 4.3 because the datasets were generated exhaustively and the model was not retrained.

4.2.2 Style space exploration

Figure 4.3 shows an example of style interpolation. Two styles are extracted from stylistically different ‘A’ characters. The style is then gradually interpolated between the two extracted style vectors displaying what the generator module generates at each

step of the style interpolation. The slow morphing of the generated characters into each other empirically shows that the style extractor does infer and encode latent information such as letter thickness, slant and presence of serifs.

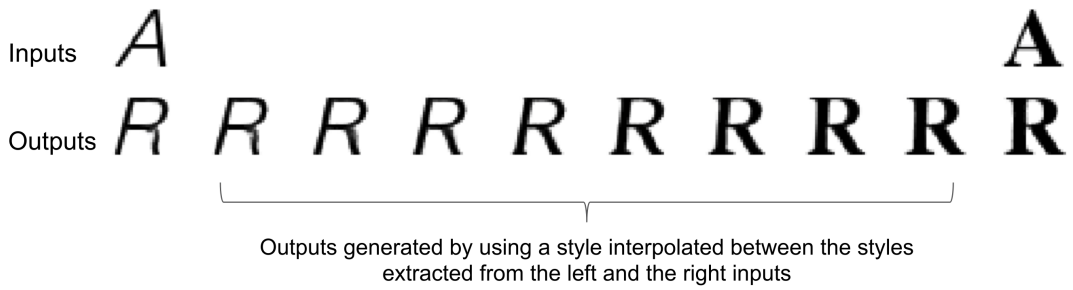

Figure 4.3: The interpolation of style between the styles extracted from left and right inputs. Note the gradual change of slant, boldness, serifs and right leg of character 'R'.

Figures 4.4 and 4.5 show examples of style arithmetic where by manipulating extracted styles it is possible to combine them and generate unseen characters that have or do not have specified attributes.

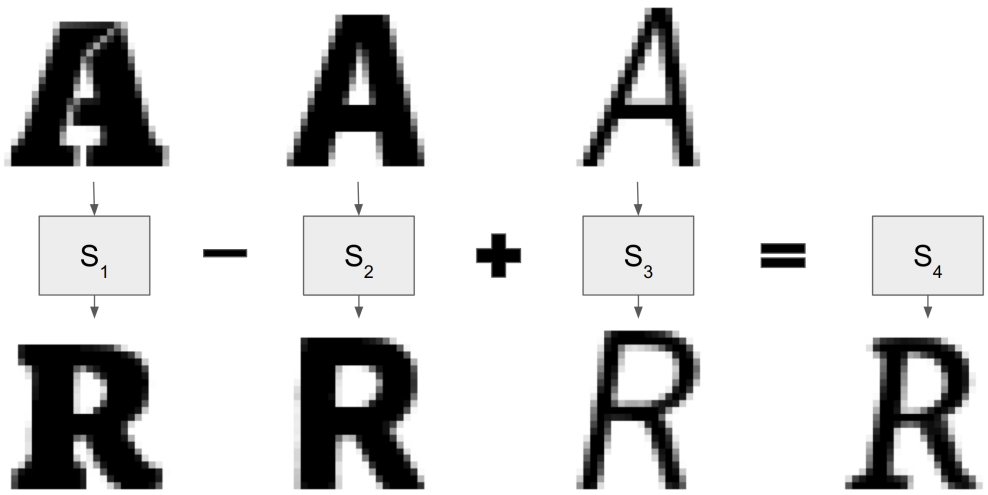

Figure 4.4: S_1, S_2, S_3 refer to the styles extracted from the top row of 'A' characters. $S_4 = S_1 - S_2 + S_3$. Intuitively, bold with serifs style minus bold style plus thin italics style equals thin italics with serifs style.

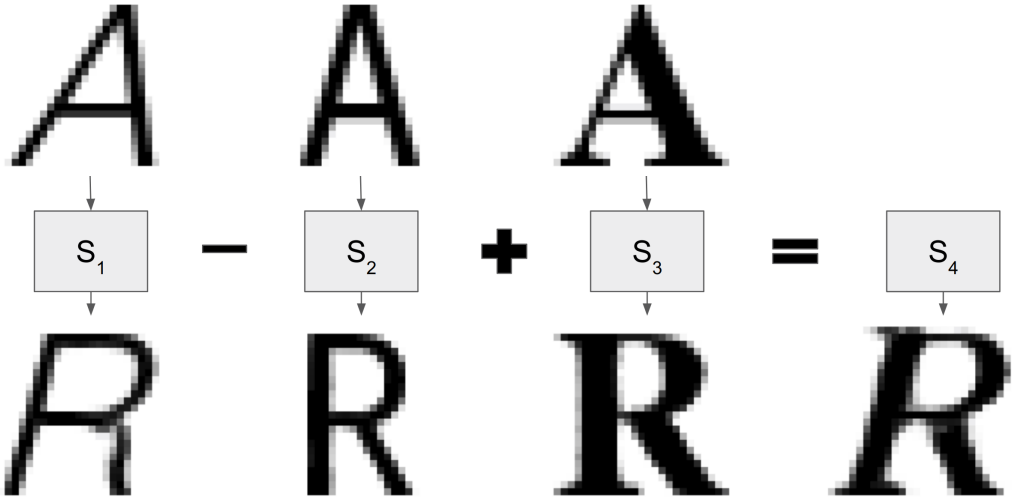

Figure 4.5: Another example of style arithmetic. Intuitively, thin italics style minus thin style plus bold with serifs style equals bold italics with serifs style.

4.3 Discussion

The model described in this chapter will henceforth be referred to as the base model. The adaptations in designing the base model for multi-output and multi-input support achieve the goal of this project. The goal was to design a model capable of taking in any subset of character and generating any other subset of characters and the base model is capable of performing the said task. The high-level architecture of the base model can be seen in Figure 4.6.

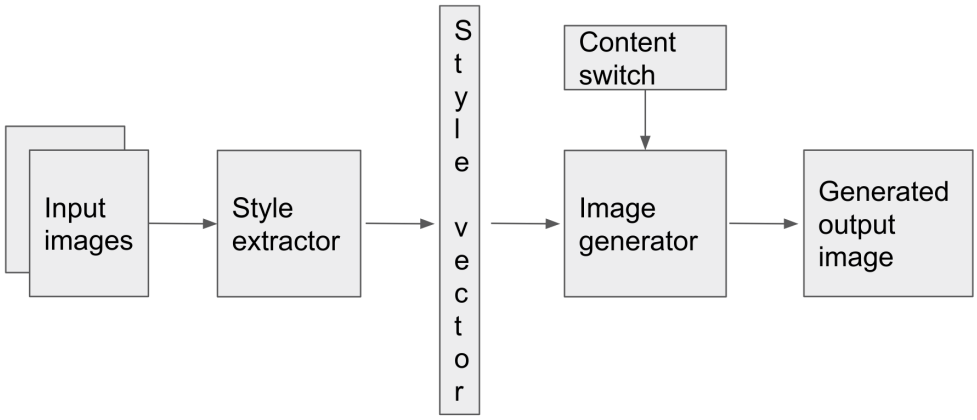

Figure 4.6: The composition of the base model.

There are two major branches of possible further work.

4.3.1 Style extraction

Improving the accumulation of styles extracted from multiple inputs. The accumulation is especially important in the cases of more decorative fonts where certain font features can only be seen in certain characters. Informally, the model needs to learn to pick and choose which font attributes it can learn from which characters in order to make the best guess at what the required output could look like.

Some work was done to try and provide a more focused extraction. The model was trained having separate style extractor parts for each possible character in the input as well as have a single style extractor but adding an input switch witch provides the information from which character the model is extracting the style from. However, the experiments showed no improvement. Improving the style extraction part of the process is still an open-ended part of the problem and could provide a major boost in performance if its design would be improved.

4.3.2 Image quality improvements

The second branch includes the optimisation objectives, evaluation and the subjective quality of the outputs. The base model strictly optimised for L1 error and the results were compared using DSSIM scores. Both measures are convenient for quantitative analysis but are flawed in evaluating how close the result is from something that could be reasonable put in printed text.

The next chapter explores one of the ways of improving the optimisation criteria to try and achieve more plausible results.

Chapter 5

Adversarial training

It was noted in the previous chapters that the outputs of the tried models are often not realistic. That is, the outputs look blurry, do not have sharp edges or the low resolution artefacts are different than they are in the target images. A few examples of the unrealistic behaviour can be seen in the Figure 5.1

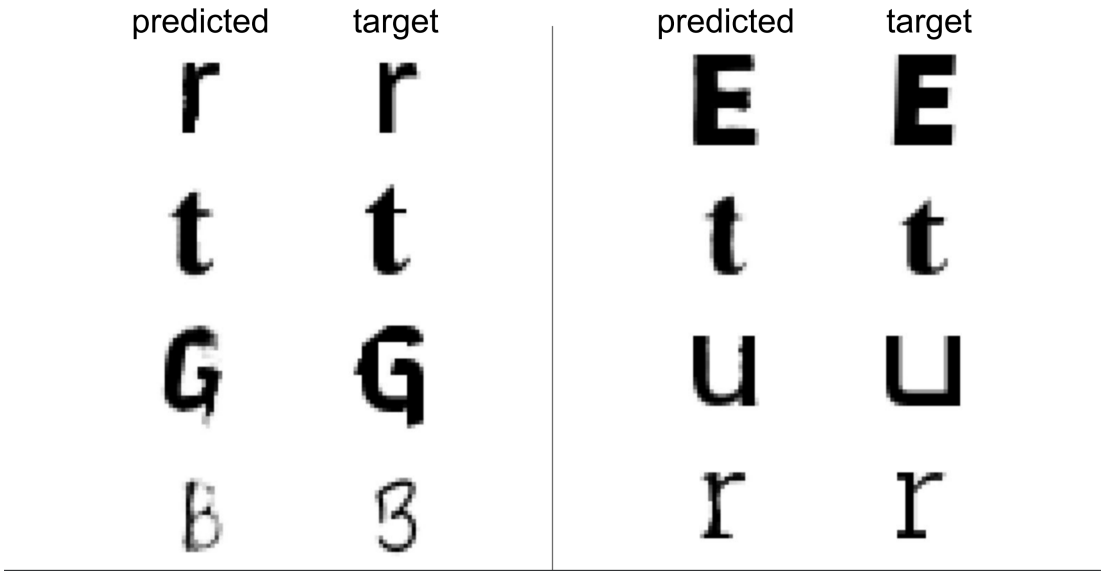

Figure 5.1: Base model refers to the model described in this section. GT refers to ground truth. A random sample of fonts from the test set.

From a practical perspective, it would be desirable if the model could generate images that are always plausible character images even if not stylistically accurate. This chapter explores the suitability of adversarial training to try and enforce the constraint of always generating plausible images. Informally, adversarial training pits two models against each other. The two models are known as the generator and the discriminator. The generator generates images while the discriminator is given images generated by the generator intermixed with the images from the training set and tries to predict where

each image came from. The intention is that eventually the generator learns to generate realistic images which the discriminator can no longer distinguish from the images of the training set. The key element of generative adversarial networks (GANs) is that the models are both optimised to improve with respect to each other. Informally, the generator tries to fool the discriminator into predicting that the generator's output came from the training data. The discriminator tries to find the features that can distinguish between the real images in the training data and the generator synthesised images.

More formally, the training is split into two steps for each minibatch of data. Firstly, the discriminator is trained to minimise the classification cross entropy loss $Loss = -(y\log(p) + (1-y)\log(1-p))$ where y is either 0 or 1 depending on whether the image was generated or taken from the training set and p is the probability predicted by the discriminator that the image was taken from the training set. Secondly, the generator is trained to maximise the loss stated before. The set-up results in a zero-sum game where the best response of each model is to get better at its task via training.

5.1 Initial set-up

Traditionally, generative adversarial training is an unsupervised method. Note no mention of targets in the description above. In this project, we use conditional adversarial networks [7] where both the generator and the discriminator take an input and must make decisions conditioned on that input.

In the simplest GAN set-up for this project, the base model developed previously can be considered to be the generator. The only change needed to the generator is to inject some noise into the generator input. While this is necessary in the unsupervised setting to obtain a variety of generator outputs, it is not as important in the supervised setting. However, it helps to prevent the generator from simply memorising the training set images. This is done by adding dropout [31] to the style vector similarly to how it is done in [7].

The discriminator needs to be designed separately. Following the conditional set-up in [7] the discriminator would need to take the same input as the generator. In base model case the input is the variable number of images of given characters. Therefore, the discriminator must have the same style extractor part to take in the variable number of input images. In addition, the discriminator must take in the output of the generator as the input to guess in which way it was produced.

The requirements described above allow to set up the discriminator to be of the following architecture. Firstly, the discriminator features exactly the same style extractor part as described previously in the base model. Secondly, instead of the image generator part of the base model, the discriminator has a convolutional classification module that takes the extracted style and the generator output as the inputs and produces binary classification as the output. The Figure 5.2 can help understand the similarities and the differences between the compositions of the generator and the discriminator.

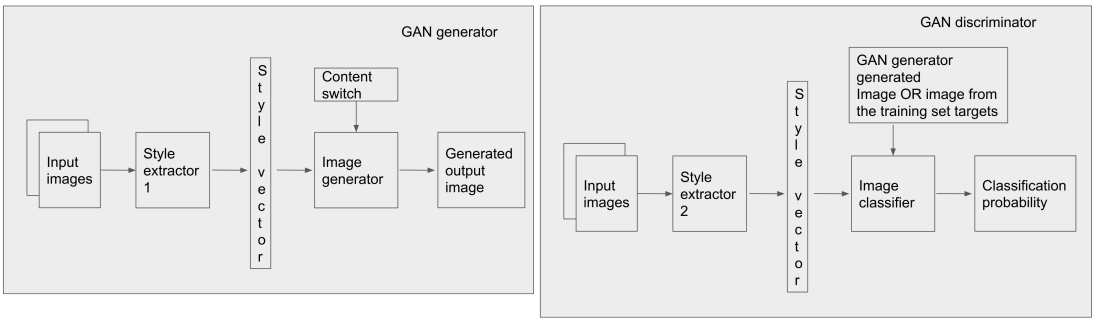

Figure 5.2: The compositions of the generator and the discriminator for the GAN set-up. Note that the generator is the same base model described in the previous chapter.

5.2 Initial GAN experiments

The set-up described in the previous section was trained to optimise both, the GAN classification loss and the usual supervised training L1 loss following [7]. Unfortunately, GAN training is notoriously unstable and the simple set-up usually ended up diverging to the state where the generator and the discriminator try to trick each other but the generated images do not look anything like the targets. An example of the divergent behaviour can be seen in Figure 5.3.

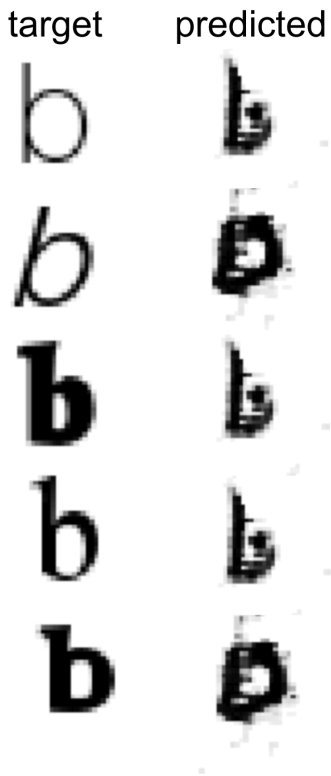

Figure 5.3: Degenerate cases of the GAN training results. Note that some of the outputs are the same, meaning that the generator is at least partially ignoring the conditional input which was the letter ‘a’ in these cases.

5.2.1 GAN stability tricks

GAN training instability is a common problem and an active research area. There is a lot of empirical knowledge on how to improve the training stability by making small adjustments to the training procedure. Some of the tricks commonly used in practice are described in [23]. Some of these tricks were applied in follow-up GAN experiments:

- Occasionally flipping the labels when training the generator. That is, with small probability optimise the same loss as the discriminator.
- Avoiding sparse gradients in the layers used for the generator and the discriminator models. *LeakyReLU* activations were used instead of *ReLU* and all downsampling was done using transpose convolutions.
- Adding noise to the inputs of the discriminator to prevent it from memorising exact images.

The stability tricks seem to help somewhat, however the training is still largely unstable and it is hard to estimate to what extent the tricks are helping if at all.

5.3 Jury of discriminators

One of the intuitions behind the instability of GAN training is that it is too easy for the discriminator and the generator to exploit the architectural weaknesses of each other. For this reason, it was tried to have multiple discriminators instead of just one. The key part is having discriminators of different architectures and even different inputs and outputs.

This set of discriminators was tried:

- The standard discriminator as described before implemented as a convolutional neural network. Discriminating between generated images and training set targets given training set input images.
- The standard discriminator as above but implemented as a fully-connected neural network.
- Unconditional discriminator that discriminates between generated and real images with no other information available. Implemented as a convolutional neural network.
- PatchGAN as described in [7] where each 5×5 window of the generated image is classified separately conditioned on the input images. Implemented as a convolutional network.

The idea of multiple meaningfully different discriminators is that it would make it harder to find and exploit degenerate cases that fool all of the discriminators at once.

The training while using a jury of discriminators did prove to be more stable and the images did not diverge to degenerate cases. Some of the example outputs can be seen in Figure 5.4. However, the images still exhibit some of the artefacts shown in the beginning of the chapter. In most cases, the results look worse than the outputs of models without adversarial training.

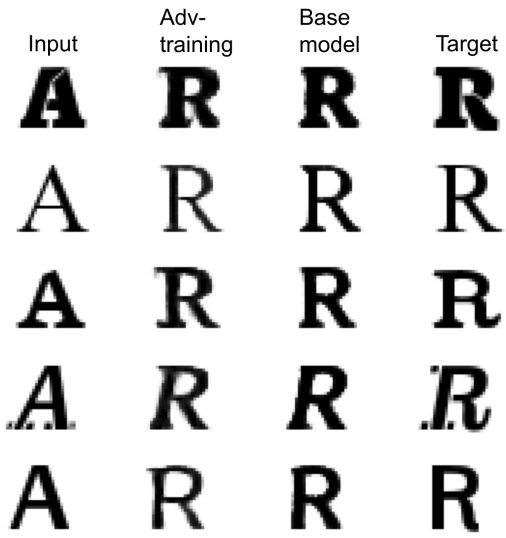

Figure 5.4: Adv-training refers to base model trained with an additional GAN loss. Comparison of results produced by the base model and base model with adversarial training.

5.4 GAN adaptations

To improve the previously described GAN set-up, two observations can be made:

- The changes required to remove the artefacts shown in the beginning of the chapter are often tiny. Even a single pixel or a small blurry spot can make the image look unrealistic for the human eye. However, these changes have little impact in terms of the L1 loss function. As a result, it must be possible to fix the unrealistic artefacts while keeping the L1 loss mostly unaffected.

- The problematic artefacts have little to do with any style or other information extracted from the input image. It is likely that the conditional discriminators are not necessary since it is usually trivial to tell whether a generated image is realistic without having to see what the input images of characters looked like.

Both problems can be addressed. To prevent adversarial training from making major image changes, the L1 loss can be weighted much higher than the adversarial training loss. The discriminator does not have to be conditional. The discriminator can be simplified by removing the conditional input and leaving just the generator generated image or an image from the training set targets as the input to the discriminator.

However, there are other reasons why the current adversarial learning set-up is not working as well as it could.

5.4.1 The problem with conditional GAN training

So far, the addition of adversarial training did not bring any improvements to how realistic the generated images look. There are major differences between the application of this project and the conditional adversarial set-up that was followed in [7]. More specifically, the tasks that the paper was applied to were image translation to corresponding targets such as image colourisation, converting aerial photos to maps or converting edge sketches to photos. In all the cases the input images and the output images match structurally. Therefore, conditioned convolutional discriminator network is easily able to pick up structural differences such as presence of edges at the same positions in input and target images. For example, if there was a face in the centre of an black and white input image, there must be a face in a colourised output image. Therefore, the conditioning for GANs in this case is more direct.

In the case of this project, the inputs and outputs are not corresponding. In addition to there possibly being multiple input images, none of the inputs and outputs match structurally. Image of 'A' has some information of how an image of 'R' could look like but it is not nearly as direct as in the case of corresponding image translation tasks in [7].

The discriminator inputs and outputs as well as the composition needs to be changed to provide more relevant context.

5.5 Unconditional order discriminators

The discriminator composition is modified in the following way. Firstly, the discriminator is made unconditional as mentioned before. That is, the discriminator no longer takes the input images as one of its inputs. Secondly, the discriminator takes both the generator generated image and the target image as its inputs at the same time in a random order. The discriminator then predicts the order of the images. That is, it predicts which image was the target and which image was generated by the generator.

This modification to the prediction of the discriminator allows it compare corresponding images. For example, it should be able to learn behaviour such as "if one image has a fuzzy edge at the location where the other image has a straight edge, the one with the fuzzy edge is likely generator generated". The composition of the modified discriminator can be seen in Figure 5.5. Since the image comparer can be implemented as a convolutional network, it has the advantages of conditional image translation [7] described in the previous subsection. Note that the idea of the discriminator modification is not original but I could not find a source to cite.

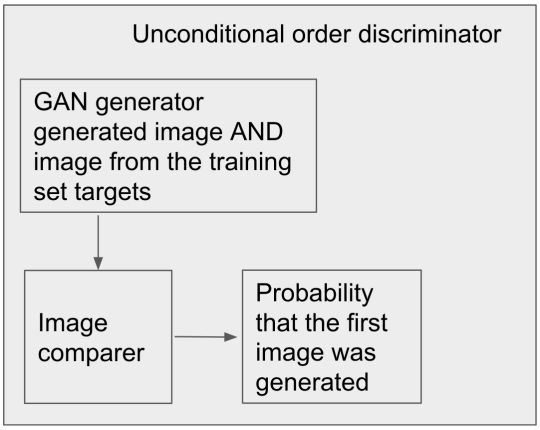

Figure 5.5: The composition of the unconditional order discriminator.

5.6 Results and discussion

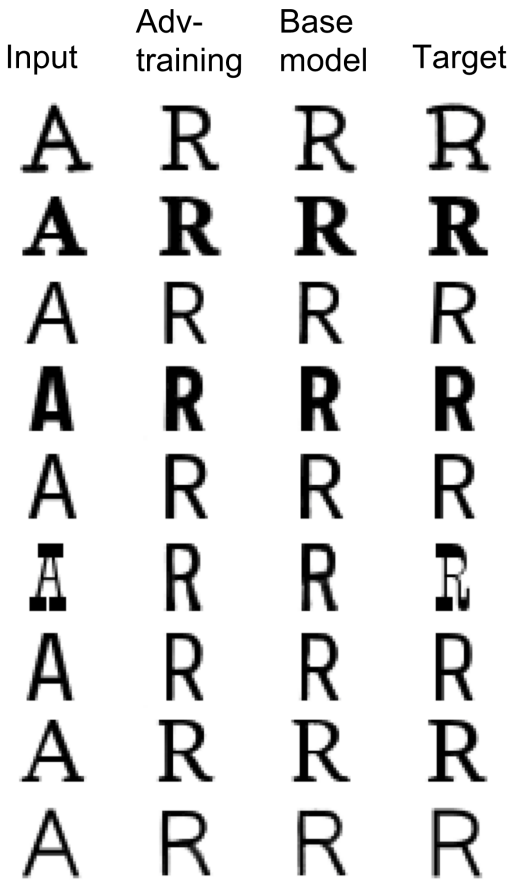

Figure 5.6: Validation samples of the results generated by the base model and the base model trained with an additional GAN loss. Samples where there was no noticeable difference between the outputs of the two models were omitted.

The Figure 5.6 compares the results from the previously trained base model and the base model trained with an additional GAN loss term coming from the unconditional order discriminator. As enforced, the results do not differ by much. It requires careful inspection but for most problematic cases, the edges look slightly cleaner in the results of the model with the additional GAN loss. Rarely the result of the base model is the more realistic image, however in about a third of the cases where the base model generated image was already good, there is no noticeable difference.

The images generated by the model with additional adversarial loss were not significantly degraded in terms of the average DSSIM score. The results for more decorative fonts were changed significantly but with no improvement either in terms of the DSSIM scores or visual quality.

The adversarial training in this case was used only to clean up the images and it helped slightly. The effect would likely be better if the training data was of higher resolution and did not contain as much shadowing and blur itself.

It remains for future work to explore if adversarial training could be applied more aggressively to tackle the problem of generating plausible characters even for decorative fonts and to regularise the generated character shapes further than just making small modifications.

Chapter 6

Approximating image functions by neural networks

The results of the previous chapters still leave a lot of room for improvement of generated image quality. Undoubtedly, part of the problem is the low resolution data used for training. The jagged or blurred lines in the training targets might limit the models to fit to those artefacts rather than model the smooth character shapes. This chapter presents experiments with a different representation of images that might avoid the pixelated nature of images generated in the previous chapters.

6.1 Images as functions

The traditional image representation that is usually used with neural networks and has been used in this project so far is the pixel representation. Pixel representation consider images as grids of greyscale values. That is, each image is represented by a 2-dimensional array of numbers representing the whiteness of each pixel.

The experiments in this chapter use a representation where each image is represented by a function $f(x,y)$ which maps location coordinates x,y to a whiteness value in that location of the image. x,y are both real numbers in range $[0,1]$. Such representation means that once the image representing function is found, the image can be infinitely scaled. That is, since x,y are real numbers, it would be possible to render the image to a pixel representation at any resolution.

Neural networks are universal function approximators which means that they could be capable of representing images as such functions. Since the data is only available at the low resolution of 40×40, the training of a neural network to represent images as functions would require the network to interpolate between the 1600 given points. It depends on the internal neural network representations whether working with functional image representations could provide higher quality high resolution images than simply upsampling the data from the outputs of the traditional neural networks used in the previous chapters.

Some similar work in generating high resolution images from latent vectors has been done in [32]. The idea and some implementation details have been taken from there.

6.2 Experiments and results

The initial experiment was set up for a simple autoencoding task where a fully-connected neural network takes the style vector extracted from an input image as in the previous chapter as well as coordinates of the value to generate. The network then is trained to reconstruct the greyscale value at supplied location in the input image. The network composition can be seen in Figure 6.1

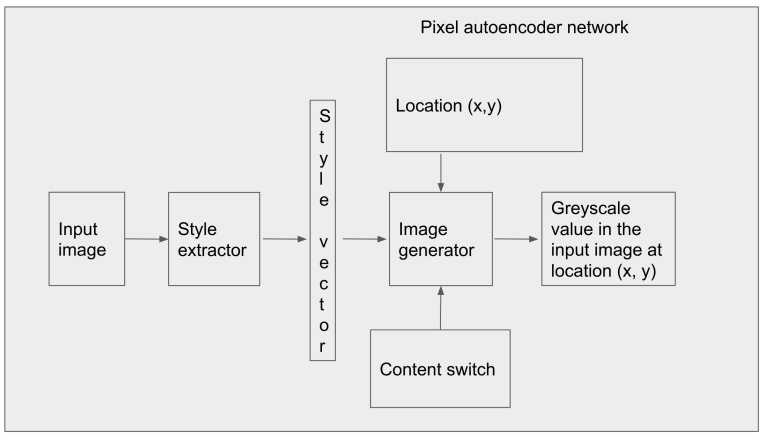

Figure 6.1: The composition of the pixel autoencoding network.

Training such network allows to observe the behaviour of the network approximating image as a function representation. The key thing to check is whether the trained network generates meaningful images at higher resolution rather than just at the 40×40 resolution that is was trained at.

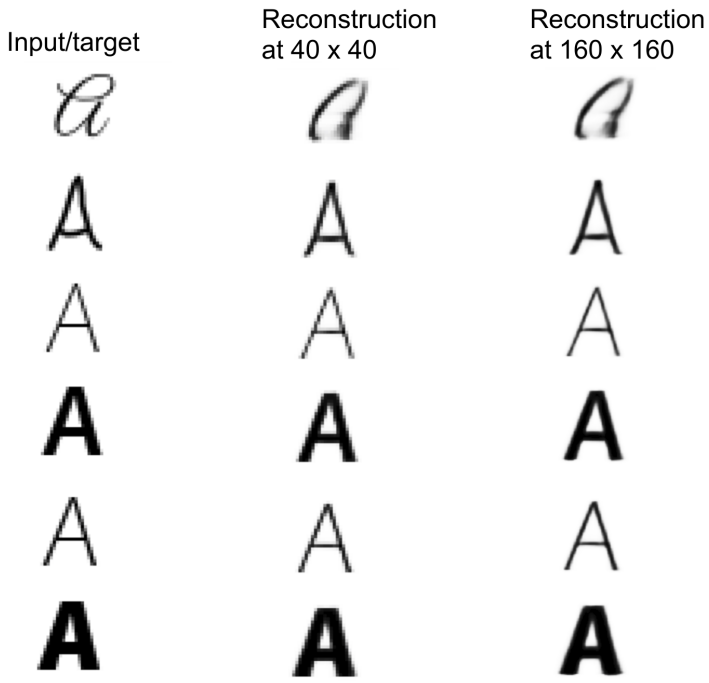

Figure 6.2: Validation reconstruction samples from the trained pixel autoencoding network. Left column of images shows the images the network was trained on. Middle and right columns showcase the network capability to reconstruct images at varying resolutions.

Figure 6.2 shows some of the reconstruction samples produced by the trained network. The reconstruction quality is poor for more unique fonts. However, the high resolution outputs for standard fonts mostly avoid the pixelated artefacts and produce smooth looking shapes. The results look promising in terms of practicality. The smoother shapes is something more likely to be useful in practice.

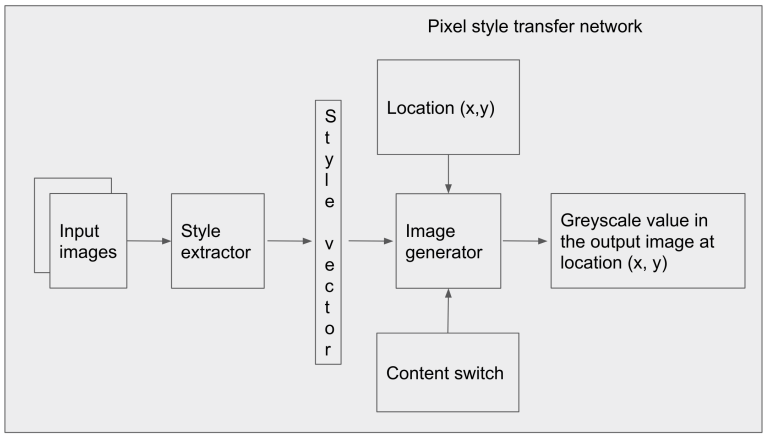

Figure 6.3: Composition of the base model adapted for the image function representation.

For further experimentation, the base model was adapted for the image function rep-

resentation. The resulting composition can be seen in Figure 6.3. Due to the image generator component being a fully-connected network instead of a convolutional neural network, the whole model is more difficult to train. The training often encounters the vanishing gradient problem and gets stuck at generating empty images. These problems are likely solvable but would require further iteration on the lower level details of the network. Therefore the style transfer network for image function representation was only trained on the small task of generating character ‘R’ given character ‘A’.

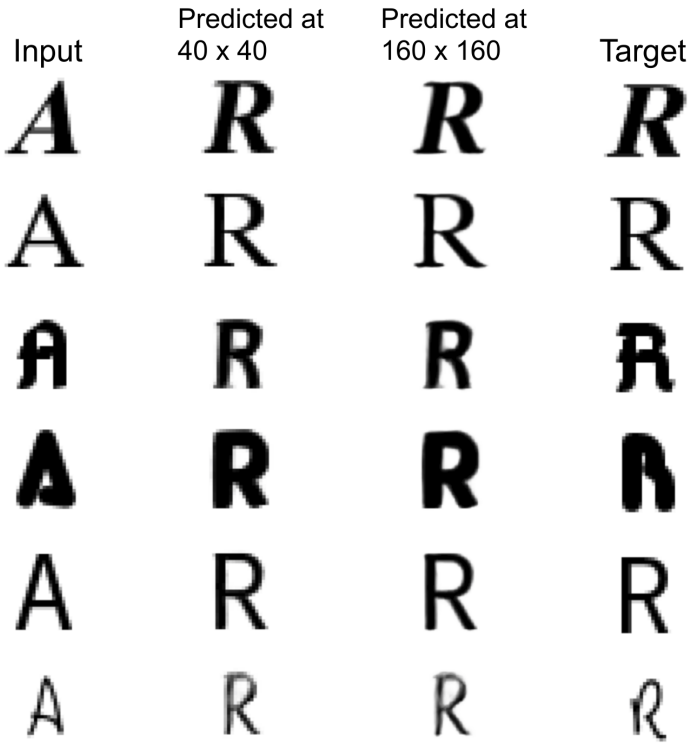

Figure 6.4: Validation samples of the inputs and outputs to the base model adapted for the image function representation.

Figure 6.4 shows the quality of the results. In terms of the DSSIM scores, the images predicted at 40×40 resolution reached approximately the same values reached by the base model in the previous experiments. To further show the difference between predictions at different resolutions, Figure 6.5 shows one of the samples at a larger resolution difference. The high resolution images are in general smoother and look more natural and sharper but still not as sharp as real vector images would be.

Figure 6.5: Left image shown a validation image generated at the resolution of 40×40, right image shows the same image generated at the resolution of 240×240.

The smoothness of high resolution characters also makes the style interpolation better as can be seen in Figure 6.6.

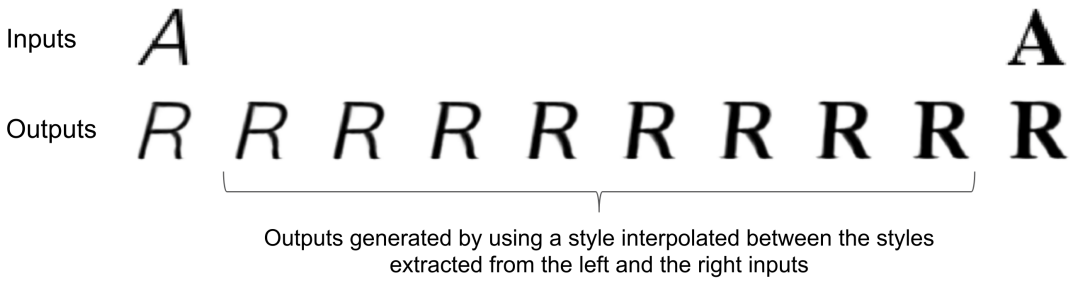

Figure 6.6: The interpolation of style between the styles extracted from left and right inputs. Outputs generated by the image function generator at 240×240 resolution.

6.3 Discussion

Working with image function representations shows promising results. The generated images are overall cleaner and have less noise due to the way they are produced. Since the image function can be rendered to a pixel representation at any resolution, the neural network approximating the image function essentially stores the vector representation.

All the results in this chapter where produced while training on the same low resolution data. While the hardware often limits the resolution of images produces by traditional CNNs generating pixel images, it could be possible to train networks to generate image function representations and measure the error on much larger images.

Nonetheless, the results from training on low resolution data already look decent. The only two issues that would need fixing are the slight "wobliness" of outlines that should straight and slight blur at the outline edges. It would be difficult to fix the wobbly lines with the current image function representation. Some regularisation would need to be enforced that would bias the output towards straight and smooth polynomial lines.

The slight blur at the outline edges might be a result of training on low resolution data where a lot of points are interpolated and do not contain saturated pure white or pure black values. However, since the outputs are well behaved, it might be possible to recover the sharp edges after converting to a pixel representation by some standard image processing techniques such as sharpening algorithms. A similar idea would be to scale the final *tanh* activation to make it steeper at test time to shrink the blur effect or change the activation to a step function to obtain completely saturated values. [32] also provides some interesting experiments applying adversarial training to networks approximating image functions and some results with varying image sharpness.

Chapter 7

Conclusions and future work

The goal of this project was to explore the possibilities of generating missing characters in fonts given that some characters are available. A system capable of learning from multiple available characters and generating multiple required characters was designed and implemented in Chapter 4. The system was evaluated by comparison to a previous approach of [1]. Improvements to the proposed system were explored in further chapters.

The problem of transferring style involves many different considerations. A major consideration is the difference of solving the problem for standard and decorative fonts. The results of this project did not showcase many examples of model performance on decorative fonts since they are less represented in the project dataset but more importantly because the model was designed considering standard fonts. After the experiments in this project, it is clear that a completely different approach would need to be taken to tackle style transfer for decorative fonts with any reasonable success.

Another important consideration in approaching the problem is the representation of the font character data. The pixel representation used in the majority of this project is sufficient to infer information such as extracting the style information from a set of characters. However, a lot of issues arise from the fact that neural networks are tasked with generating characters in pixel representation rather than in a more constrained generation space.

From a practical perspective, the experiments and results of this project form a good intuition about the difficulty of the problem and a starting point in applying neural networks. However, the problem explored in this project is likely more general than what might be needed in practice. The best approach to solving a specific problem such as extending fonts to language scripts with bigger character sets would likely be different to what was used in this project.

7.1 Future work

The work done in this project opens many possible directions for further exploration.

The base model was designed as a relatively simple composition of convolutional networks. While the switch to convolutional networks brought a lot of improvement when compared to [1], it might be worth exploring more advanced style modelling neural network architectures such as variational autoencoders or other modern neural network representation learning techniques.

A little explored direction was the accumulation of styles extracted from different images in the input. Further work is required to figure out a way of combining inferred styles in a way improves the overall style estimation in all cases. Adding more input to a model should not make the output worse.

Generative adversarial networks are a active research area with new improvements coming out frequently. The GAN set-up in this project was simple relative to the most recent advances in the field. The research in GAN stability could allow to apply adversarial training more aggressively or enforce some task specific constraints that might help to improve the consistency of the model.

The most promising research direction is the exploration into using different representations for neural network outputs. As mentioned previously, most issues with the approaches in this project are related to the fact that neural networks predictions are underconstrained. The work done in Chapter 6 was a step into the right direction and seem to be able to fix some of the issues with the outputs of the initial models. However, it is likely that constraining the prediction space even more would bring further improvements. For example, a neural network could possibly predict the outlines directly instead of modelling greyscale values since most fonts have no intermediate colours between pure white and pure black. Differentiable data structures that can be learned by neural networks is also an active research area. Tying the neural network predictions directly to a vector representation would be the ultimate goal.

The specific area of machine learning for fonts, however, would most be benefited by standardised problems and accessible datasets. Common resources would allow researchers to fairly compare approaches and iterate faster.

Bibliography

[1] Paul Upchurch, Noah Snavely, and Kavita Bala. From A to Z: supervised transfer of style and content using deep neural network generators. *CoRR*, abs/1603.02003, 2016.

[2] How to choose the right face for a beautiful body — Smashing Magazine. https://www.smashingmagazine.com/2012/05/how-to-choose-the-right-face-for-a-beautiful-body/. (Accessed on 01/23/2018).

[3] How to choose the right font for your design — Designlab. http://trydesignlab.com/blog/how-to-choose-the-right-font-for-your-design/. (Accessed on 01/23/2018).

[4] Font technology – globalization — Microsoft Docs. https://docs.microsoft.com/en-gb/globalization/input/font-technology#font-fallback. (Accessed on 01/26/2018).

[5] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2414–2423, 2016.

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[7] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5967–5976, 2017.

[8] Yaroslav Ganin and Victor S. Lempitsky. Unsupervised domain adaptation by backpropagation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1180–1189, 2015.

[9] Joshua B. Tenenbaum and William T. Freeman. Separating style and content with bilinear models. *Neural Computation*, 12(6):1247–1283, 2000.

[10] Neill D. F. Campbell and Jan Kautz. Learning a manifold of fonts. *ACM Trans. Graph.*, 33(4):91:1–91:11, 2014.

[11] Quoc Huy Phan, Hongbo Fu, and Antoni B. Chan. Flexyfont: Learning transferring rules for flexible typeface synthesis. *Comput. Graph. Forum*, 34(7):245–256, 2015.

[12] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

[13] Kieran G. Larkin. Structural similarity index ssimplified: Is there really a simpler concept at the heart of image quality measurement? *CoRR*, abs/1503.06680, 2015.

[14] Samaneh Azadi, Matthew Fisher, Vladimir G. Kim, Zhaowen Wang, Eli Shechtman, and Trevor Darrell. Multi-content GAN for few-shot font style transfer. *CoRR*, abs/1712.00516, 2017.

[15] Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, and Zhen Wang. Multi-class generative adversarial networks with the L2 loss function. *CoRR*, abs/1611.04076, 2016.

[16] Shumeet Baluja. Learning typographic style: from discrimination to synthesis. *Mach. Vis. Appl.*, 28(5-6):551–568, 2017.

[17] zi2zi: Master chinese calligraphy with conditional adversarial networks. `https://kaonashi-tyc.github.io/2017/04/06/zi2zi.html`. (Accessed on 01/28/2018).

[18] Learning to write Devanagari-Hindi M-x learn. `https://mridul.github.io/blog/learning-to-write-devanagari-hindi/`. (Accessed on 01/28/2018).

[19] Google fonts. `https://fonts.google.com/`. (Accessed on 01/28/2018).

[20] The FreeType project. `https://www.freetype.org/`. (Accessed on 01/28/2018).

[21] Who is the best at X? `http://rodrigob.github.io/are_we_there_yet/build/`. (Accessed on 03/11/2018).

[22] Po-hsun-su/pytorch-ssim: pytorch structural similarity (SSIM) loss. `https://github.com/Po-Hsun-Su/pytorch-ssim`. (Accessed on 04/01/2018).

[23] soumith/ganhacks: starter from "How to Train a GAN?" at NIPS2016. `https://github.com/soumith/ganhacks`. (Accessed on 03/31/2018).

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[26] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[29] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[30] Diederik P. Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3581–3589, 2014.

[31] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[32] Generating large images from latent vectors — . `http://blog.otoro.net/2016/04/01/generating-large-images-from-latent-vectors/`. (Accessed on 04/11/2018).