Preference Based Sketch Generation Using the Sketch-RNN Framework

Rory Nicholas

April 2023

BSc Computer Science[1]

Supervised by Dr. Youngjun Cho

[1]This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This project aims to work closely with a user to provide them with artistic inspiration and assistance to help create simple sketches. More specifically, it builds upon existing research into the field of sequence-based sketch generation using machine learning models to reinterpret and auto-complete a user-created sketch in ways which cater to their expressions of preference. This project particularly leverages the Sketch-RNN framework (Ha, David and Eck, Douglas, 2017)[1] to do so.

After conducting research into this area, requirements and use cases were developed. Following an analysis of these, a robust design which is resilient to future extensions of the framework was developed and implemented. Additionally, an investigation into the viability of transfer learning to facilitate future improvements on the tool was carried out and evaluated.

An interactive application allowing users to easily sketch and generate sketches was delivered, alongside insight into further enhancements to the framework and application which could be carried out in the future.

Contents

1 Introduction **5**
1.1 Problem Outline . . . 5
1.2 Project Goals . . . 5

2 Context **6**
2.1 Background Information . . . 6
2.1.1 Generative Adversarial Networks . . . 6
2.1.2 Variational Autoencoders . . . 7
2.1.3 Generative Approaches Summary . . . 9
2.1.4 Recurrent Neural Networks . . . 10
2.1.5 Long Short Term Memory . . . 10
2.1.6 Bidirectional Recurrent Neural Networks . . . 10
2.1.7 Bivariate Gaussian Mixture Models . . . 11
2.1.8 Transfer Learning . . . 11
2.2 Sketch-RNN . . . 12
2.2.1 Framework . . . 12
2.2.2 Related Work . . . 13

3 Requirements and Analysis **14**
3.1 Problem Statement . . . 14
3.2 Deliverables . . . 14
3.3 MoSCoW Requirements . . . 15
3.3.1 Must Have - Critical Requirements . . . 15
3.3.2 Should Have - Higher Priority Requirements . . . 15
3.3.3 Could Have - Lower Priority Requirements . . . 15
3.3.4 Won't Have - Out-of-Scope Requirements . . . 16
3.4 Use Cases . . . 16
3.4.1 Sketch Application Use Case Diagram . . . 16
3.4.2 Transfer Learning . . . 17
3.5 Analysis . . . 17
3.5.1 Sketch Application . . . 17
3.5.2 Transfer Learning . . . 18

4 Design **19**
4.1 Programming Language . . . 19
4.2 Graphical User Interface . . . 19
4.2.1 PyGame . . . 19
4.2.2 Tkinter . . . 20
4.2.3 PyQt5 . . . 20
4.2.4 GUI Summary . . . 21
4.3 Sketch Application Structure . . . 21
4.3.1 General Process Overview . . . 21
4.3.2 Model-View-Controller Architecture . . . 22
4.3.3 UML Class Diagram . . . 23

4.3.4 Latent Vector Variation 24
4.4 User Interface Design 24
4.4.1 Main Window 24
4.4.2 Generated Display Window 25
4.5 Transfer Learning Investigation 26
4.5.1 Adapting the Pre-Existing Training File 26
4.5.2 Data Tweaking Module 27
4.6 Training Plan 28

5 Implementation **30**
5.1 Sketch Application 30
5.1.1 User Interface 30
5.1.2 Data Processing 33
5.1.3 Model Interaction 33
5.1.4 Overall Structure 35
5.2 Transfer Learning 35
5.2.1 Data Tweaking 35
5.2.2 Training 36

6 Evaluation **36**
6.1 Testing Strategy 36
6.2 Sketching Application 37
6.2.1 Overall Appearance 37
6.2.2 End-to-End Testing 38
6.2.3 Sketch Generation Assessment 39
6.3 Transfer Learning 40

7 Conclusions **44**
7.1 Achievements 44
7.2 Future Work 44
7.3 Challenges 44

8 Appendix **46**
8.1 Code Repository 46
8.2 Diagrams 47
8.3 Code Listing 53
8.3.1 SketchTool.py 53
8.3.2 SketchToolInterfaces.py 66
8.3.3 SketchToolUtilities.py 67
8.3.4 model.py 72
8.3.5 TransferLearning.py 83

1 Introduction

1.1 Problem Outline

From a young age, many of us are taught to observe and re-create the world around us using a pencil and paper. In doing so, we learn to communicate thoughts, feelings, and concepts by mindlessly doodling our environment; and we all develop a unique style with which to represent these ideas. However, at times one may become stuck for inspiration or frustrated at an inability to express themselves with the deftness they had in mind. At other times, one may simply wonder about the many different interpretations which different people may hold of a single piece or art. Additionally, certain physical or mental disabilities may at times leave someone unable to express themselves artistically to the extent that is enjoyed by the rest of the world.

Thankfully, recent advances in the domain of generative artificial intelligence have allow those lacking in confidence to generate striking artworks which appear to show a level of creativity previously thought to be inherently human. Impressive models such as DALL-E 2 [2] allow users to generate high quality images based on a wide range of text prompts, allow users to let their mind wander with the assistance of technology. However, the majority of mainstream attention into image generation using artificial intelligence has been focused on pixel-based methods. While proven to deliver exciting results, this strays far from the intuitive human understanding of artwork. Instead, humans tend to understand artworks as a sequence of strokes, colours and textures.

Therefore, a tool which generates pieces of art for the purpose of working closely with users to learn from sketches and provide many different interpretations of the same drawing may well benefit from working with images in a sequential, vector-based format. Indeed, the potential for success in such models has been shown in papers such as (Alex Graves, 2014) [3]. By interpreting a drawing as a series of strokes, a model may have a higher potential to work closely with users by generating or adapting existing sketches in a more familiar and understandable way.

1.2 Project Goals

The goal of this project is to build upon existing research into sequential vector-based image generation in order to provide a tool which will work closely with users to augment existing sketches in ways catering to their specific preferences. This project will focus on close interaction with a wide range users.

2 Context

This chapter outlines current machine learning algorithms for image generation, and compares the advantages and disadvantages of each in order to justify the particular focus on the Sketch-RNN model to achieve the aim of a tool to augment sketches. It then describes key techniques and mathematical concepts which may be used to improve the performance and ability to train such models, along with ways in which to maximise their potential post-training.

2.1 Background Information

Within the field of image generation, two prominent approaches are Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs). Both have potential to deliver high quality outputs, however each has advantages and disadvantages which mean they cater to different situations. Additionally, standard Dense Neural Networks generally use vectors of a static size as input and output - in order train on and generate sketches as a series of strokes. To adapt to this Recurrent Neural Networks (RNNs) may be used for these purposes. RNNs may utilise a variety of cells - a notable example being Long Short Term Memory (LSTM) cells.

In order to learn the basics of machine learning, the book 'Neural Networks from Scratch in Python'[4] was used. This provides the background knowledge to design, build, and train standard deep neural networks.

2.1.1 Generative Adversarial Networks

As written in (Goodfellow 2014)[5] in 2014, the aim of a Generative Adversarial Network (GAN) is to produce a model which, given some dataset, is able to take in some random noisy vector as input and produce an output such that it is statistically similar to said dataset. The training process is such that we have a 'generator' G and a 'discriminator' D; the goal of G is to map random noisy vectors to an output which is statistically similar to the dataset, whereas the goal of D is to decide given whether some input data belongs to the original dataset.

More formally, where θ_g are the parameters of G, p_{data} describes a distribution over the dataset, and z represents some random noisy vector, the aim is to find the optimal θ_g such that $G(z; \theta_g) \approx p_{data}$.To achieve this, we train D with the aim that where θ_d are the parameters of D and x is some input data:

$$D(x; \theta_d) = \begin{cases} 1 & \text{if } x \in p_{data} \\ 0 & \text{otherwise} \end{cases}$$

For a given training step, we use a batch of random noisy vectors through G to generate 'fake' images. Along with a sample of 'real' images from the dataset, we may feed these to D so as to obtain probabilities of the likelihood that each image belongs to the original dataset. From here we can calculate some loss function; D is penalised for the degree of wrongness in the decisions, and G is penalised for the likelihoods of

the generated samples being fake. This approach is demonstrated in Figure 1.

In this way, D and G learn throughout training as adversaries; as D improves its ability to discriminate between 'real' and 'fake' data, G learns to improve the quality of its outputs to try and fool D. The training is likely to be complete when D predicts correctly exactly half of the time; in other words G produces outputs which so realistic that D has no better tactic than to 'flip a coin' as to whether some input data is real. At this point we can throw away D and consider G as properly trained, taking samples by feeding it some noisy vector and receiving our believable output.

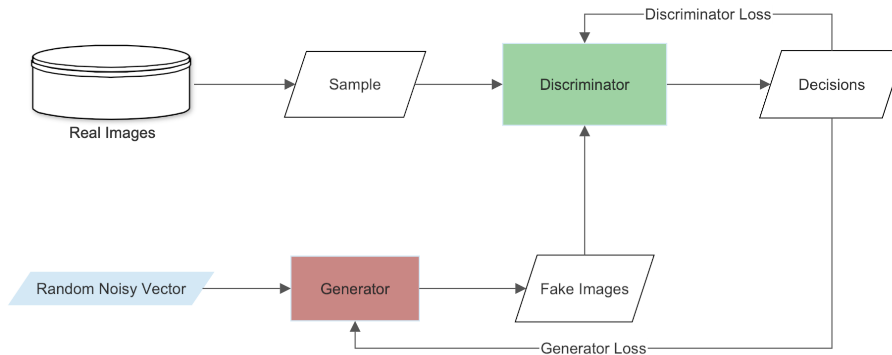


Figure 1: An diagram showing the GAN training process

While this approach has a strong potential to generate high quality images, in should be noted that this training method provides absolutely no necessity for the model to create any structure within its generation. That is to say, there is no guarantee that this model can be fed specific inputs in order to generate an output with certain features. For example we may train a GAN to generate high quality images of faces, however if a user specifically wanted to generate a male face with ginger hair it may be difficult to find an input vector which will prompt the model to do so.

2.1.2 Variational Autoencoders

Autoencoders

An autoencoder[6] is a network comprised of two sub-networks; namely the 'encoder' and 'decoder'. The encoder is designed so as to map input data into a 'latent vector' - a vector describing a position in some latent space. Conversely, the decoder is designed to map a latent vector into an output image.

The idea of this approach is that through training, the encoder is fed several images from the dataset and outputs their corresponding latent vectors. The decoder then takes these latent vectors and outputs images with the aim of reconstructing the original image as closely as possible. This forces the encoder and decoder to 'agree' (for lack of

a better word) on what exactly the latent vector represents. The hope is that by penalising the networks based on the accuracy of reconstruction, the encoder is forced to map inputs into the latent space such that it describes some features of the input data - allowing the decoder to have some direction as to how to produce a satisfactory image. After training, by feeding the decoder various samples from the latent space generated images can be retrieved, even using vectors not specifically seen during training.

For example, Figure 2 shows a possible latent space representation on a model trained to generate pictures of living things with a latent vector size of just 2. If, say, the encoder recognises an image as a bird, it should provide the decoder with a latent vector describing that corresponding position. Provided the decoder has learned to interpret the encoders decided structure of the latent space, it should proceed to generate an image of a bird. Furthermore, within these constituent areas there may be a higher level of detail (depending on the complexity of the model). For example, the 'human' subspace may be further divided into different hair colours, gender, etc.

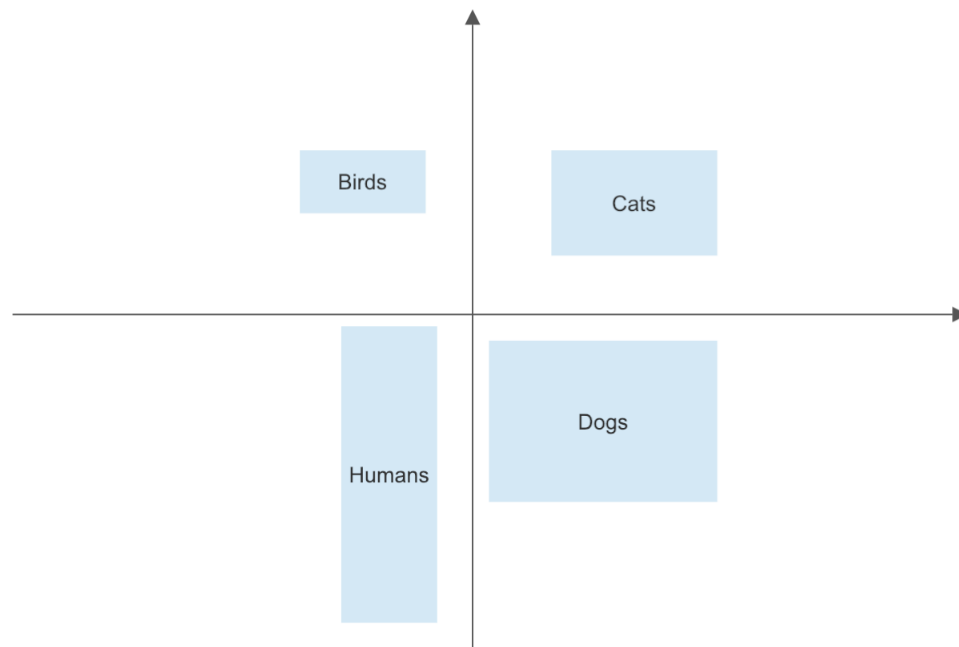


Figure 2: Unregularised Latent Space

Variational Autoencoders[7]
There remains one key issue with a plain autoencoder. Where the aim of the model is to generate new, unseen data by sampling from the latent space, this cannot reliably be done with the process described so far. Even if a latent space is created through training which provides some reasonable separation between different classes of data, no

necessity has been imposed for this space to be well-formed. For example, again using Figure 2 we can see a situation in which there exist areas which correspond to nothing; sampling the area between cats and birds may produce nothing but meaningless noise. Additionally, at times it is desirable to interpolate between different points in the latent space to generate data belonging to a mix of two classes at once. To do so, it would be necessary for there to be some amount of overlap between significant parts of the space.

To this end, during training the loss function is adapted such that the model is penalised based on the Kullback-Leibler (KL) divergence[8] between encodings made onto the latent space and a standard Gaussian distribution - . This creates an effect analagous to ‘squeezing’ the latent space into a smaller area; forcing overlaps between meaningful areas and pressuring the encoder to make full use of the area it occupies - reducing redundant gaps. This is known as ‘regularising’ the latent space.

This means the loss function is now made up of two summed terms: the reconstruction and KL losses. Finally, the model can may be further adjusted by applying a weight to the KL term such that we may decide to give control the model’s focus on generating accurate reconstructions with the need to regularise the latent space.

Additionally, one can introduce noise to the latent vector produced by the encoder doing training. In other words, for a given training sample, given output z from the encoder, the vector $z' = z \odot \mathcal{N}(0, 1)$ is fed to the decoder. This can prevent overfitting - as the decoder must learn to overcome this disruption by learning the correct interpretation of the latent space instead of memorising exact outputs for particular latent vectors.

Overall, the advantages of Variational Autoencoders (VAEs) include that they allow for data generation over which users can maintain a higher level of control, sampling from specific areas of the latent space to create the desired features in the output. However, it should be noted that a latent space will rarely be contained to two dimensions - this would not provide the model with much room for expression when training on more complicated datasets (in fact the dimensionality of the latent space is regularly in the hundreds). Furthermore, VAEs often tend to produce blurry outputs[9]. This may be explained as the model finding a local minimum of loss in which it approximates a valid reconstruction over a range of possible outputs as opposed to a specific one to maintain a decent — but not spectacular — average performance.

2.1.3 Generative Approaches Summary

To recap this subsection:

- GANs are capable of producing very high-quality images.
- GANs are **not** designed to be sampled for specific features within a dataset.
- VAEs occasionally produce lower-quality, blurry images.
- VAEs **can** be designed be sampled for specific features within a dataset.

After considering the advantages and disadvantages of each, it seems that VAEs will be better suited to the project, as we would like to be able to sample outputs with specific features.

2.1.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs)[10] are an adaptation to allow models to handle sequences as input and output. The idea is that the network unrolls into a series of cells such that for each element of the input sequence, there is a matching cell which takes the element as input, optionally giving an output and passing some information about the data to the next cell. Therefore as a sequence of data is passed through the network, each cell receives information of the context from the previous cell, updates its own state, processes a piece of the data, and passes on the information it gained itself. The contextual interpretation of the data sequence therefore cascades down the network.

2.1.5 Long Short Term Memory

Within RNNs, there are many options as to which type of cell to use; some are simpler and hold a small amount of information of the context, and others use more complex methods to initialise a detailed cell state. One type of cell which has gained prominence is the Long Short Term Memory Cell (LSTM)[11]. These cells use more intricate operations to forget, update, and carry information through a cell state which is passed on the the following cell, which allows certain important pieces of information to be held for a long time and less important pieces of information to be forgotten or updated (hence 'long short term memory').

It's worth noting there are a number of variants on this vanilla LSTM which may add or remove components of the cell and/or compute their states and output differently. However, improvements upon the the vanilla cell are limited[12].

2.1.6 Bidirectional Recurrent Neural Networks

When using an RNN, at a given step of an input sequence the context of future steps which have not yet been seen can be important to aid the current cell in its interpretation.

For example, take the sentence "She looked at the bat before it flew out of the cave". If this sentence were fed sequentially into an RNN, by the time the network sees the word 'bat' it has only seen the preceding section of the sentence "She looked at ". In this case, the network may not have sufficient contextual information and therefore interpret 'bat' to refer to, say, a cricket bat; an output may therefore be produced using this mistaken interpretation.

To remedy this issue, one may use a Bidirectional RNN[13], in which rather than one RNN passing over the data in order, two RNNs simultaneously pass over the input sequence; one passes over the sequence in the original order, and another passes over it

in reverse order. From there, outputs may be produced by some combination of the two RNNs outputs and states. In this way, the context learned from later parts of a sequence may be used to inform the context as a whole.

2.1.7 Bivariate Gaussian Mixture Models

A mixture model[14] serves as a method to describe populations in a more statistically specific way by altering distribution parameters with which predictions are made based on the subclass with which individual samples belong. For example, suppose cats sleep a higher number of hours on average than dogs per day. If one tried to model the average time these pets sleep per day using a single Gaussian distribution, the model would fail to make precise predictions as the two distinct peaks of each pet distribution cannot be matched by the singular peak of the Gaussian. Instead, a mixture model may be used, in which multiple distributions may be applied for each subclass.

In a mixture model (in this case Gaussian) we can describe the probability of an event occuring as:
$p(x) = \sum_{i=0}^{K} \Pi_i \mathcal{N}(x|\mu_i, \sigma_i)$,
$\sum_{i=0}^{K} \Pi_i = 1$
where x is an event, K denotes the number of subclasses, Π_i, μ_i, σ_i denote the proportion of the population which is made up of the ith subclass, along with the mean and variance of this subclass respectively. In this way, probability estimates may be made with a higher level of precision.

In bivariate Gaussian distributions, one can model several related (or not) attributes of a population with a single distribution[15]. For example, one may model the number of hours a dog sleeps along with the number of hours of physical activity it does in a day. To do so, one may model $x, y \sim \mathcal{N}(\mu_{sleep}, \mu_{activity}, \sigma_{sleep}, \sigma_{activity}, \rho)$, where μ and σ denote the mean and standard deviation of each attribute respectively, while ρ represents the correlation between them.

The two above ideas may be combined to result in a bivariate Gaussian Mixture Model - simply modelling multiple attributes for several distinct subclasses within a population. To describe this model, one may configure an array of 6-tuples $(\Pi_i, \mu_{i,x}, \mu_{i,y}, \sigma_{i,x}, \sigma_{i,y}, \rho_i)$ for $i \in 1, 2, ..., K$ where $\sum_{i=0}^{K} \Pi_i = 1$. Notice that this array would have a size of $6K$.

2.1.8 Transfer Learning

Transfer learning is a machine learning method through which one may increase the speed of training by utilising training which has already been carried out on a similar model.[16]

When training neural networks, there is an emphasis on ensuring that the model generalises well to the set of data it is trained on. In other words, a model should be effective when tested on data from outside the training dataset, rather than overfitting

to specifically perform well on the exact examples given to it during training.

Transfer learning aims to utilise this property. Assuming a trained model generalises well to a particular problem, it may be the case that this model can serve as a good starting point for new models which aim to carry out a similar function. Training may be sped up by using the parameters learned by the older model, with the idea that re-using its parameters (weights, biases, etc.) will enable the new model to adapt to its own aim more quickly than initialising parameters at random. This is analagous to the idea that someone who is experienced in playing badminton may initially be better at playing tennis than others who are not.

Transfer learning is generally carried out by creating a new model by reusing parameters from a previously trained model, and then 'tweaking' the model by training it on a new dataset. As this means the model may achieve results more quickly than otherwise, transfer learning is in large part useful when there is a scarcity of data for a specific purpose, but an abundance of similar data.

2.2 Sketch-RNN

2.2.1 Framework

Sketch-RNN[1] is a paper in which a framework was proposed with which one may train a model to produce or finish sketches based on user input. It is trained using the Quick, Draw![17].

As described in the paper, the framework consists of a sequence-to-sequence VAE which uses 'stroke-5' format for input and output. This stroke format represents a drawing as a series of vectors, where each vector is a 5-tuple where a stroke $S = (x_\Delta, y_\Delta, p_0, p_1, p_2)$ where x_Δ, y_Δ represent the the direction in which the current stroke moves the pen, and p_0, p_1, p_2 act as a one-hot vector to signify pen state:
$p_0 \rightarrow$ 'the pen is currently touching the paper'
$p_1 \rightarrow$ 'the pen is lifted from the paper after this stroke'
$p_2 \rightarrow$ the drawing has finished.
From this it can be inferred that once $p_2 = 1$ is met, the remainder of the sequence can be disregarded. Additionally, once a stroke denoting this has been produced by the decoder, any subsequent strokes generated are meaningless.

The encoder is a BRNN where $h_\rightarrow, h_\leftarrow$ are the outputs of the forward and backward passing components of the BRNN respectively, the latent vector $z = \mu + \sigma \odot \mathcal{N}(0, 1)$ where $\sigma = exp(\frac{\hat{\sigma}}{2})$ and $\mu, \hat{\sigma}$ are retrieved by passing $(h_\rightarrow; h_\leftarrow)$ through a dense layer. Notice the Gaussian noise added when producing z, as described in section 2.1.2

The decoder is an RNN where the initial states are the result of passing z through a single dense layer with a tan activation. More formally, $(h_0, c_0) = \tan(W_z, b_z)$ where h_0, c_0 are the starting cell states (c_0 is optional, depending on the type of cell used). With an initial stroke of $(0, 0, 1, 0, 0)$, a sequence is generated through the RNN until

some pre-defined maximum sketch length N_{max} is reached, or a stroke with $p_2 = 1$ has been produced. In order to generate the sequence, a given cell will take a concatenation of the previous stroke with the latent vector as input to create its states before passing them through a dense layer to provide an output. More formally, at time step i, input $x_i = S_{i-1}; z$ where S_{i-1} denotes the previously generated stroke. Cell states $h_i, c_i =$ forward(x_i, h_{i-1}, c_{i-1}), and the output $y_i = W_y h_i + b_y$, with $y_i \in \mathbb{R}^{6M+3}$ where M is some pre-defined constant to denote the number of distributions to produce in the mixture model. The size of y_i is due to using $6M$ values to represent the bivariate gaussian mixture models (see section 2.1.7) and 3 additional values to describe the probability distribution of the three pen states. Note that $softmax$ would be applied to the set $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_M\}$ of mixture model probabilities and to the set of pen states to ensure the probabilities within each sum to 1.

The framework also discusses an introducing an element of randomness by using a ‘temperature’ hyperparameter to the model, as shown in equation 8 of the paper.

Using this framework, one can train or load pre-trained models to encode sketches in ‘stroke-5’ format to encode a sketch onto a latent space, or decode a latent vector so as to sample a sketch. This can be used to either sample images from scratch or finish incomplete sketches by feeding pre-existing strokes into the decoder before sampling strokes to complete them. Alongside the research paper, repositories containing code to interact with and use the framework are provided in JavaScript and Python.

2.2.2 Related Work

There is a vast amount of projects related to sketch generation using crude user-supplied sketches, in a somewhat similar manner to Sketch-RNN. These include:

- SketchyGAN[18]: Conversion of simple sketches into photo-realistic images.
- Sketch your own GAN[19]: Create a GAN to generate images using a single user-input sketch
- Sketch-Based Image Retrieval[20]: Retrieving an item from a database based on a user sketch

However, there is a lack of projects which use latent space interpolation within a sequence-to-sequence framework in order to generate sketches increasingly to a users preference.

3 Requirements and Analysis

This chapter outlines the specific aims of the project based on the particular problem being solved. Subsequently, the requirements learned will be analysed so as to discover a high-level overview of how the project may be structured.

3.1 Problem Statement

There are currently few projects marrying the idea of sequential, vector based art generation with utilising the latent space developed by a VAE to work closely with users to create sketches based on their preferences. However, there may be a high potential for such projects to benefit from a model trained to interpret sketches in a way close to that which is inherent almost unanimously within humanity. Such a project may provide inspiration or assistance to aspiring artists, or facilitate the rapid development of a range of designs based on users' tastes.

To that end, this project aims to attend to these ideas by creating a tool which builds upon existing research to explore these possibilities. Moreover, many existing works assume some level of existing skill or ability in the user. Certain demographics would particularly benefit from tools with which one can work closely with few-to-no barriers to entry. This includes children and those with a physical or mental disability — it may provide great benefit for a tool to aid the artistic ability of those for whom such a process is not normally accessible.

A significant existing work into this area is the Sketch-RNN framework[1], discussed in section 2.2.1. By building upon this work to incorporate users' preferences, a tool could afford the creation of high quality sketches to a wide range of users without restricting their expression.

However, one limitation with this framework is that its stroke format does not encode a stroke's colour or size, which restricts the possibility for a tool to expand users' artistic expression.

3.2 Deliverables

Based on the problem statement, two deliverables have been identified:

- A tool allowing users have their own sketches reinterpreted and completed in a way that caters to their preferences by exploring the latent space of a trained VAE.
- An experiment into the effectiveness of transfer learning within the Sketch-RNN[1] framework.

The first of these deliverables has been identified to directly address the problem statement. The latter has been identified to investigate the flexibility of the framework. Firstly, a transfer learning investigation may suggest whether an expansion of the stroke

format currently used by the Sketch-RNN framework could be aided by transfer learning, due to the scarcity of such a data format. Secondly, this explores alternative ideas as to how sketch generation may be catered to a users' preferences, for example by adapting an existing model to generate sketches with slight tweaks with respect to user preferences learned over time.

For the most part, these two deliverables will act as independent parts of the project, with the idea that the concepts may be used in conjunction in some future work.

3.3 MoSCoW Requirements

The MoSCoW method has been used to identify and prioritise the requirements of the project.

3.3.1 Must Have - Critical Requirements

1. An application to enable the user to create sketches.
2. The sketch application is clearly visible.
3. The sketch application allows users to submit a drawing for re-interpretation.
4. The sketch application allows users to submit an incomplete sketch for completion.
5. Upon sketch submission, the application displays results clearly.
6. The user may control the generated sketches through an expression of preference.
7. The user may save a generated sketch.
8. Transfer learning is conducted on models within the Sketch-RNN framework[1].

3.3.2 Should Have - Higher Priority Requirements

1. The application allows users to undo their previous stroke.
2. The application allows users to browse several generated sketches.
3. The application is responsive and easy to use for a wide range of users.
4. The application allows fluid movement between its different functions.
5. The effectiveness of transfer learning is evaluated based on the degree of difference between datasets.

3.3.3 Could Have - Lower Priority Requirements

1. It is easy to plug-in a variety of models to the sketch application.
2. The sketch application is resilient to future extensions of stroke format.

3.3.4 Won't Have - Out-of-Scope Requirements

1. The sketch application will not be designed for detailed or complicated sketches.
2. The Sketch-RNN framework will not be adapted in any significant way.
3. The application and transfer learning investigation will not specifically work in conjunction.

3.4 Use Cases

This section models a use case diagram for the sketching application. In this diagram, 'sketch completion' refers to an incomplete sketch being finished by the model, whereas 'sketch reinterpretation' refers to a full sketch being taken in and reinterpreted by the model.

Use cases are also discussed for the transfer learning component of the project, although due to its nature a diagram would not be necessary.

3.4.1 Sketch Application Use Case Diagram

Figure 3 maps the use cases for some potential users of the sketch application.

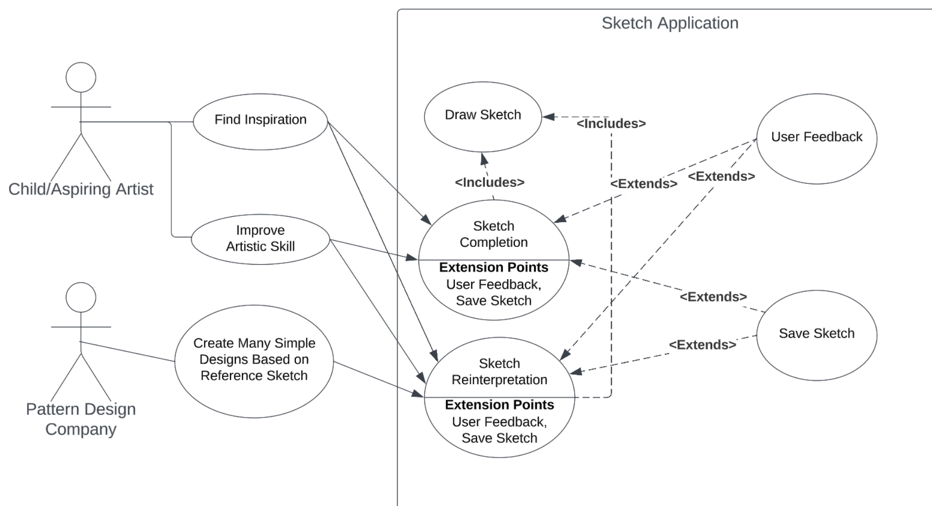


Figure 3: Sketch Application Use Case

One potential actor would be those who are struggling to improve their sketching skills, such as younger users, those with physical or mental disabilities, or simply aspiring artists. These actors may hope to use the tools to see different ways in which a

particular sketch may be interpreted or completed. In the specific case of a disability, the tool may be used to aid the user to create a sketch in *exactly* the way they were intending. For example, a user with a motor disorder may not be able to draw a smooth sketch exactly as they'd like, however they may have a sketch reinterpreted by the application such that their sketch is re-drawn with the particular strokes they had in mind.

Another potential actor may be a pattern design company, where simple repeatable designs may be desired. In this case they may use the application to re-interpret a design such that they may rapidly prototype a variety of designs to offer to a customer.

3.4.2 Transfer Learning

The investigation into the effectiveness of transfer learning would benefit anyone who is aiming to expand the use of the framework. For example, it may justify the ability to use such a technique to expand the existing stroke format in a way which requires minimal data. Additionally, it could prove viable a future improvement to the sketch application which may perform adaptations over time to an existing model based on long-term user preferences.

3.5 Analysis

3.5.1 Sketch Application

To model the general way in which the sketch application may function, I created the flowchart in Figure 4 according to the relevant MoSCoW requirements and Use Cases.

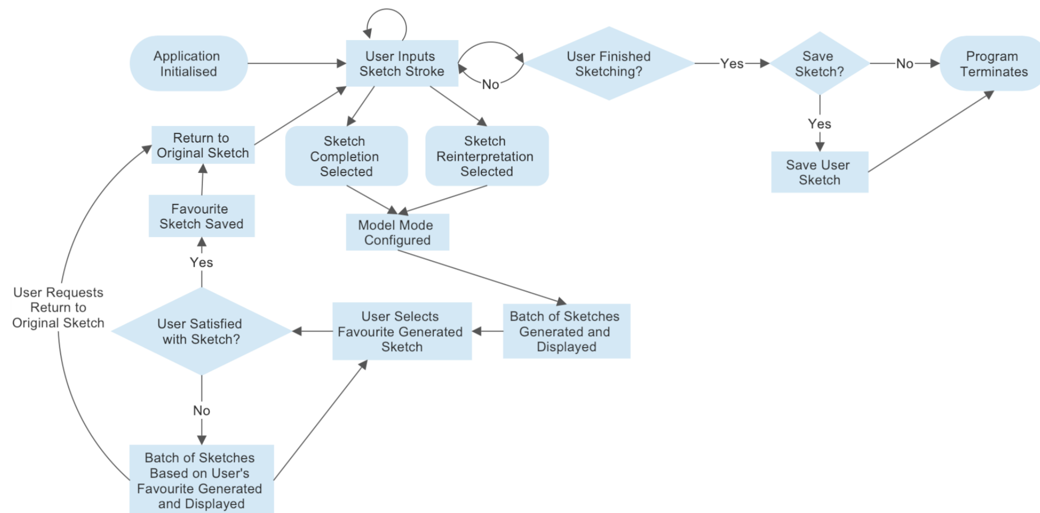


Figure 4: Sketch Application Flow Chart

In this approach, the format through which the user may express preference for a particular sketch by picking their favourite from a batch generated by the model. Then, more sketches are generated with the favourite sketch as the source, such that the user may ‘walk’ through the latent space of the model in order to find a design which suits them. In this way, they may also continue to select their favourite sketches for as long as desired until they find one that satisfies. At any stage of this process the user should be free to return to their own sketch or save their favourite.

Another note is the ability for the user to save their own sketches. While this application is *not* in any way expected to be used as a primary art application for detailed sketch creation, the idea is that after browsing the sketches generated by the model, the user may continue their own sketch in order to incorporate aspects from the generated sketches into their own drawing and then save it for future reference.

3.5.2 Transfer Learning

Transfer learning will be assessed by using a pre-trained model and dataset provided by the existing Sketch-RNN repository. From there, some ‘tweaks’ will be made to the dataset, with the intention that a new dataset is created which is distinct yet similar to the dataset on which the model was trained. From there, training and evaluation will be carried out on this model using the new dataset. The validation costs throughout training along with sketches sampled from the new model will tell whether the technique is successful.

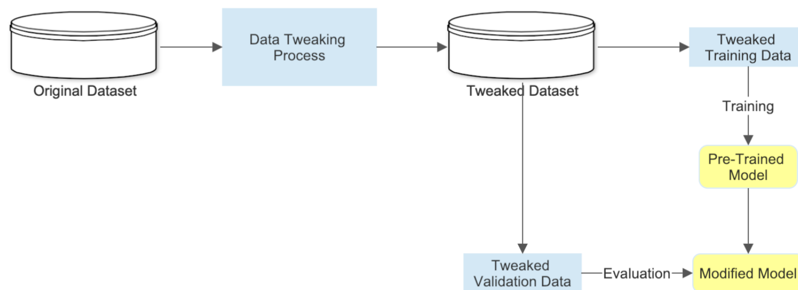


Figure 5: Transfer Learning Flow

Additionally, the data tweaking process should have some degree of variability such that there is some control over the difference to the original. This is to evaluate the effectiveness of transfer learning with respect to datasets with different degrees of similarity.

4 Design

This chapter describes the design process which follows from the requirements and their analysis set out in the previous chapter.

4.1 Programming Language

As discussed in section 2.2.1, repositories implementing Sketch-RNN are provided in both Python and JavaScript. Upon reviewing these repositories, it can be seen that the Python repository offers more extensive and detailed code than the JavaScript repository. Most importantly, the Python repository includes a file which can be used to train a new model, which will be key in the implementation of transfer learning.

Due to the above, the project will be written in Python to speed up development by more easily training models during the transfer learning investigation.

One thing to note however is that the Python implementation of Sketch-RNN is written in TensorFlow Version 1, which drastically differs from the newer version 2. This means many of the functions and structures used in the repository are declared as deprecated and the code may not run as efficiently as possible.

4.2 Graphical User Interface

The sketch application will require a Graphical User Interface (GUI) in order to provide the user with an easy way in which to sketch and view generated sketches. As discussed in the previous section, Python will be the language of choice and so the choice of GUI must reflect that.

The requirements of the GUI for this project include:

- Is a Python library
- Capability to process mouse events representing strokes
- A way in which to clearly display sketches in 'stroke-3' format
- Widgets such as buttons to allow users to interact easily with functions of the application
- The ability to for a user to move fluidly between different application state (where a 'state' means the application is being used for a particular function)

The following subsections will discuss some GUIs available as options for the project.

4.2.1 PyGame

PyGame is a library which is best suited for creating dynamic and responsive displays. It offers a simple API to draw sprites and objects onto the screen, and gives the developer a high degree of control over the application mainloop meaning the interface

should never become ‘stuck’ as due to backend processes.

One downside is that PyGame offers very little in the way of functional widgets. For instance, there is no built in way to create a button - a developer must instead create a ‘pseudo-button’ by creating a rectangle overlaid with some text, and binding a mouse click event to the area covered by this button.

Overall, PyGame would be well suited to the application in terms of creating an interface which the user can sketch on, while making sure the interface remains responsive whilst sketches are being generate by the model. However, the lack of built-in widgets would make it much more difficult to create robust and easy-to-use functionality to the application.

4.2.2 Tkinter

Tkinter has been described as the “de-facto Python GUI”[21]. It is a well-established library which provides cross-platform support - elements of the interface even change appearance to match the operating system from which the application is being run.

Tkinter provides functionality in the form of widgets. Built-in buttons, value scales, and entry boxes mean it provides a good solution for applications which require a high degree of functionality. It also provides a ‘Canvas’ object which allows drawing complex shapes and supports binding a range of events which would be well suited to a drawing application.

One downside is that due to its age, Tkinter provides interfaces which tend to look rather outdated - and does not provide much opportunity to fix this.

Additionally, Tkinter works on a rather strict mainloop. This means that the interface will only update at the end of each mainloop — and events will block this loop. In other words, if an event triggered by a button press takes a long time to process, the interface may become unresponsive and display ‘application not responding’ until the process finishes, which can easily confuse a user.

Overall, Tkinter would be an excellent choice in terms of providing functionality easily to the application and would be well suited to creating an interface on which the user can sketch. Tkinter is also well established and it is often included when a user first downloads Python, meaning it does not add additional dependencies. However, it may become unresponsive whilst sketches are being generated by the model. Additionally, the user interface will not look as professional and polished as alternative libraries would allow.

4.2.3 PyQt5

PyQt5 is a library which provides cross-platform support for a range of applications, a wide variety of widgets, and visual design tools to aid developers to quickly create

an interface. Additionally, many components of this library are written in C/C++ and there is a fair degree of developer freedom over the mainloop, meaning the application would remain responsive, smooth, and fast.

The downside is that PyQt5 requires the purchase of a commercial licence; it does not come pre-installed with any Python distribution. It also only provides support for Python version 3, which could make the project less accessible for some.

Overall whilst PyQt5 is an strong tool for any application, the purchase of a licence adds a risk factor to the project which cannot be justified — the project simply does not require such a complicated GUI such that a purchasable tool is likely to be necessary. Additionally, the increase dependencies of the tool would unnecessarily impede potential users from considering using the application.

4.2.4 GUI Summary

Overall, having considered some potential options for GUI libraries, I decided that the project would be best suited to Tkinter. This is because it is not expected that backend processes will have a long duration, meaning mainloop blocking will not be a significant issue. Additionally, the functionality provisions of the library will be invaluable. Whilst it would be preferred for the GUI to have a clean and modern appearance, the functionality of the application should be of the highest priority in this instance.

4.3 Sketch Application Structure

4.3.1 General Process Overview

In order to utilise the strengths of Tkinter, the approach this project takes is to constantly have a ‘main’ window on which the user can sketch. Upon request to generate sketches, a new window is opened which displays the new sketches based on the strokes made by the user in a grid, on which the user can select their favourite. This new window acts independently from the main window following its initialisation. When a favourite is chosen, the window can request a new set of sketches from the model based on this sketch. More specifically, the model uses the latent space on which the favourite was generated. An array of latent vectors is generated, where each vector is the vector corresponding to the favourite sketch with some Gaussian noise added to it. This array of vectors can then be used to generate new sketches, at which point the process repeats with the user again picking their favourite. In this way, the user can ‘walk’ around the latent space - as they choose their favourite sketch the new sketches are generated from the latent space close to this favourite and so resemble it more closely with small deviations in their characteristics.

This process may be best described with an example. A user may draw a picture of an owl with no wings and request new similar sketches to be generated. Their sketch is encoded into a latent vector. From this latent vector, an array of vectors is created from a similar position in the latent space. These latent vectors are then decoded into

an array of sketches, which are displayed in a new window on a grid to the user. Since the vectors are from a similar latent space as the user's sketch, they should resemble it with slight differences and so have slightly different features. This may for example generate some of the sketches in the grid as a similar sketch to the original, but with wings in various positions. The user may then select their favourite (say, an owl with wings in an upward position). The latent vector used to generate this owl is then used to generate an array of latent vectors in the same way as the previous, and a new grid of sketches are displayed in which the owls closely resemble an owl with upward wings and the process repeats. The user continually selects their favourite owls until it reaches one which has their desired features, at which point their favourite may be saved.

Note that by opening new windows on which to display generated sketches and ensuring they work entirely separately from the main sketching window, the user may very fluidly move between different parts of the application. If they decide they don't want to use the generated sketches, they may simply close the extra window and return to their sketch. They could also open several generation windows at a time, based on different stages of their drawing.

Recalling the need for a user to be able to undo a selection (for example they may have accidentally selected a sketch they didn't like) it would also be necessary for a stack of latent arrays to be maintained, such that the user can return to a previous grid.

4.3.2 Model-View-Controller Architecture

Based on the flow chart discussed in section 3.5.1, one's first thought as to the structure of the sketch application may be to use a Model-View-Controller (MVC) architecture. In this way, the function of the model can be easily abstracted such that we satisfy the requirement that it is easy to 'plug-in' different models into the application. If someone wanted to use the user interface with a different model (for instance one developed with some alternative framework to Sketch-RNN) the separation of concerns of this architecture would require them to adapt a minimal amount of code. This would also enable the application to be resilient to future developments which may occur, such as the extension of the stroke format to incorporate more stroke attributes as the developer would simply need to modify the way in which the 'View' handles stroke input.

N.B. Due to the use of the word 'model' colliding between a Sketch-RNN model and a model in the MVC architecture, in this chapter the terms 'sketch-model' and 'model' will be used for the respective purposes.

In this approach, the responsibilities of each component of the MVC architecture would be as follows:

- Model:
    1. Load a trained sketch-model

2. Generate a new sketch from a latent space using the sketch-model
3. Generate a sketch from existing strokes using the sketch-model
4. Provide information about the sketch-model (such as expected input size, latent space size etc.)
5. Save sketches
6. Normalise sketch strokes
7. Apply variance to latent spaces

- View:
  1. Provide interactive sketching interface
  2. Signal Controller to generate new sketches
  3. Signal Controller to generate sketches from existing strokes
  4. Signal Controller to save sketches
  5. Relay selected favourite generated sketch to Controller
  6. Clear the existing sketch strokes

- Controller:
  1. Upon corresponding signal from View, fetch either new sketches or sketches generated from existing strokes from Model and pass on to View
  2. Track and provide the latent spaces from which Model should generate sketches
  3. Maintain stacks of latent vectors to facilitate an 'undo selection' operation

A quick note on normalising sketch strokes - the application will work by downloading and using pre-trained sketch-models provided by the Sketch-RNN implementation. In the paper's appendix, it is described that they are specifically trained on datasets such that the standard deviation of the stroke offsets Δ_x, Δ_y have a standard deviation of 1. Due to this, when passing a user-drawn sketch to the model it is important to make sure this too has the same standard deviation, otherwise the sketch will be improperly interpreted by the sketch-model, hence the need for the Model to provide this function.

4.3.3 UML Class Diagram

Figure 17 (see appendix) shows a UML class diagram through which the structure of the sketching application can be realised. In this structure, the components of MVC are clearly separated. The Controller acts as a central entity to which entities of the Model and View are aggregated.

A notable part of this diagram is the SketchWindow. This is because whilst this class forms part of the View, it must also maintain and use a reference to the Controller in order to trigger events such as the user requesting that sketch generation. Therefore whilst other classes of the Model and View are connected to the Controller by aggregation, the SketchWindow has a more symmetric relationship with it.

4.3.4 Latent Vector Variation

In section 4.3.1 it was noted that an array of latent vectors would be generated by adding Gaussian noise to one specific vector.

To be more specific, given a latent vector z, an array of vectors $\{z_1, z_2, ..., z_n\}$ would be produced such that $z_i = z \odot \mathcal{N}(0, \sigma)$ where σ is some given variable.

Note that the mean of the distribution is 0 — the aim of this process is not to 'push' the latent vectors into any specific direction. σ, by describing the standard deviation of the noise, controls how 'different' the produced latent vectors will be. This can be used in a way such that the user may select an amount of variance they would like in the generated sketches. If they would like sketches with very few tweaks to be generated, σ can be set to some low value and vice versa.

4.4 User Interface Design

To plan for the user interface, I created simple prototypes to demonstrate the elements which would make up both the 'main' and 'generation display' windows.

4.4.1 Main Window

This window is the window on which the user will sketch and is displayed when they first open the application.

Recalling the purpose of this window, it should enable a user to:

- Sketch using mouse movements
- Generate both reinterpreted sketches and sketch completions
- Use different stroke sizes and colours (resilience to future stroke formats)
- Save their sketch
- Clear the sketchpad
- Undo their last stroke

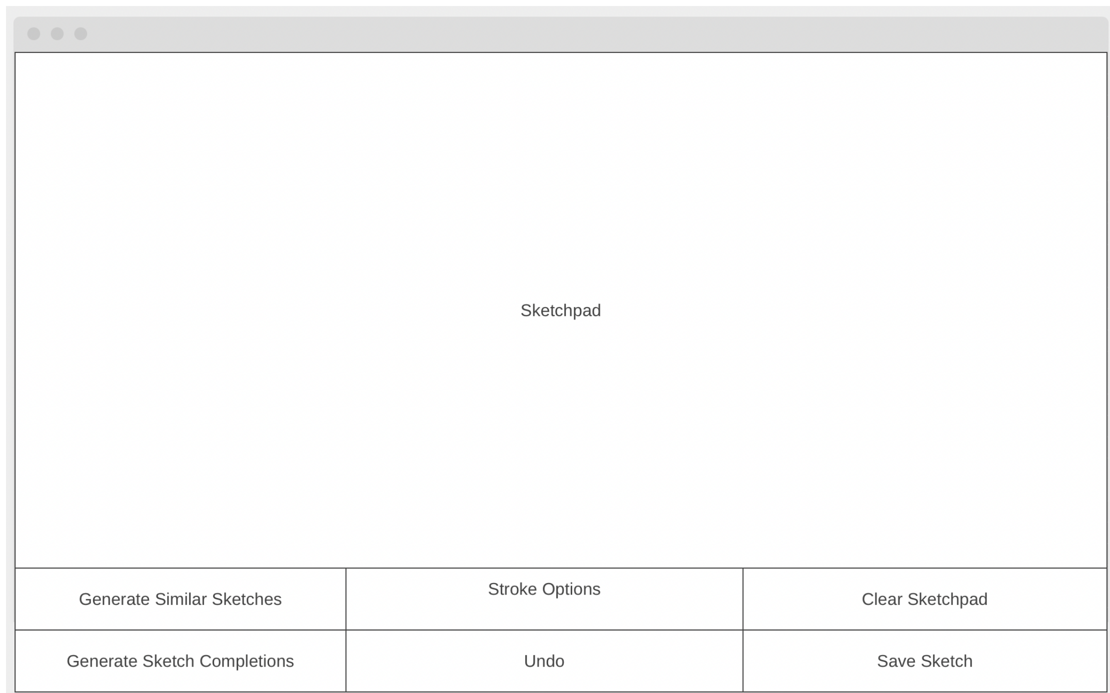


Figure 6: Main window prototype

With the above in mind, the prototype in Figure 6 was created. In this prototype, the sketchpad is prioritised within the window such that it will be easy to see and use for a variety of users.

Note that the 'stroke options' section allows users to change the colour and size of a stroke, such that the GUI will be more resilient to any future stroke format extensions.

4.4.2 Generated Display Window

This window is opened upon user request and displays a grid of generated sketches.

Recalling the purpose of this window, it must enable the user to:

- View a grid of generated sketches
- Return to a previous grid
- Save a selected sketch
- Control the variance of the generated sketches.

With the above in mind, the prototype in Figure 7 was created. In this prototype, a grid of 9 sketches is displayed such that the user may view several at once without them

being too small to see. Additionally, the central sketch is highlighted to indicate to the user that this is the 'favourite' sketch, and that this is the one which will be saved upon their request.

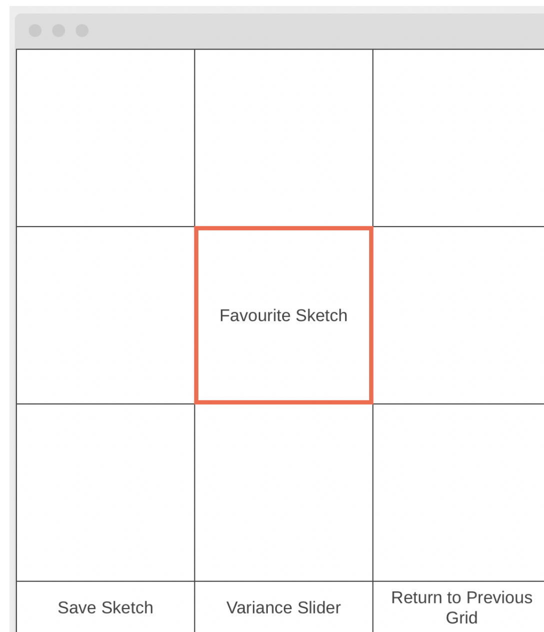


Figure 7: Sketch generation display window prototype

4.5 Transfer Learning Investigation

The investigation of transfer learning does not require such a complex structure as the sketching application. A file which trains a new model is already made available in the Sketch-RNN repository. In order to carry out transfer learning, what is required is to adapt the existing functions to apply the learning to a pre-existing model, and to apply tweaks to the loaded dataset before initiating training.

4.5.1 Adapting the Pre-Existing Training File

A model in the Sketch-RNN framework uses a number of 'hyperparameters' to control the behaviour of a model over time. Relevant hyperparameters include:

- max_seq_len → the maximum sequence length the model will accept
- z_size → size of a latent vector
- learning_rate → the **initial** learning rate of the model
- global_step → the step of training the model is on (enables training to be distributed or resumed more easily)

- save_every → frequency with which to validate and save a model
- num_mixture → number of GMMs to output at each decoder step.

In addition, as is normal in ML models there are a number of trainable parameters such as weights and biases.

The existing process for training a file in the Sketch-RNN repository is as follows:

1. Initialise model hyperparameters
2. Load dataset (using hyperparameters such as batch size to facilitate data fetching helper classes)
3. Initialise Model object (also initialises trainable parameters within the model)
4. Begin training loop (using hyperparameter 'global_step' to track the training step, calculate learning rate decay etc.)
5. Validate model with frequency defined in hyperparameter 'save_every'

Adapting this process for transfer learning is fairly simple:

1. Initialise model hyperparameters **using hyperparameters of pre-trained model**
2. **Reset hyperparameter global_step to 0**
3. Load dataset (tweaked)
4. Initialise Model object
5. **Restore trainable parameters from pre-trained model**
6. Begin training loop
7. Validate model with frequency defined in hyperparameter 'save_every'

4.5.2 Data Tweaking Module

The aim for the data tweaking module is to be able to apply transformations on a dataset such the output is distinct and yet hold somewhat similar characteristics to the original. In addition, it should be possible to tweak the dataset to some degree of intesity such that it is possible to track the effectiveness of transfer learning on datasets with different degrees of similarity.

To this end, the data tweaking module will include two transformations: rotation, and scaling. In this way, results can be compared between a single transformation being applied as opposed to both transformations, with the idea that it will be more difficult for a model to adapt to both transformations as opposed to just one.

Scaling

The horizontal scaling is fairly simple to apply. Given a vector of pen offsets in a particular stroke $\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix}$, a scaling matrix $\begin{bmatrix}a & 0 \\ 0 & b\end{bmatrix}$ can be applied such that $\begin{bmatrix}a & 0 \\ 0 & b\end{bmatrix}\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix} = \begin{bmatrix}a\Delta_x \\ b\Delta_y\end{bmatrix}$ gives the pen offsets of the stroke scaled by a factor of a horizontally and factor b vertically.

Rotation

In order to apply a rotation to the pen offsets $\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix}$ one must apply the rotation matrix $\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$ such that $\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix} = \begin{bmatrix}\Delta'_x \\ \Delta'_y\end{bmatrix}$ gives a **counter-clockwise** rotation of the stroke by θ degrees.

In combination

The above transformations can be combined into a single calculation. In order to scale *then* rotate the pen offsets $\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix}$ one can apply the matrix $\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}a & 0 \\ 0 & b\end{bmatrix} = \begin{bmatrix}a\cos\theta & -b\sin\theta \\ a\sin\theta & b\cos\theta\end{bmatrix}$ such that $\begin{bmatrix}a\cos\theta & -b\sin\theta \\ a\sin\theta & b\cos\theta\end{bmatrix}\begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix} = \begin{bmatrix}\Delta'_x \\ \Delta'_y\end{bmatrix}$ gives the pen offsets scaled horizontally by a, vertically by b, then rotated **counter-clockwise** by θ.

4.6 Training Plan

After creating the data tweaking module and adapting the existing training process, all that is left to do is run the program and collect evaluation data throughout the process. In order to do this, the program will be run on two datasets; one with a rotation transformation only, and one with a scaling and rotation transformation.

The scaling transformation will specifically scale the sketches horizontally. If scaled both horizontally and vertically, the effect would just be to enlarge the sketches, which would not create a significant distinction between the datasets. There is also a question as to the magnitude of the scaling, as well as which θ by which to rotate the sketches. These magnitudes should be dynamic — to scale and rotate all sketches in the dataset by the same value would be a simple enough pattern such that the training of the new model would not be sufficiently tested.

In light of this, the magnitude of scaling and degree of rotation will have a random value for each sketch. The range of these values is difficult to necessarily plan — to some extent it will be advantageous to adapt these ranges based on training results. For example, if the difference in training cost between the two datasets (one containing just one transformation, the other containing two) is very small, it *could* suggest that the scaling magnitude is not sufficient to challenge the model. If a model trains equally as effectively on both datasets regardless of the scaling magnitude, that in turn *could* suggest that the challenge of adapting to a rotated dataset is large enough such that additional scaling transformations are negligible. All in all, I would suggest that

there should not be such a restrictive plan for this section of the project, and instead it may be more effective to adapt the investigation based on the conditions under which interesting results arise.

5 Implementation

This chapter aims to describe the process through which the project was created following from the previously outlined design.

5.1 Sketch Application

5.1.1 User Interface

Main Sketch Window
The first step was to create the main sketching window. This was implemented in a class named 'SketchWindow'. Using the tkinter 'grid' layout manager, sections of the window were created in a way corresponding to the design prototype. In order to display a sketch and allow a user to draw, a *tkinter.Canvas* object was used. An event can be bound to the canvas object such that whenever a mouse click is made *and* the mouse is moved, coordinates can be learnt which track the movement of the user's cursor as they dragged it across the screen. By maintaining a cursor attribute within the SketchWindow class, stroke-3 format data can easily created by simply updating the cursor position with each movement and adding a 'pen lift' stroke whenever the user lifts the mouse button. This stroke information can then be drawn using the *tkinter.Canvas.create_line* function. Additionally, each time the cursor position is updated that is a sign for the position of the cursor to be stored in the attribute *SketchWindow.__sketch*.

One thing to note is that the user's sketch is not stored in *SketchWindow* in stroke-3 or stroke-5 format. This is because as described in section 1 of the appendix of the paper, the Sketch-RNN implementation uses Ramer-Douglas-Peucker[22](RDP) simplification to prepare sketches. RDP is an algorithm which, given a distance ϵ, simplifies a line represented using a series of points such that the new line is no further than ϵ distance away from the original line at any point. In other words, the algorithm allows sketches to be simplified by smoothing the strokes according to ϵ where a smaller value requires a more conservative simplification. As RDP works specifically on a sequence of rather than pen offsets, it makes more sense to store the users sketch in this way and then only convert the sketch into the stroke formats when necessary.

Upon showing this interface to peers for design opinions, what immediately become apparent is that for many, the first instinct was to attempt to resize the window. With a regular Tkinter canvas, this is not properly responded to. By default, the grid manager sets row and column 'weights' to 0. This essentially means that rows and columns placed by the grid manager will attempt to claim 0% of any space which becomes available beside them; upon resizing the window, the canvas does not stretch to fill this new area. However, even with non-zero weights set, a default Tkinter canvas will extend to fill the space rather than stretch. This is not satisfactory — one would hope that if they make a large stroke from the top of the canvas to the bottom, it should remain as just that even when the window is resized. Additionally, if a user reduces the size of a window the sketch should be squeezed into the space as opposed to the edges

of the sketch simply being 'chopped off'.

Therefore, the class *ScalingCanvas* was created. This is a subclass of *Tkinter.Canvas*, however an event is bound to any resizing of the window. This event uses the new dimensions of the canvas to calculate the factor by which they were changed compared to the previous dimensions of the canvas. Using this factor, all the components of the canvas can be resized. For example, if a line is drawn and then the canvas is resized such that it is 2 times as wide and 1.5 times as tall, the stroke's appearance will be stretch such that it still fills the same proportion of the canvas after resizing. Additionally, these changes must be reflected as the strokes points are recorded to represent the user's sketch. The canvas notifies the parent window *SketchWindow* know of any resizing which takes place. The SketchWindow then maintains *x_factor* and *y_factor* attributes such that all stroke inputs made by the user are scaled in a way that the strokes fed into the model are scaled to match the appearance of the sketch that the user sees.

As in the design, along the bottom of the window are the widgets used to provide functionality for the user. Separated into logical sections are buttons corresponding to sketch reinterpretation, sketch completion, stroke colour, clearing the sketchpad, and saving the user's sketch, alongside a slider to determine stroke size.

It should be notes that the stroke colour and stroke size widgets have no effect on the internal representation of the sketch. This is because the Sketch-RNN framework is currently limited to strokes of a uniform thickness and colour. However, these widgets are included in the window to afford the user more expressivity in their sketches, alongside allowing the tool to be more easily adapted to any future extensions of stroke format which may take place at the expense of misleading a user into thinking it would make a difference.

The 'Clear Sketchpad' and 'Save Sketch' provide their namesake functions as expected — the sketchpad is simply cleared by removing all existent elements from the canvas and resetting the internal sketch representation to an empty array, while the sketch is saved as an svg in a similar manner to a function in the default Sketch-RNN implementation.

The sketch reinterpretation and completion buttons proceed to open a new window, as described in the design, along with the necessary information to perform its function. Specifically, the buttons open a new instance of the *ExplorationWindow* class. Sketch reinterpretation requires passing information of the latent space onto which the user's sketch is decoded, and so when the corresponding button is pressed the user's sketch is converted into stroke-5, encoded into a latent vector, and an *ExplorationWindow* is initialised with this vector. Otherwise for sketch completion, no latent vector is required from the new window, however the user's existing strokes must be passed to the new instance.

<u>ExplorationWindow</u>
This class carries out the function of fetching generated sketches from the model and

displaying them in a grid, as previously discussed.

Both sketch reinterpretation and completion would be carried out in an almost identical manner from the point of view of the user interface. Therefore, both processes are carried out using the same window. The class *ExplorationWindow* performs these functions. It is a child of the main stroke window, however performs independently with no further information from its parent needed after initialisation.

If the window is made to carry out sketch reinterpretation, then it is initialised with the latent vector corresponding to the user's sketch. This is maintained as an class attribute. From here, the window may apply deviation to the latent vector (as described in section 4.3.4) to build an array of 9 vectors. These vectors then each correspond to a canvas in a grid of 9 canvases which dominates the window.

Otherwise for sketch completion the window is initialised using the user's existing strokes. These strokes are maintained as an attribute. Since the window in this case is *not* initialised with a latent vector, the array of 9 latent vectors is created using a Gaussian distribution such that each latent vector $z = [z_0, z_1, ..., z_i]$ where $z_i \sim \mathcal{N}(0, 1)$.

From this point, both types of *ExplorationWindow* have an array of latent vectors. They use this array to generate sketches using the model — in sketch reinterpretation the model generates the sketches from scratch, and in sketch completion the model generates the sketches using the stored existing strokes attribute. Thus, using the model the window builds up an array of sketches where each sketch corresponds to the latent vector in the matching index of the latent vector array.

These sketches are then displayed in order onto 9 *ScalingCanvas* instances, which are again organised using the Tkinter grid manager. Events are bound to each canvas such that the clicked canvas notifies the class that the ith sketch (and therefore its corresponding latent vector) is the favourite. From this point, this vector can be set as the 'favourite' vector — variation is then applied to this vector to build up a new array of latent vectors, and the cycle continues with new sketches generated from the new array of vectors.

One nuance to note is that when the array of latent vectors is created using variation applied to one 'favourite' vector (except in the initialisation of the window for sketch completion, in which case all latent vectors are initialised at random) this vector is preserved. In other words, where are array of latent vectors $z_0, z_1, ..., z_8$ is generated such that $z_i = z \odot \mathcal{N}(0, \sigma)$, $z_4 = z$ is specifically maintained. This is because the fourth latent vector and corresponding sketch are displayed in the centre of the screen, and represent either the direct reconstruction of the user's sketch (in the case of sketch reinterpretation) or the sketch which the user selected as their favourite in the previous display grid. In other words, if the user has just selected a sketch as their favourite, the worst thing to do would be to then generate an array of vectors which does *not* include the vector they just selected, such that the sketch they just marked as preferred disappears. Instead, by keeping the exact vector corresponding to their favourite sketch

in the highlighted centre of the grid means they can retrieve and save the sketch they specifically chose.

The window also includes buttons along the bottom which display ‘Save Sketch’ and ‘Return to Previous Grid’. To return to a previous grid, a stack of arrays of latent vectors is maintained by the class. Each time an array of vectors is generated, they are ‘pushed’ onto the top of the stack. This means in order to return to a previous grid all that needs to be done use the stack’s ‘pop’ method to retrieve the most recent array of vectors, from which point sketches can be generated again by the model and then displayed as usual. The ‘Save Sketch’ button causes the ‘favourite’ sketch to be saved. This is the sketch which is displayed in the centre of the grid, highlighted in red to indicate this being the favourite sketch.

Finally, also located on the bottom of the window is a *Tkinter.OptionsMenu* which allows the user to select ‘low’, ‘medium’, and ‘high’ variance. This adapts the standard deviation σ of the Gaussian noise applied during the latent vector array generation to be $0.2, 0.5, 0, 75$ respectively. These values were chosen purely through experimentation — at higher values the sketches generated by the model become increasingly unpredictable (sometimes leading to low quality sketches), and so these values were found to give the user enough room to control the degree of difference within the sketches whilst restricting them to variances which actually produce meaningful outputs.

5.1.2 Data Processing

Data Utilities
The class *DataUtilities.py* was used to perform useful functions to manipulate and handle data which would be used throughout the application. It is implemented as a stateless class with exclusively static methods for this purpose. Its methods include:

- *normalise(sketch)* → normalises a sequece of strokes to have a standard deviation of 1 as described in section 1 of the Sketch-RNN paper appendix.
- *strokes_to_svg(sketch, filename)* → saves a sketch in stroke-3 format as an svg file
- *selectiveRDP(sketch, epsilon)* → simplifies a sketch using the aforementioned rdp algorithm, while protecting ‘pen up’ strokes to maintain the sketch structure
- *simplify_as_possible(sketch, max_len)* → simplify a sketch as little as possible to be under a given maximum sketch length
- *points_to_strokes(points)* → turn a sequence of points into stroke-3 format
- *stroke_3_to_stroke_5* → convert stroke-3 format to stroke-5

5.1.3 Model Interaction

The following classes were created to abstract the responsibility of dealing with the model away from the user interface. Although the flexibility of Python means class

interfaces do not hold much functional value and so there is no concrete way to create them in standard Python, I decided to use pseudo-interfaces to improve the readability of the code and make it more adaptable for others to use the tools with their own implementations of a sketch-model. I created these interfaces by creating a class which defines the necessary methods to immediately raise a *NotImplementedError*. The classes implementing the interface then inherit from this superclass, and so must override the method definitions.

ModelFactoryInterface
This class is the pseudo-interface from which a *ModelFactory* should inherit. The purpose of this class is to load and maintain the sketch-model(s) and tensorflow session. The methods included are:

- *getEncodeModel()* → return the sketch-model used to encode a single sketch into a latent vector
- *getDecodeModel()* → return the sketch-model used to decode a latent vector into a sketch
- *getSession()* → return the tensorflow session to which the models are linked

ModelHandlerInterface
This class is a pseudo-interface from which a *ModelHandler* should inherit. The purpose of this class is to interact with models maintained by a *ModelFactory* to encode sketches onto the latent space and decode latent vectors into sketches. The methods included are:

- *getMaxSeqLen()* → get maximum expected sequence length of the encoder model. Required in some *DataUtilities* methods.
- *getLatentSize()* → get size of latent space expected by decoder model. Required during latent vector array generation.
- *sketchToLatent(sketch)* → encode a sketch onto the latent space
- *generateFromLatent(z, existing_strokes)* → generate sketch from latent vector z. Sketch completion from $existing_strokes$ if any given.

Interface Implementation The aforementioned interfaces were then implemented according to this specification. Particularly, the Sketch-RNN demonstration notebook was used as a reference for the loading and sampling of a pre-trained model. The model pre-trained on a dataset of Owl sketches was chosen to be used, however by changing the constant value `MODEL_DIR` in the code different models provided by the Sketch-RNN implementation can be used.

The reason for there being separate methods for encoding and decoding is because in the Sketch-RNN implementation, certain hyperparameters may be changed depending on what exactly the model is being used for, and so whilst the two methods *could* return the same model this is not necessarily always the case. In this case, the two

share the same hyperparameters except for the decoding model having the hyperparameter ‘max_seq_len’ set to 1. This protects the model from being used for encoding, as the model will throw an error if fed a sequence consisting of anything more than a single ‘end of sequence’ stroke (`[0, 0, 0, 0, 1]` in the Sketch-RNN framework). A future developer implementing this interface may well decide not to impose such restrictions.

5.1.4 Overall Structure

Overall, the structure of the application does not follow the design overly closely. Specifically, while the MVC architecture is somewhat followed, there is far from perfect separation of each component. Figure 8.2 (see appendix) shows the final UML class diagram of the project. As frequently the Model and View were required to reference and use each other, the lines between the Controller and the other components became somewhat blurred; it was decided that to strictly impose an MVC architecture on the project would require too many unecessary obstacles while possibly even reducing the readability of the code.

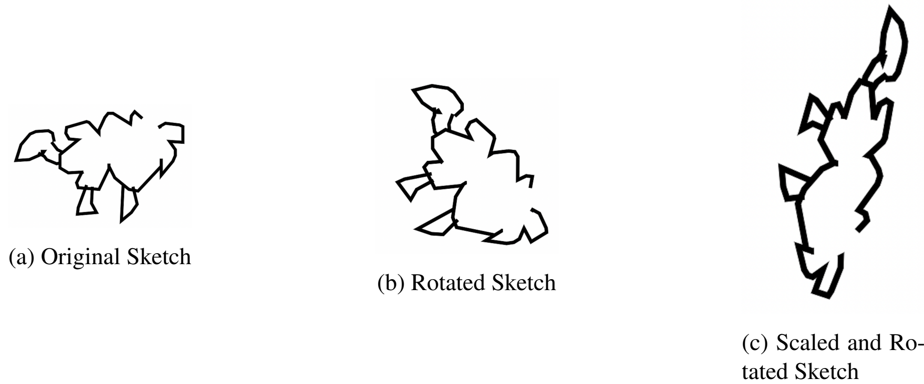
5.2 Transfer Learning

This section briefly describes implementation details of the transfer learning section of the project. The implementation closely followed the plan described in the design.

5.2.1 Data Tweaking

The original implementation of Sketch-RNN uses a *DataLoader* class to handle a dataset by performing functions such as normalisation and sorting the dataset into batches. In order to make modifications to the dataset, the class *DataTweaker* was used. It inherits from *DataLoader*, with the only adaptation being a *tweakData* method being used at the end of the constructor in addition to the subroutines *c_rotation* and *scale_horizon* to perform the transformations.

As discussed in section 4.5.2, the rotation method uses a matrix applied to the pen offsets. However, this was unnecessary for the scaling transformation, as a more readable solution can trivially be achieved in a single line:
`stroke[0] := scaling_factor * stroke[0]`.



(a) Original Sketch

(b) Rotated Sketch

(c) Scaled and Rotated Sketch

Figure 8: A single sketch's resulting appearance in each dataset

The rotations are performed with uniformly randomly generated $\theta \in [-135°, -45°]$ for each sample (note that negative θ corresponds to a clockwise rotation) and the scaling was performed to a uniformly randomly generated factor $a \in [1, 3]$. Examples of the transformations of a single sketch can be seen in Figure 8.

5.2.2 Training

The training loop was implemented as described in the design, and the aforementioned adaptations were made to the training file in the Sketch-RNN repository. All training was carried out on the Aaron Koblin sheep dataset[23] using pre-trained models provided in the Sketch-RNN implementation.

One issue encountered is that due to the way in which Tensorflow version 1 restores model parameters, the validation model could not be used in the same way as in the training loop in the Sketch-RNN repository. This is because the pre-trained validation model have the *is_training* hyperparameter set to 0, which means it does not contain a cost attribute. This hyperparameter cannot be changed without Tensorflow being unable to restore the model checkpoint correctly. To solve this, I applied the validation data to the regular training model and the initialised *DataLoader* class differently to reflect this.

6 Evaluation

6.1 Testing Strategy

The strategy to test both components of the project will differ greatly.

For the sketching application, the functionality of the program will be tested by usinf end-to-end testing to check that the promised functions are executable. Additionally, qualitative testing will take place. Due to the nature of the application, some parts

of the evaluation will be somewhat subjective — the quality of generated sketches cannot necessarily be assessed with conrete tests, however examples of generated sketches will be shown and commented on from as neutral a viewpoint as possible.

As for the transfer learning investigation, two datasets are tested on — one where both a horizontal scaling and rotation are applied, and one with just a rotation. Two models will then be trained on each dataset — one from scratch, and one by using transfer learning on an existing pre-trained model. The validation costs throughout training will then be displayed and discussed.

6.2 Sketching Application

6.2.1 Overall Appearance

Figures 12 and 13 (see appendix) show the appearance of the main sketching window and the generated sketch display window respectively. As expected and discussed in section 4.2.2, the windows have a slightly outdated appearance. Otherwise, the window follows the design closely and the components of the window are composed such that the sketch canvas takes priority in the screen and is clearly visible, while the buttons along the bottom of the windows are given enough space to be simple to use. However, one could argue that the grid display of the sketch generation window leaves a small space for each canvas, meaning users (especially those who have issues with close vision) may struggle to see the sketches properly. While the window is resilient to being resized, the scale-able canvases may leave the sketches greatly stretched. Upon reflection, the buttons of this window could have been placed along the right side such that the display great has a larger area and so each sketch does not need to be squeezed into such a small space.

One issue is that the larger stroke sizes of the main window appear jagged. This is because Tkinter does not use anti-aliasing, which means any diagonal line will not be smoothly rendered, and the poor appearance is particularly noticeable with larger line thicknesses. This is an issue inherent to Tkinter which was not uncovered during research.

6.2.2 End-to-End Testing

Test Number	Test Description	Test Passed?
1	Make single straight stroke	Yes
2	Make curved stroke	Yes
3	Make multiple strokes	Yes
4	Change stroke thicknesses	Yes
5	Change stroke colour	Yes
6	Resize canvas; window components remain in same structure	Yes
7	Resize structure with strokes already made; strokes are stretched based on resize	Yes
8	Make stroke and save sketch	Yes
9	Clear sketchpad	Yes
10	Draw sketch and trigger sketch reinterpretation	Yes
11	Draw strokes and trigger sketch completion; existing strokes are present in generated sketches	Yes
12	Resize sketch generation window; sketches stretch appropriately	Yes
13	Click a generated sketch; new sketches generated	Yes
14	Save generated sketch	Yes
15	Return to previous generated sketch grids	Yes
16	Interact with window during sketch generation	No
17	Attempt to return to generated grid when no previous grid exists; application notifies and ignores	Yes
18	Make strokes in various colours	Yes
19	Make coloured strokes and generate sketches; strokes are treated as though they're black	Yes
20	Make strokes in both different sizes and colour	Yes
21	Attempt sketch completion after drawing more strokes than the model's stroke limit; application manages to continue generation	Yes
22	Make large curved stroke	Partial

As can be seen in the table, most of the functionality of the application works. However, there are two tests where the behaviour of the application is not ideal.

Test 16 shows that the window cannot be interacted with during sketch generation. After a user selects a favourite sketch, the application become unresponsive for around one second. This is due to a limitation of Tkinter discussed in section 4.2.2 — when the user selects a sketch it takes a moment for the model to generate the new sketches, during which the mainloop of the window blocks. The same limitation means it is difficult to incorporate something like a loading symbol. This means upon sketch selection the application appears to have not recognised the user's input, which may prompt them to click again which will then register a second undesired sketch selection.

Test 22 is marked as partially failed, because while the line is correctly created it suffers the previously noted anti-aliasing issue.

6.2.3 Sketch Generation Assessment

Inherently, the performance of the application depends on the performance of the sketch-model used. With the pre-trained models provided in the Sketch-RNN implementation, the generated sketches can be somewhat inconsistent — occasionally all generated sketches are of a good quality, and at other times most generated sketches can be completely incoherent. However, there is certainly situations in which the generated sketches have a higher (or lower) probability of being of sufficient quality.

Experimentation was carried out using a model in the Sketch-RNN implementation pre-trained on the a dataset containing sketches of owls. Overall, it's generally found that it is almost always *possible* to produce a decent sketch, but it is entirely possible that low quality sketches are produced along the way, and in certain cases it is not easy for the user to find a sketch which is satisfactory; it sometimes takes several rounds of generation to produce any respectable output at all.

The cases in which the model does not produce satisfactory outputs are usually when the high variance option is selected. Predictably, the generated sketches are more inconsistent on this setting, and so often in this situation the generated sketches will include some which are completely incoherent. Usually, provided the user continues to select 'good' sketches, a quality output can still be found. However it is sometimes the case that one wrong selection can 'push' the latent space into an area in which very few of the generated sketches look good, and so it can be hard to return back to a 'good' area of the latent space. Figure 14 (see appendix) shows an example of this — on high variance, after selecting a strange sketch, the generated sketches have become completely unrecognisable, and a user may not know which sketches to select such that they may find their way back to normality. In this situation the only option is sometimes to use the 'undo' function to return to previous grids until the sketches return to normality.

One possible explanation of this issue may be that through several selections of sketches, the 'favourite' latent space may have certain values pushed to extremes by conincidence. For example, the value in a particular index may by chance be increased with each sketch selection. After some time, this means the latent vector leaves the area of the latent space which the decoder is designed to interpret. As is described in section 3.4 of the Sketch-RNN paper, the models are trained to limit the latent space's size, meaning by design once certain values of a latent vector become too extreme they leave the area where the model is designed to 'understand'.

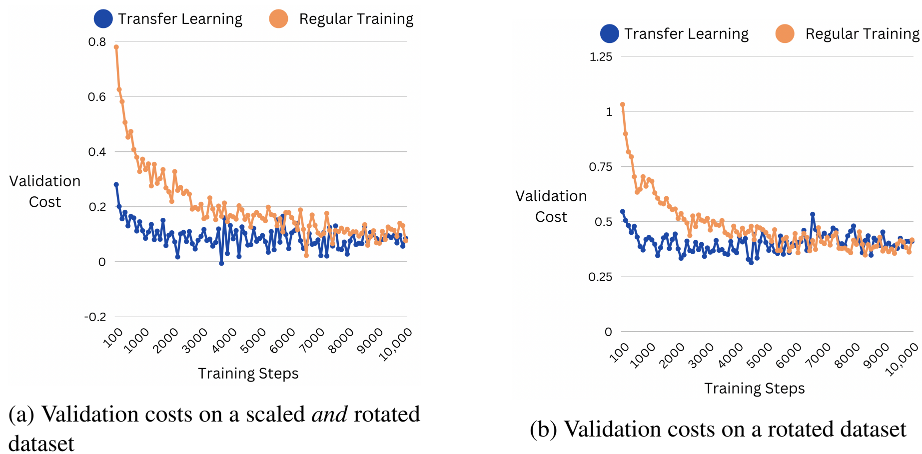
If I were to re-implement the project, I would solve this issue by using a more sophisticated method when generating an array of latent vectors. Rather than blindly applying Gaussian noise, a good option instead may be to apply the noise in such a way that extreme values are unlikely. In other words, when creating an array $[z_1, z_2, ..., z_9]$ based on some $z = [x_1, x_2, ..., x_n]$, a given $z_i = [x'_0, x'_1, ..., x'_J]$ where $x'_j = x_j + \beta_j$ such that $(\lim_{x_j \to 1} P(\beta_j > 0) = 0) \wedge (\lim_{x_j \to -1} P(\beta_j < 0) = 0)$ for some arbitrary probability distribution of β. This would force the standard deviation $\sigma \leq 1$, which matches the

standard deviation sought after when applying the Kullback-Leibler loss when training a VAE.

The cases in which high quality sketches are produced by the model tend to be on the low and medium variance settings. Here, the behaviour of the model is more controlled and adaptations of the sketches are smaller. In general, the best way to use the application was found to be by using medium variance until the structure and largest features of a sketch are as desired, and then using low variance to specifically hone in on the smaller desired sketch features. This is demonstrated in figure 15 (see appendix) — all the latent vectors have remained in the area of 'good' model behaviour and so are of decent quality, with a small amount of variance in each sketch.

6.3 Transfer Learning

Figure 9 show validation data during training with a new model and pre-trained model on datasets scaled and then rotated and scaled only respectively.



(a) Validation costs on a scaled *and* rotated dataset

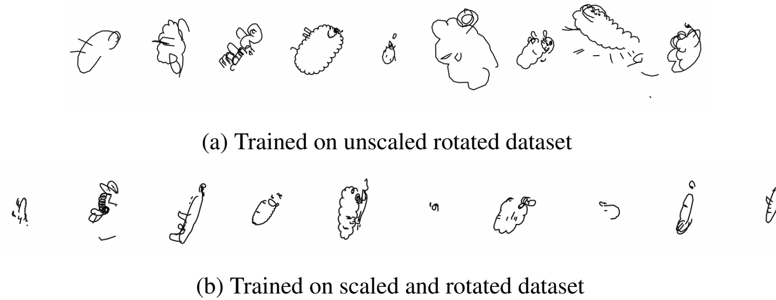
(b) Validation costs on a rotated dataset

Figure 9: Validation costs during training

Both graphs show that the models trained using transfer learning learn achieve significantly lower validation costs from the very first training step as expected. However, these models stagnate extremely quickly; their final validation costs are only a slight improvement from their initial costs. This is expected — the idea is that pre-trained models already 'know' how to draw a sheep, and so just need to tweak their parameters slightly to generate them in a different way.

In training on both datasets, it was consistently found that the validation costs of a model training from scratch caught up with the costs of a model using transfer learning after around $5,000 - 6,000$ training steps. Therefore, after looking solely at the

validation costs throughout training, one could conclude that transfer learning significantly increases the speed of training in the short-term, but will not make a meaningful difference in the long term.

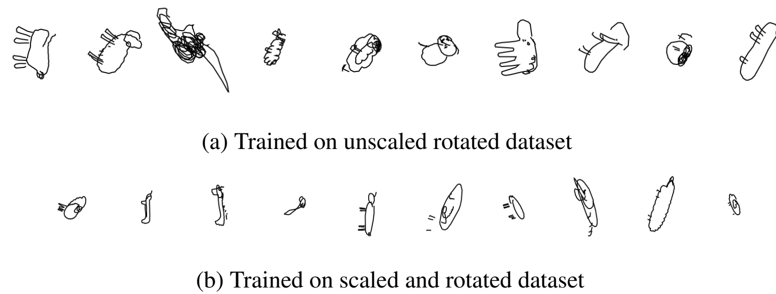


(a) Trained on unscaled rotated dataset

(b) Trained on scaled and rotated dataset

Figure 10: Sketches sampled on models trained using transfer learning

Figure 10 shows sketches generated by the models trained using transfer learning on each dataset after $10,000$ steps. Both models appear to have learned to draw rotated sheep, and the model trained on the scaled dataset generates longer and thinner sketches.



(a) Trained on unscaled rotated dataset

(b) Trained on scaled and rotated dataset

Figure 11: Sketches sampled on models trained from scratch

Figure 11 shows sampled sketches from the models trained from scratch after $10,000$ training steps. The sketches appear to be roughly the same quality as sketches generated using transfer learning.

There is one very unexpected aspect of the training data — the validation costs of models trained on the *scaled* dataset are actually significantly lower than those trained on the *unscaled* dataset; scaled, the validation costs reaches stagnation at around 0.1, compared to around 0.4 on the unscaled dataset, and these results are consistent across many training runs. Where the aim of having these distinct datasets was specifically to create one which is more difficult to adapt to than the other, these results are the opposite to those expected.

These results suggest that — contrary to expected — the *scaled* dataset was actually more easily to learn than the *unscaled* dataset. However in hindsight, there may be a straightforward reason for this which was overlooked during design.

The Sketch-RNN framework expects sketches to be normalised — that is, the pen offsets of strokes in a given sketch have a standard deviation is 1. This has the effect of 'squeezing' the sketches to be a certain size. The 'DataTweaker' class loads a dataset, scales the sketches, and *then* normalises them. Since the scaling has the effect of increasing the sketches' size horizontally, and then the sketch strokes are normalised, this has the effect of 'losing vertical detail'. To demonstrate this more clearly: if one were to scale a sketch by an extremely large number, and then normalise it such that strokes have a standard deviation of 1, then the sketch would effectively appear like a series of horizontal lines with few features or specific patterns, on which it would then be more easy for a model to train. This, to a less extreme degree, may explain the reason that training on the scaled dataset resulted in lower training costs.

This proposed explanation is solely speculation. However, the fact is that regardless of the explanation, the fact is that the datasets *do* appear to differ in their level of difficulty, and resulted in different final validation costs during training. To that end, it can be seen from the graphs that there was a minimal impact of the difficulty of a dataset on the effectiveness of transfer learning. In both cases, transfer learning essentially offered a 'headstart' in training by several thousand training steps, after which point the newly-trained model has improved sufficiently such that no noticeable improvement was gained from the method.

Overall, these results suggest that transfer learning is useful for the fast adaptation of new models to provide variation on those previously trained. Additionally, it appears that this utility is somewhat invariant to the degree of difference between datasets used during the training process. Additionally, in the long-term no real advantage was gained from using transfer learning.

From this, one could imagine a few uses for transfer learning in this field.

Firstly, the method is likely to be useful in cases where data in the correct stroke format is scarce. Suggested by the data, in the current framework the training costs achieved by a model trained using transfer learning for around $1,000$ steps would only be reached by a newly-trained model after around $5,000 - 6,000$ steps — a significant improvement. In cases where only a small number sketches would be available as training data, transfer learning would make training to a reasonable standard possible.

Secondly, one could use transfer learning in this way to achieve rapid-adaptation of sketch generation models to increase the flexibility of such applications. For example, some dataset could be tweaked in several ways. Then, through using transfer learning to adapt some pre-trained model to each tweaked dataset, a user could be presented with sketches generated by several models and choose their favourite model, meaning sketches are specifically catered to a user. This could be used in combination of the

‘Sketch Application’ part of this project to afford more choices in preference for a user.

7 Conclusions

7.1 Achievements

Overall, the Sketch Application provided the expected functionality. It can be used to generate quality sketches based on a user's desired features, remains simple and easy to use. The application works fluidly and can work closely with a user to help them produce simple, aesthetically pleasing sketches. The application was also coded such that it is easily adaptable to using different models, and support was put in place such that future extensions to the stroke format to incorporate stroke thickness and colour would a small amount of code to be changed.

Additionally, it was shown that transfer learning has specific situations in which it could be utilised effectively, however in the general case where there is sufficient training data it would not be necessary. That being said, there does not appear to be any disadvantage to using the technique.

7.2 Future Work

The principle way in which to extent this project would be in the development of a new stroke format. While there is a lack of data to support this endeavour, transfer learning could be used to alleviate this scarcity.

From there, a model trained to use different stroke formats could be applied to the sketch application, such that the user is afforded a higher degree of expressivity.

Finally, the Sketch-RNN repository used throughout the project is written in Tensorflow version 1. It would be worthwhile to migrate this fully to version 2, as this would improve the performance of the models as well as making the code much more accessible.

7.3 Challenges

The biggest challenge in this project was that I underestimated the extent to which version 1 of Tensorflow differs from version 2. During the initial stages of this project, research was largely based around learning the fundamentals of machine learning and Tensorflow version 2, under the assumption that it would be simple to migrate the existing Sketch-RNN repository (written in version 1). However, it was only deeper into the project that I learned that the two versions differed so significantly. Although there is a script provided on the Tensorflow website to automate this process, it fails to be effective for of the vast majority of the code. Not only does this mean the models' performance could be greatly improve using the newest version, but more importantly the repository would be much more readable and accessible to developers hoping to work with the code.

While refactoring the repository to version 2 was an option, this would have taken up a significant amount of development time. Instead, the project's aims were changed from working to *adapt* the Sketch-RNN framework to working *with* the framework. As a result, minimal amounts of changes were made to the existing code, and instead time was focused on using it similarly to an API. This meant development relied significantly upon the code included in the Sketch-RNN Jupyter Notebook.

8 Appendix

8.1 Code Repository

The source code for this project can be found in the folder ‘code’ accompanying this report. This folder includes a read me containing simple instructions for running the application and transfer learning files.

8.2 Diagrams

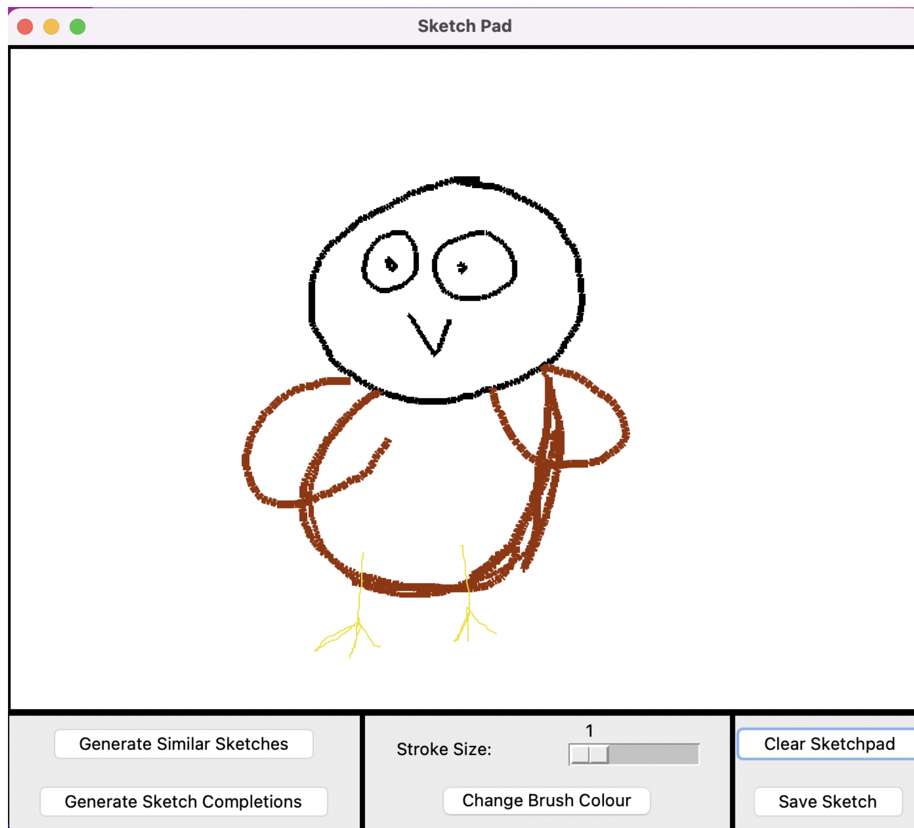


Figure 12: Sketch application main window appearance

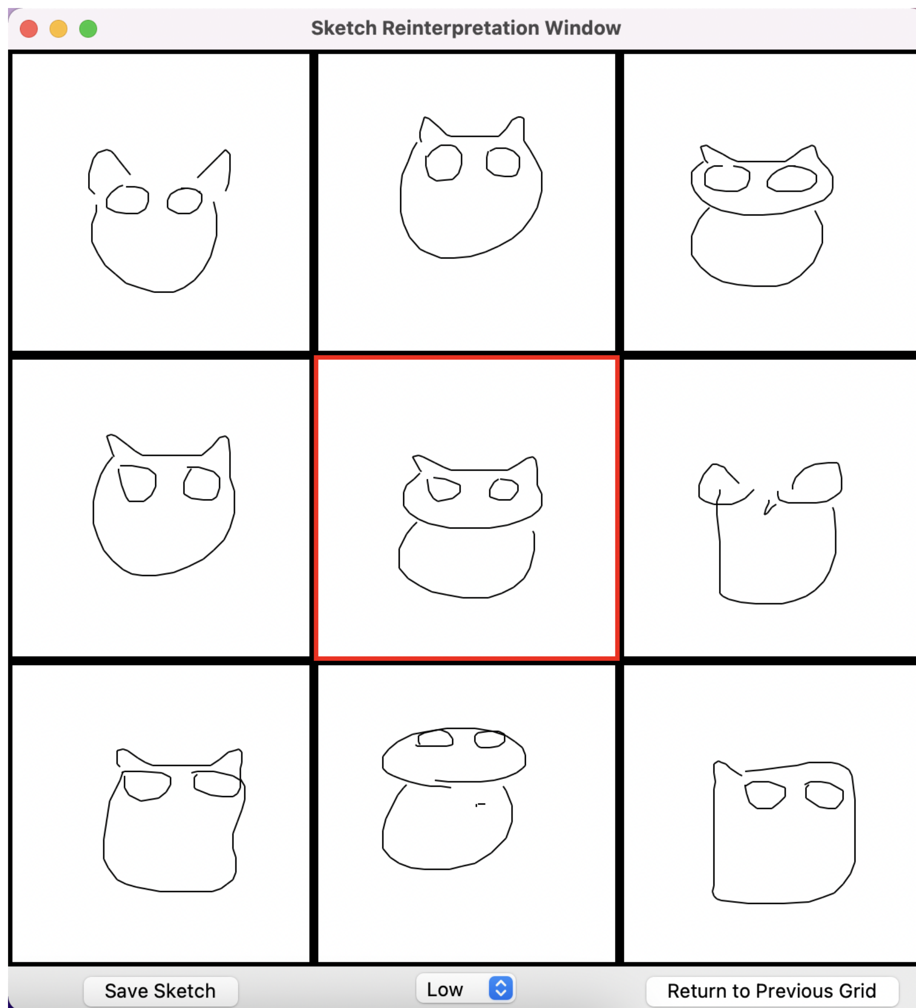


Figure 13: Sketch application generation window appearance

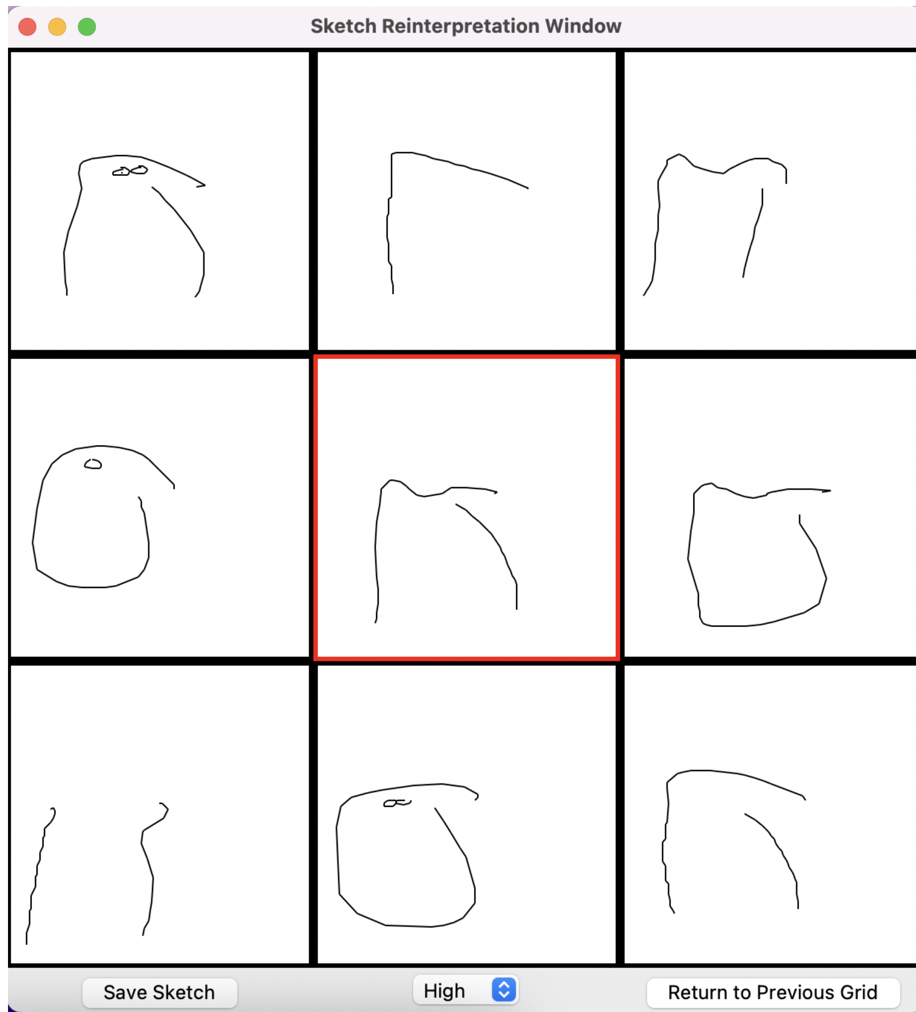


Figure 14: A ‘bad’ area of the latent space

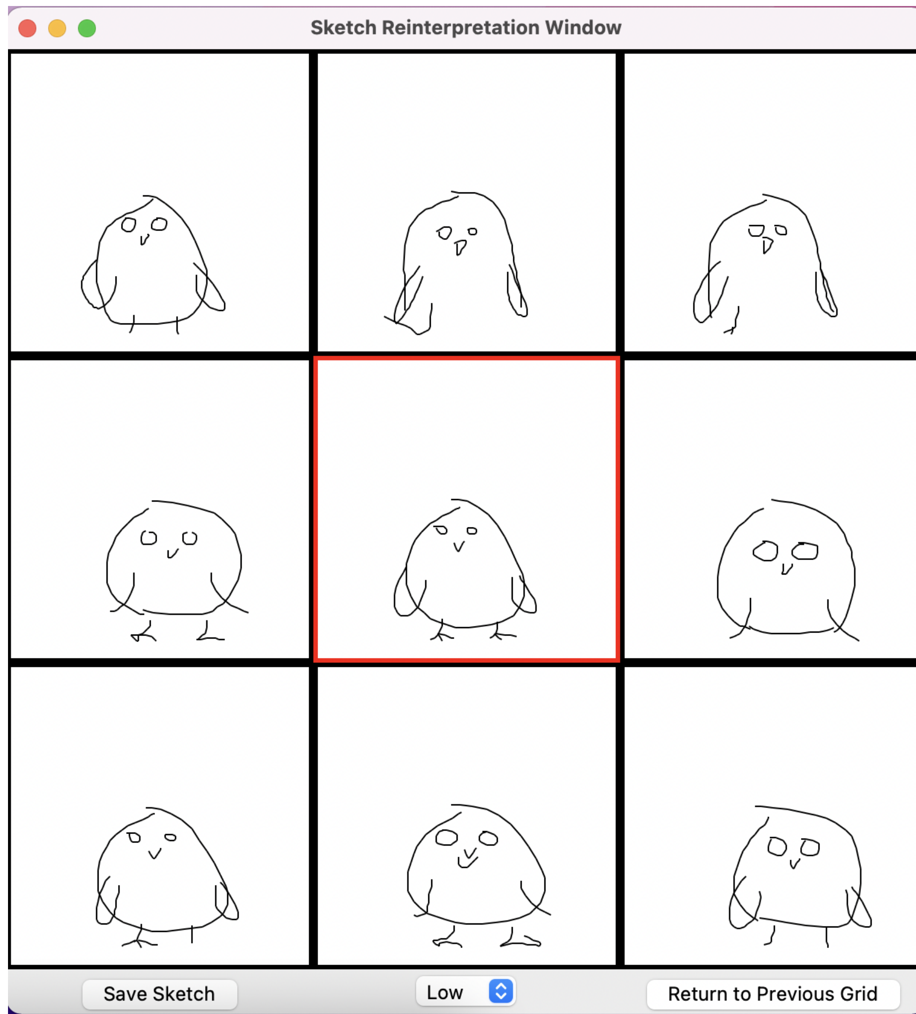


Figure 15: A ‘good’ area of the latent space

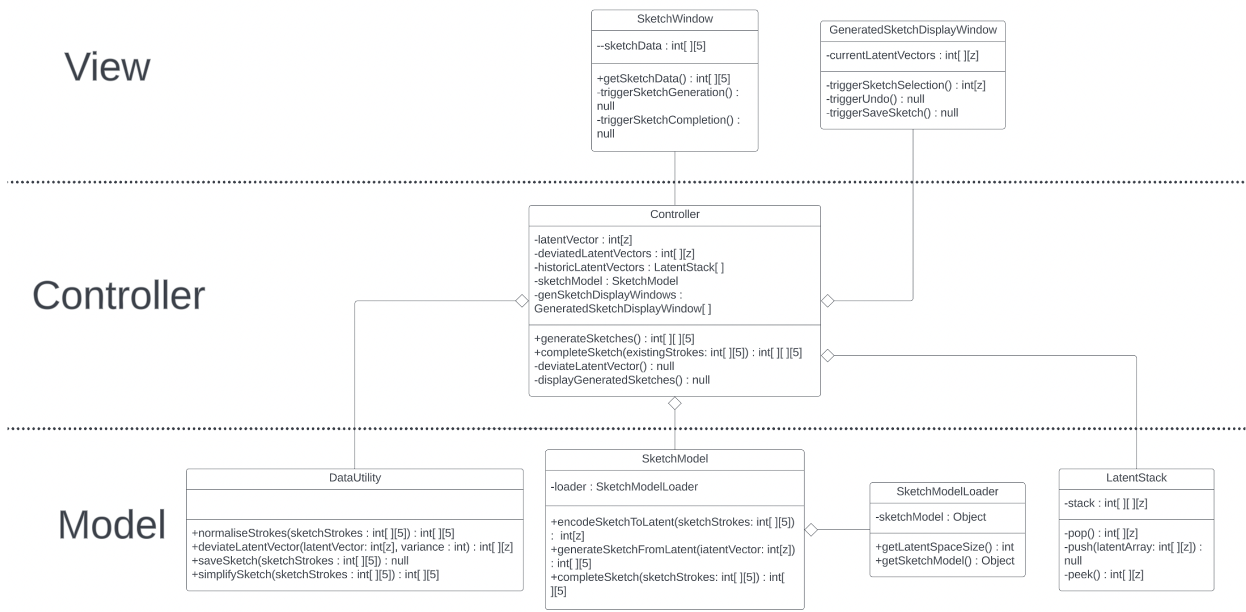


Figure 16: Sketch Application Design UML Class Diagram

Figure 17: The sketch application UML class diagram in the design

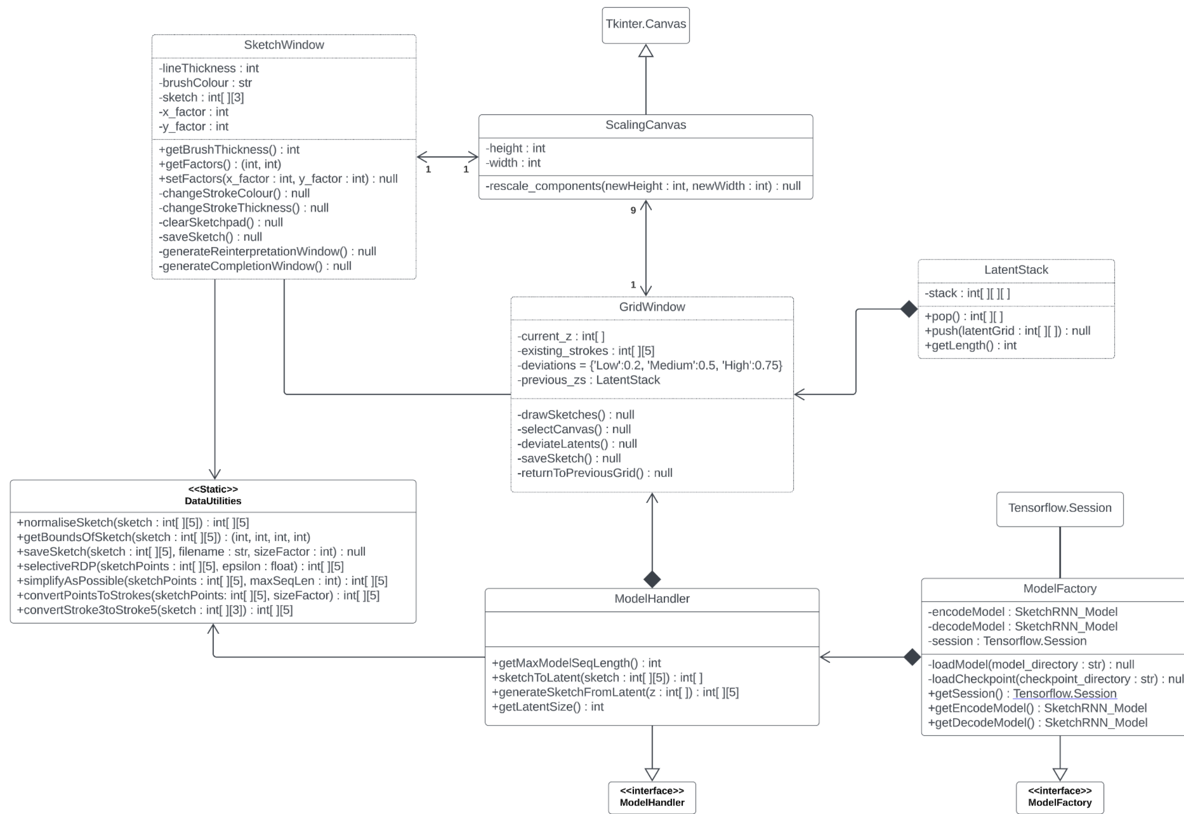


Figure 18: The sketch application UML class diagram in the implementation

8.3 Code Listing

This project is largely based on the existing Sketch-RNN Repository from which several files were used. Listed are the files which were either newly created for the project or contain meaningful modifications from the original file.

8.3.1 SketchTool.py

```
# NOTICE
# This code is derivative of the Sketch-RNN repository
# (https://github.com/magenta/magenta/tree/main/magenta/models/sketch_rnn)
# and the accompanying Jupyter notebook
# (https://github.com/magenta/magenta-demos/blob/main/jupyter-notebooks/Sketch_RNN.ipynb)
# and its used in compliance with the Apache Licence 2.0.

import tkinter as tk
from tkinter.colorchooser import askcolor
import tkmacosx as tkm
import tensorflow as tf
import numpy as np
import os
import json

from SketchToolUtilities import DataUtilities, Stack
from SketchToolInterfaces import ModelFactoryInterface, ModelHandlerInterface
import utils
import sketch_rnn_train
import model

EPSILON = 0.1
MODEL_DIR = '/tmp/sketch_rnn/models/owl/lstm'


class ModelFactory(ModelFactoryInterface):
    def __init__(self):
        sketch_rnn_train.download_pretrained_models()
        self.__encode_model, self.__decode_model = self.__load_model(MODEL_DIR)
        self.__session = tf.compat.v1.InteractiveSession()
        self.__session.run(tf.compat.v1.global_variables_initializer())
        self.__load_checkpoint(self.__session, MODEL_DIR)

    def getSession(self):
        return self.__session

    def getEncodeModel(self):
        return self.__encode_model
```

```
def getDecodeModel(self):
    return self.__decode_model

def __load_checkpoint(self, sess, checkpoint_path):
    saver = tf.compat.v1.train.Saver(tf.compat.v1.global_variables())
    ckpt = tf.compat.v1.train.get_checkpoint_state(checkpoint_path)
    saver.restore(sess, ckpt.model_checkpoint_path)

def __load_model(self, model_dir):
    model_params = model.get_default_hparams()
    path = os.path.join(model_dir, 'model_config.json')
    with tf.compat.v1.gfile.Open(path, 'r') as f:
        data = json.load(f)

    fix_list = [
        'conditional',
        'is_training',
        'use_input_dropout',
        'use_output_dropout',
        'use_recurrent_dropout'
    ]

    for fix in fix_list:
        data[fix] = (data[fix] == 1)

    model_params.parse_json(json.dumps(data))

    model_params.batch_size = 1
    encode_model_params = model.copy_hparams(model_params)

    # No dropout because we aren't training
    encode_model_params.use_input_dropout = 0
    encode_model_params.use_recurrent_dropout = 0
    encode_model_params.use_output_dropout = 0
    encode_model_params.is_training = 0

    decode_model_params = model.copy_hparams(encode_model_params)
    decode_model_params.max_seq_len = 1

    sketch_rnn_train.reset_graph()

    encode_model = model.Model(
        encode_model_params,
        reuse=tf.compat.v1.AUTO_REUSE
    )
```

```
        decode_model = model.Model(
            decode_model_params,
            reuse=tf.compat.v1.AUTO_REUSE
        )

        return encode_model, decode_model


class ModelHandler(ModelHandlerInterface):
    def __init__(self):
        self.__modelFactory = ModelFactory()

    def getMaxSeqLen(self):
        return self.__modelFactory.getEncodeModel().hps.max_seq_len

    def sketchToLatent(self, sketch):
        strokes = DataUtilities.stroke_3_to_stroke_5(
            sketch,
            max_len=self.__modelFactory.getEncodeModel().hps.max_seq_len
        )

        seq_len = [len(sketch)]

        feed = {
            self.__modelFactory.getEncodeModel().input_data: [strokes],
            self.__modelFactory.getEncodeModel().sequence_lengths: seq_len
        }

        z = self.__modelFactory.getSession().run(
            self.__modelFactory.getEncodeModel().batch_z,
            feed_dict=feed
        )

        return z

    def generateFromLatent(self, z, existing_strokes=None):
        strokes, _ = model.sample(
            self.__modelFactory.getSession(),
            self.__modelFactory.getDecodeModel(),
            seq_len=self.__modelFactory.getEncodeModel().hps.max_seq_len,
            temperature=0.1,
            z=z, greedy_mode=True, existing_strokes=existing_strokes)
        return strokes

    def getLatentSize(self):
```

```
        return self.__modelFactory.getDecodeModel().hps.z_size


class ScalingCanvas(tk.Canvas):
    def __init__(self, window, parent, parent_sketchable, **kwargs):
        super(ScalingCanvas, self).__init__(parent, **kwargs)
        self.bind('<Configure>', self.__rescale_components)
        self.__height = self.winfo_reqheight()
        self.__width = self.winfo_reqwidth()
        self.__window = window
        self.__parent_sketchable = parent_sketchable

    def __rescale_components(self, event):
        # Find factors by which the canvas was resized
        width_factor = float(event.width)/self.__width
        height_factor = float(event.height)/self.__height

        # We also need do rescale the stroke magnitudes.
        # E.g. a 5 cm stroke will span a whole small canvas,but a small amount
        # of a large one. But the stroke encoding should be the same.
        if self.__parent_sketchable:
            self.__window.adjustStrokeFactors((width_factor, height_factor))

        # Update width and height to new values
        self.__width = event.width
        self.__height = event.height

        # Re-scale all the components inside the canvas
        # args->(tags, offsetX, offsetY, xScale, yScale)
        self.scale('all', 0, 0, width_factor, height_factor)


class ExplorationWindow:
    def __init__(self, model_handler, z=None, existing_strokes=None):
        # Erroneous attributes
        # We expect either a z (if the window is for sketch reinterpretation)
        # or some existing strokes (for sketch completion)
        assert (
            (z is None and existing_strokes is not None) or
            (z is not None and existing_strokes is None)
        )

        # Main window properties
        self.__root = tk.Tk()
        self.__root.minsize(500, 500)
```

```
if z is None:
    # Completion
    self.__z = np.random.normal(
        size=(1, model_handler.getLatentSize())
    )
    self.__existing_strokes = existing_strokes
    self.__root.title('Sketch Completion Window')
else:
    # Reinterpretation
    self.__z = z
    self.__existing_strokes = None
    self.__root.title('Sketch Reinterpretation Window')

self.__deviations = {
    'Low': 0.2,
    'Medium': 0.5,
    'High': 0.75}
self.__model_handler = model_handler
self.__padding = 15
self.__previous_zs = Stack()

# Widget creation
self.__initialiseCanvases()
self.__saveButton = tk.Button(
    self.__root,
    text='Save Sketch',
    command=self.__saveFavourite
)
self.__saveButton.grid(row=3, column=0)
self.__varianceOptions = ['Low', 'Medium', 'High']
self.__variance = tk.StringVar(self.__root)
self.__variance.set('Low')
self.__varianceMenu = tk.OptionMenu(
    self.__root,
    self.__variance,
    *self.__varianceOptions
)
self.__varianceMenu.grid(row=3, column=1)
self.__returnButton = tk.Button(
    self.__root,
    text='Return to Previous Grid',
    command=self.__return
)
self.__returnButton.grid(row=3, column=2)

self.__root.columnconfigure([i for i in range(3)], weight=1)
```

```
        self.__root.rowconfigure([i for i in range(4)], weight=1)
        self.__root.rowconfigure(3, minsize=30)

        self.__deviateLatents()
        self.__updateSketches()

        self.__root.mainloop()

    def __initialiseCanvases(self):
        self.__canvases = [
            ScalingCanvas(
                self,
                self.__root,
                parent_sketchable=False,
                width=200,
                height=200,
                bg='white',
                highlightbackground='black'
            ) for i in range(9)]

        self.__canvases[4].config(highlightbackground='red')

        for i in range(3):
            for j in range(3):
                self.__canvases[i*3+j].grid(row=i, column=j, sticky='nesw')
                self.__canvases[i*3+j].bind(
                    '<Button-1>',
                    lambda event, i=i, j=j: self.__selectCanvas(i*3+j)
                )

    def __selectCanvas(self, position):
        self.__previous_zs.push(self.__zs)
        self.__z = [self.__zs[position]]
        self.__deviateLatents()
        self.__updateSketches()

    def __drawSketch(self, sketch, canvas_index):
        canvas = self.__canvases[canvas_index]
        canvas.delete('all')
        canvas_width = canvas.winfo_reqwidth()-self.__padding
        canvas_height = canvas.winfo_reqheight()-self.__padding
        min_x, max_x, min_y, max_y = DataUtilities.get_bounds(sketch, factor=1)
        sketch_width = max_x-min_x
        sketch_height = max_y-min_y
        w_factor = canvas_width/(sketch_width*2)
        h_factor = canvas_height/(sketch_height*2)
```

```
        cursor = (canvas_width/2, canvas_height/2)
        pen_up = 1
        for stroke in sketch:
            new_cursor = (
                cursor[0]+(stroke[0]*w_factor),
                cursor[1]+(stroke[1]*h_factor)
            )
            if not pen_up:
                canvas.create_line(
                    cursor[0],
                    cursor[1],
                    new_cursor[0],
                    new_cursor[1]
                )
            cursor = new_cursor
            pen_up = stroke[2]

    def __updateSketches(self):
        sketches = []
        for z in self.__zs:
            if self.__existing_strokes is not None:
                sketch = self.__model_handler.generateFromLatent(
                    [z],
                    np.array(self.__existing_strokes)
                )
            else:
                sketch = self.__model_handler.generateFromLatent([z])
            sketch = utils.to_normal_strokes(sketch)
            sketches.append(sketch)

        for i in range(9):
            self.__drawSketch(sketches[i], canvas_index=i)

    def __deviateLatents(self):
        z = self.__z[0]
        self.__zs = []
        for i in range(9):
            dev = np.random.normal(
                scale=self.__deviations[self.__variance.get()],
                size=z.shape
            )
            self.__zs.append(z + dev)
        self.__zs[4] = z

    def __saveFavourite(self):
        sketch = self.__model_handler.generateFromLatent([self.__z[0]])
```

```
        sketch = utils.to_normal_strokes(sketch)
        DataUtilities.strokes_to_svg(sketch)

    def __return(self):
        if self.__previous_zs.getLength() == 0:
            print('No space to go back!')
        else:
            self.__zs = self.__previous_zs.pop()
            self.__updateSketches()


class Sketch_Window:
    def __init__(self):
        # Erroneous attributes
        self.__modelHandler = ModelHandler()
        self.__lineThickness = 3
        self.__brushColour = 'black'
        self.__sketch = []
        self.__base_window_dims = (700, 600)
        self.__x_factor = 1
        self.__y_factor = 1
        self.__cursor = None

        # Main window properties
        self.__root = tk.Tk()
        self.__root.title('Sketch Pad')
        self.__root.geometry('700x600')
        self.__root.minsize(615, 350)

        # Canvas to draw on
        self.__sketchCanvas = ScalingCanvas(
            self,
            self.__root,
            parent_sketchable=True,
            width=50,
            height=50
        )
        self.__sketchCanvas.config(
            bg='white',
            highlightbackground='black'
        )
        self.__sketchCanvas.grid(
            row=0,
            column=0,
            columnspan=5,
            sticky='nesw'
```

```
)
self.__sketchCanvas.bind('<B1-Motion>', self.__pen_down)
self.__sketchCanvas.bind('<ButtonRelease-1>', self.__pen_up)

# Buttons to initiate sketch generation or completion
self.__generateButtonFrame = tk.Frame(
    self.__root,
    highlightbackground='black',
    highlightthickness=2
)
self.__generateButtonFrame.grid(row=1, column=0, sticky='nesw')
self.__generateButtonFrame.columnconfigure(0, weight=1)
self.__generateButtonFrame.rowconfigure((0, 1), weight=1)
self.__generateButton = tkm.Button(
    self.__generateButtonFrame,
    text='Generate Similar Sketches',
    command=self.__generateExplorationWindow,
)
self.__generateButton.bind(
    '<Enter>',
    lambda _: self.__generateButton.config(bg='gray58')
)
self.__generateButton.bind(
    '<Leave>',
    lambda _: self.__generateButton.config(bg='white')
)
self.__generateButton.grid(row=0, column=0)

self.__completeButton = tkm.Button(
    self.__generateButtonFrame,
    text='Generate Sketch Completions',
    command=self.__generateCompletionWindow
)
self.__completeButton.bind(
    '<Enter>',
    lambda _: self.__completeButton.config(bg='gray58')
)
self.__completeButton.bind(
    '<Leave>',
    lambda _: self.__completeButton.config(bg='white')
)
self.__completeButton.grid(row=1, column=0)

self.__saveAndClearButtonFrame = tk.Frame(
    self.__root,
    highlightbackground='black',
```

```
    highlightthickness=2
)
self.__saveAndClearButtonFrame.grid(
    row=1,
    column=4,
    columnspan=1,
    sticky='nesw'
)
self.__saveAndClearButtonFrame.columnconfigure(0, weight=1)
self.__saveAndClearButtonFrame.rowconfigure((0, 1), weight=1)
self.__clearButton = tkm.Button(
    self.__saveAndClearButtonFrame,
    text='Clear Sketchpad',
    command=self.__clearSketchpad
)
self.__clearButton.bind(
    '<Enter>',
    lambda _: self.__clearButton.config(bg='gray58')
)
self.__clearButton.bind(
    '<Leave>',
    lambda _: self.__clearButton.config(bg='white')
)
self.__clearButton.grid(row=0, column=0)

self.__saveButton = tkm.Button(
    self.__saveAndClearButtonFrame,
    text='Save Sketch',
    command=self.__saveSketch
)
self.__saveButton.bind(
    '<Enter>',
    lambda _: self.__saveButton.config(bg='gray58')
)
self.__saveButton.bind(
    '<Leave>',
    lambda _: self.__saveButton.config(bg='white')
)
self.__saveButton.grid(row=1, column=0)

# Stroke scale and colour inputs
self.__optionsFrame = tk.Frame(
    self.__root,
    highlightbackground='black',
    highlightthickness=2
)
```

```
self.__optionsFrame.grid(
    row=1,
    column=1,
    columnspan=3,
    sticky='nesw'
)
self.__sizeSlider = tk.Scale(
    self.__optionsFrame,
    from_=1,
    to=10,
    orient=tk.HORIZONTAL
)
self.__sizeSlider.bind(
    '<ButtonRelease-1>',
    self.__updateBrushThickness
)
self.__sizeSlider.grid(
    row=0,
    column=1,
    pady=(0, 10)
)
self.__sizeLabel = tk.Label(
    self.__optionsFrame,
    text='Stroke Size:'
)
self.__sizeLabel.grid(
    row=0,
    column=0,
    padx=(0, 10)
)
self.__changeColourButton = tk.Button(
    self.__optionsFrame,
    text='Change Brush Colour',
    command=self.__changeBrushColour
)
self.__changeColourButton.grid(
    row=1,
    column=0,
    columnspan=2,
    pady=(0, 10)
)
self.__optionsFrame.columnconfigure((0, 1), weight=1)
self.__optionsFrame.rowconfigure((0, 1), weight=1)

# Make sure grid cells scale properly with window resize
self.__root.columnconfigure([i for i in range(3)], weight=1)
```

```
        self.__root.rowconfigure(0, weight=1)

        self.__root.mainloop()

    def getBrushThickness(self):
        return self.__brushThickness

    def __changeBrushColour(self):
        colour = askcolor()
        self.__brushColour = colour[1]

    def __updateBrushThickness(self, event):
        self.__lineThickness = self.__sizeSlider.get()

    def getBaseDims(self):
        return self.__base_window_dims

    def getFactors(self):
        return self.__x_factor, self.__y_factor

    def __clearSketchpad(self):
        self.__sketch = []
        self.__sketchCanvas.delete('all')

    def __saveSketch(self):
        DataUtilities.strokes_to_svg(
            DataUtilities.stroke_3_to_stroke_5(
                self.__sketch,
                max_len=self.__modelHandler.getMaxSeqLen(),
            )
        )
        print('Sketch Saved!')

    def adjustStrokeFactors(self, factors):
        (x_factor, y_factor) = factors
        self.__x_factor *= x_factor
        self.__y_factor *= y_factor

    def __pen_down(self, event):
        if self.__cursor is not None:
            self.__paint_line(event)

        self.__cursor = (event.x, event.y)

        # Add movement to stroke list after scaling based on canvas size
        self.__sketch.append([
```

```
            event.x/self.__x_factor,
            event.y/self.__x_factor,
            0
        ])

    def __paint_line(self, event):
        self.__sketchCanvas.create_line(
            self.__cursor[0],
            self.__cursor[1],
            event.x,
            event.y,
            width=self.__lineThickness,
            fill=self.__brushColour
        )

    def __pen_up(self, event):
        if len(self.__sketch) > 0:
            self.__sketch[-1][-1] = 1
            self.__cursor = None

    def __generateExplorationWindow(self):
        simplified_sketch_points = DataUtilities.simplify_as_possible(
            self.__sketch,
            self.__modelHandler.getMaxSeqLen()
        )
        simplified_sketch_strokes = DataUtilities.convertPointsToStrokes(
            simplified_sketch_points,
            factor=10
        )

        sketch = DataUtilities.normalise(simplified_sketch_strokes)
        z = self.__modelHandler.sketchToLatent(sketch)
        ExplorationWindow(
            model_handler=self.__modelHandler,
            z=z
        )

    def __generateCompletionWindow(self):
        simplified_sketch_points = DataUtilities.simplify_as_possible(
            self.__sketch,
            self.__modelHandler.getMaxSeqLen()/2
        )
        simplified_sketch_strokes = DataUtilities.convertPointsToStrokes(
            simplified_sketch_points,
            factor=10
        )
```

```
        sketch = DataUtilities.normalise(simplified_sketch_strokes)
        sketch = DataUtilities.stroke_3_to_stroke_5(
            sketch,
            self.__modelHandler.getMaxSeqLen()
        )
        ExplorationWindow(
            model_handler=self.__modelHandler,
            existing_strokes=sketch
        )


if __name__ == '__main__':
    tf.compat.v1.disable_v2_behavior()
    sketch_window = Sketch_Window()
```

8.3.2 SketchToolInterfaces.py

```
# Defines pseudo-interfaces for use in SketchTool.py

class ModelFactoryInterface:
    ''' Class used by ModelHandler to acces models '''

    def getSession(self):
        ''' Returns the session used by the loaded model '''
        raise NotImplementedError()

    def getEncodeModel(self):
        ''' Returns the model used to encode sketches
            onto latent space. May be same model as
            decoder '''
        raise NotImplementedError()

    def getDecodeModel(self):
        ''' Returns the model used to decode latent vectors
            into sketches. May be same model as encoder '''
        raise NotImplementedError()


class ModelHandlerInterface:
    ''' Class used by GUI to abstract task of interacting
        with models '''

    def getMaxSeqLen(self):
        ''' Returns maximum number of strokes expected in a sketch '''
        raise NotImplementedError()
```

```
    def sketchToLatent(self, sketch):
        ''' Encode sketch into latent vector '''
        raise NotImplementedError()

    def generateFromLatent(self, z, existing_strokes):
        ''' Generate a sketch given a latent space. If existing strokes
            given, complete sketch. Otherwise generate from scratch. '''
        raise NotImplementedError()

    def getLatentSize(self):
        ''' Return size of latent vector expected by model '''
        raise NotImplementedError()
```

8.3.3 SketchToolUtilities.py

```
# NOTICE
# This code is derivative of and directly
# uses content belonging to the Sketch-RNN repository
# (https://github.com/magenta/magenta/tree/main/magenta/models/sketch_rnn))

# Provides utility methods for use in SketchTool.py and TransferLearning.py

import tensorflow as tf
import numpy as np

import svgwrite
import os
import rdp
import utils
import random


class Stack:
    def __init__(self):
        self.__stack = []

    def pop(self):
        if len(self.__stack) > 0:
            return self.__stack.pop()
        return None

    def getLength(self):
        return len(self.__stack)

    def push(self, element):
```

```
        self.__stack.append(element)


class DataUtilities:
    @staticmethod
    def normalise(sketch):
        # Normalises data as in section 1 of sketch-rnn paper appendix

        # First find mean
        meanx = 0
        meany = 0
        for stroke in sketch:
            meanx += stroke[0]
            meany += stroke[1]
        meanx /= len(sketch)
        meany /= len(sketch)

        # Then find variance
        varx = 0
        vary = 0
        for stroke in sketch:
            varx += (stroke[0]-meanx)**2
            vary += (stroke[1]-meany)**2

        varx /= (len(sketch)+1)
        vary /= (len(sketch)+1)

        # Then get standard deviations
        sdx = varx**(1/2)
        sdy = vary**(1/2)

        # Then normalise
        new_sketch = []
        for stroke in sketch:
            new_sketch.append([stroke[0]/sdx, stroke[1]/sdy, stroke[2]])

        return new_sketch

    @staticmethod
    def get_bounds(data, factor=10):
        """Return bounds of data."""
        min_x = 0
        max_x = 0
        min_y = 0
        max_y = 0
```

```
    abs_x = 0
    abs_y = 0
    for i in range(len(data)):
        x = float(data[i][0]) / factor
        y = float(data[i][1]) / factor
        abs_x += x
        abs_y += y
        min_x = min(min_x, abs_x)
        min_y = min(min_y, abs_y)
        max_x = max(max_x, abs_x)
        max_y = max(max_y, abs_y)

    return (min_x, max_x, min_y, max_y)

@staticmethod
def strokes_to_svg(sketch, filename='picture', size_coefficient=0.2):
    # Given a sketch in 3-stroke format, draw svg to filename. Adapted from
    # https://github.com/magenta/magenta-demos/blob/main/jupyter-
    # notebooks/Sketch_RNN.ipynb
    # in order to draw svg as series of paths for each stroke,
    # rather than a single large path such that one may include stroke
    # extensions such as colour more easily.

    tf.compat.v1.gfile.MakeDirs(os.path.dirname(filename))

    min_x, max_x, min_y, max_y = DataUtilities.get_bounds(
        sketch,
        size_coefficient
    )
    dims = (50 + max_x - min_x, 50 + max_y - min_y)
    dwg = svgwrite.Drawing(filename, size=dims)
    dwg.add(dwg.rect(insert=(0, 0), size=dims, fill='white'))
    x = 25 - min_x
    y = 25 - min_y
    lift_pen = 1

    paths = []
    for i in range(len(sketch)):
        x += float(sketch[i][0])/size_coefficient
        y += float(sketch[i][1])/size_coefficient

        end_of_stroke = sketch[i][2]

        if lift_pen:
            p = f'M{x},{y}'
        else:
```

```
            p += f'L{x},{y} '

        if end_of_stroke:
            paths.append(p)

        lift_pen = end_of_stroke

        i += 1

    for path in paths:
        dwg.add(dwg.path(path).stroke('black', 1).fill('none'))

    dwg.save()

@staticmethod
def selectiveRDP(sketch, epsilon):
    # Use rdp to simplify a drawing, but specifically protect sections
    # which correspond to 'pen up movements'.
    new_sketch = [sketch[0]]
    last_index = 1
    for i, stroke in enumerate(sketch[1:]):
        if stroke[2]:
            simplified_section = rdp.rdp(
                sketch[last_index:i+1][:-1],
                epsilon=epsilon
            )
            for simplified_stroke in simplified_section:
                new_sketch.append(simplified_stroke)
            new_sketch[-1][-1] = 1
            last_index = i+2
    simplified_section = rdp.rdp(
        sketch[last_index:][:-1],
        epsilon=epsilon
    )
    for simplified_stroke in simplified_section:
        new_sketch.append(simplified_stroke)
    return new_sketch

@staticmethod
def simplify_as_possible(sketch, max_seq_len):
    # Simplify a sketch using RDP algorithm until it
    # is shorter than a particular length
    if len(sketch) < max_seq_len:
        return sketch

    epsilon = 0.1
```

```
        while True:
            simplified_sketch = DataUtilities.selectiveRDP(sketch, epsilon)
            if len(simplified_sketch) < max_seq_len:
                return simplified_sketch
            epsilon += 0.1

    @staticmethod
    def convertPointsToStrokes(points, factor):
        # Converts a list of points to stroke-3 format
        strokes = [points[0]]
        for i in range(1, len(points)):
            currentPoint, lastPoint = points[i], points[i-1]
            movementx = currentPoint[0]-lastPoint[0]
            movementy = currentPoint[1]-lastPoint[1]
            pen_state = currentPoint[-1]
            strokes.append([movementx*factor, movementy*factor, pen_state])
        return strokes

    @staticmethod
    def stroke_3_to_stroke_5(sketch, max_len):
        # Convert sketch in stroke-3 format to stroke-5
        new_sketch = [[0, 0, 1, 0, 0]]
        for stroke in sketch:
            new_stroke = [
                stroke[0], stroke[1], not stroke[2], stroke[2], 0
            ]
            new_sketch.append(new_stroke)
        while len(new_sketch) <= max_len:
            new_sketch.append([0, 0, 0, 0, 1])
        return new_sketch


class DataTweaker(utils.DataLoader):
    def setScale(self, scale):
        self.__scale = scale

    def tweak(self):
        self.__transform()

        norm_factor = self.calculate_normalizing_scale_factor()
        self.normalize(norm_factor)

    def __transform(self):
        # Make transformations. Only scale if desired
        if self.__scale:
            self.__expand_horizon()
```

```
        self.__rotate_c()

    def __rotate_c(self):
        # Clockwise rotation
        for sketch in self.strokes:
            # random theta between 45, 135 degrees
            theta = (random.random()*(np.pi/2))+np.pi/4
            self.__rotate_individual(sketch, theta)

    def __rotate_individual(self, sketch, theta):
        rotation_matrix = np.array([
            [np.cos(theta), -np.sin(theta)],
            [np.sin(theta), np.cos(theta)]
        ])
        for stroke in sketch:
            stroke[:2] = np.dot(rotation_matrix, stroke[:2].T).T

    def __expand_horizon(self):
        for sketch in self.strokes:
            factor = (random.random()*3)+1
            self.__expand_individual_horizon(sketch, factor)

    def __expand_individual_horizon(self, sketch, factor):
        for stroke in sketch:
            stroke[0] *= factor
```

8.3.4 model.py

```
# Copyright 2022 The Magenta Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# NOTICE
# This file contains a modified version of code originally from
# the Sketch-RNN repository (https://github.com/magenta/magenta/tree/main/magenta/models/ske
# The code is identical excluding lines 389, 416-447.
```

```
"""Sketch-RNN Model."""

import random

import rnn
import numpy as np
import tensorflow.compat.v1 as tf
import magenta_training as contrib_training


def copy_hparams(hparams):
  """Return a copy of an HParams instance."""
  return contrib_training.HParams(**hparams.values())


def get_default_hparams():
  """Return default HParams for sketch-rnn."""
  hparams = contrib_training.HParams(
      data_set=['aaron_sheep.npz'],  # Our dataset.
      num_steps=10000000,  # Total number of steps of training. Keep large.
      save_every=500,  # Number of batches per checkpoint creation.
      max_seq_len=250,  # Not used. Will be changed by model. [Eliminate?]
      dec_rnn_size=512,  # Size of decoder.
      dec_model='lstm',  # Decoder: lstm, layer_norm or hyper.
      enc_rnn_size=256,  # Size of encoder.
      enc_model='lstm',  # Encoder: lstm, layer_norm or hyper.
      z_size=128,  # Size of latent vector z. Recommend 32, 64 or 128.
      kl_weight=0.5,  # KL weight of loss equation. Recommend 0.5 or 1.0.
      kl_weight_start=0.01,  # KL start weight when annealing.
      kl_tolerance=0.2,  # Level of KL loss at which to stop optimizing for KL.
      batch_size=100,  # Minibatch size. Recommend leaving at 100.
      grad_clip=1.0,  # Gradient clipping. Recommend leaving at 1.0.
      num_mixture=20,  # Number of mixtures in Gaussian mixture model.
      learning_rate=0.001,  # Learning rate.
      decay_rate=0.9999,  # Learning rate decay per minibatch.
      kl_decay_rate=0.99995,  # KL annealing decay rate per minibatch.
      min_learning_rate=0.00001,  # Minimum learning rate.
      use_recurrent_dropout=True,  # Dropout with memory loss. Recommended
      recurrent_dropout_prob=0.90,  # Probability of recurrent dropout keep.
      use_input_dropout=False,  # Input dropout. Recommend leaving False.
      input_dropout_prob=0.90,  # Probability of input dropout keep.
      use_output_dropout=False,  # Output dropout. Recommend leaving False.
      output_dropout_prob=0.90,  # Probability of output dropout keep.
      random_scale_factor=0.15,  # Random scaling data augmentation proportion.
      augment_stroke_prob=0.10,  # Point dropping augmentation proportion.
```

```
    conditional=True,  # When False, use unconditional decoder-only model.
    is_training=True  # Is model training? Recommend keeping true.
  )
  return hparams


class Model(object):
  """Define a SketchRNN model."""

  def __init__(self, hps, gpu_mode=True, reuse=False):
    """Initializer for the SketchRNN model.

    Args:
       hps: a HParams object containing model hyperparameters
       gpu_mode: a boolean that when True, uses GPU mode.
       reuse: a boolean that when true, attemps to reuse variables.
    """
    self.hps = hps
    with tf.variable_scope('vector_rnn', reuse=reuse):
      if not gpu_mode:
        with tf.device('/cpu:0'):
          tf.logging.info('Model using cpu.')
          self.build_model(hps)
      else:
        tf.logging.info('Model using gpu.')
        self.build_model(hps)

  def encoder(self, batch, sequence_lengths):
    """Define the bi-directional encoder module of sketch-rnn."""
    unused_outputs, last_states = tf.nn.bidirectional_dynamic_rnn(
        self.enc_cell_fw,
        self.enc_cell_bw,
        batch,
        sequence_length=sequence_lengths,
        time_major=False,
        swap_memory=True,
        dtype=tf.float32,
        scope='ENC_RNN')

    last_state_fw, last_state_bw = last_states
    last_h_fw = self.enc_cell_fw.get_output(last_state_fw)
    last_h_bw = self.enc_cell_bw.get_output(last_state_bw)
    last_h = tf.concat([last_h_fw, last_h_bw], 1)
    mu = rnn.super_linear(
        last_h,
        self.hps.z_size,
```

```
      input_size=self.hps.enc_rnn_size * 2,  # bi-dir, so x2
      scope='ENC_RNN_mu',
      init_w='gaussian',
      weight_start=0.001)
  presig = rnn.super_linear(
      last_h,
      self.hps.z_size,
      input_size=self.hps.enc_rnn_size * 2,  # bi-dir, so x2
      scope='ENC_RNN_sigma',
      init_w='gaussian',
      weight_start=0.001)
  return mu, presig

def build_model(self, hps):
  """Define model architecture."""
  if hps.is_training:
    self.global_step = tf.Variable(0, name='global_step', trainable=False)

  if hps.dec_model == 'lstm':
    cell_fn = rnn.LSTMCell
  elif hps.dec_model == 'layer_norm':
    cell_fn = rnn.LayerNormLSTMCell
  elif hps.dec_model == 'hyper':
    cell_fn = rnn.HyperLSTMCell
  else:
    assert False, 'please choose a respectable cell'

  if hps.enc_model == 'lstm':
    enc_cell_fn = rnn.LSTMCell
  elif hps.enc_model == 'layer_norm':
    enc_cell_fn = rnn.LayerNormLSTMCell
  elif hps.enc_model == 'hyper':
    enc_cell_fn = rnn.HyperLSTMCell
  else:
    assert False, 'please choose a respectable cell'

  use_recurrent_dropout = self.hps.use_recurrent_dropout
  use_input_dropout = self.hps.use_input_dropout
  use_output_dropout = self.hps.use_output_dropout

  cell = cell_fn(
      hps.dec_rnn_size,
      use_recurrent_dropout=use_recurrent_dropout,
      dropout_keep_prob=self.hps.recurrent_dropout_prob)

  if hps.conditional:  # vae mode:
```

```
  if hps.enc_model == 'hyper':
    self.enc_cell_fw = enc_cell_fn(
        hps.enc_rnn_size,
        use_recurrent_dropout=use_recurrent_dropout,
        dropout_keep_prob=self.hps.recurrent_dropout_prob)
    self.enc_cell_bw = enc_cell_fn(
        hps.enc_rnn_size,
        use_recurrent_dropout=use_recurrent_dropout,
        dropout_keep_prob=self.hps.recurrent_dropout_prob)
  else:
    self.enc_cell_fw = enc_cell_fn(
        hps.enc_rnn_size,
        use_recurrent_dropout=use_recurrent_dropout,
        dropout_keep_prob=self.hps.recurrent_dropout_prob)
    self.enc_cell_bw = enc_cell_fn(
        hps.enc_rnn_size,
        use_recurrent_dropout=use_recurrent_dropout,
        dropout_keep_prob=self.hps.recurrent_dropout_prob)

# dropout:
tf.logging.info('Input dropout mode = %s.', use_input_dropout)
tf.logging.info('Output dropout mode = %s.', use_output_dropout)
tf.logging.info('Recurrent dropout mode = %s.', use_recurrent_dropout)
if use_input_dropout:
  tf.logging.info('Dropout to input w/ keep_prob = %4.4f.',
                  self.hps.input_dropout_prob)
  cell = tf.nn.rnn_cell.DropoutWrapper(
      cell, input_keep_prob=self.hps.input_dropout_prob)
if use_output_dropout:
  tf.logging.info('Dropout to output w/ keep_prob = %4.4f.',
                  self.hps.output_dropout_prob)
  cell = tf.nn.rnn_cell.DropoutWrapper(
      cell, output_keep_prob=self.hps.output_dropout_prob)
self.cell = cell

self.sequence_lengths = tf.placeholder(
    dtype=tf.int32, shape=[self.hps.batch_size])
self.input_data = tf.placeholder(
    dtype=tf.float32,
    shape=[self.hps.batch_size, self.hps.max_seq_len + 1, 5])

# The target/expected vectors of strokes
self.output_x = self.input_data[:, 1:self.hps.max_seq_len + 1, :]
# vectors of strokes to be fed to decoder (same as above, but lagged behind
# one step to include initial dummy value of (0, 0, 1, 0, 0))
self.input_x = self.input_data[:, :self.hps.max_seq_len, :]
```

```
# either do vae-bit and get z, or do unconditional, decoder-only
if hps.conditional:  # vae mode:
  self.mean, self.presig = self.encoder(self.output_x,
                                        self.sequence_lengths)
  self.sigma = tf.exp(self.presig / 2.0)  # sigma > 0. div 2.0 -> sqrt.
  eps = tf.random_normal(
      (self.hps.batch_size, self.hps.z_size), 0.0, 1.0, dtype=tf.float32)
  self.batch_z = self.mean + tf.multiply(self.sigma, eps)
  # KL cost
  self.kl_cost = -0.5 * tf.reduce_mean(
      (1 + self.presig - tf.square(self.mean) - tf.exp(self.presig)))
  self.kl_cost = tf.maximum(self.kl_cost, self.hps.kl_tolerance)
  pre_tile_y = tf.reshape(self.batch_z,
                          [self.hps.batch_size, 1, self.hps.z_size])
  overlay_x = tf.tile(pre_tile_y, [1, self.hps.max_seq_len, 1])
  actual_input_x = tf.concat([self.input_x, overlay_x], 2)
  self.initial_state = tf.nn.tanh(
      rnn.super_linear(
          self.batch_z,
          cell.state_size,
          init_w='gaussian',
          weight_start=0.001,
          input_size=self.hps.z_size))
else:  # unconditional, decoder-only generation
  self.batch_z = tf.zeros(
      (self.hps.batch_size, self.hps.z_size), dtype=tf.float32)
  self.kl_cost = tf.zeros([], dtype=tf.float32)
  actual_input_x = self.input_x
  self.initial_state = cell.zero_state(
      batch_size=hps.batch_size, dtype=tf.float32)

self.num_mixture = hps.num_mixture

# TODO(deck): Better understand this comment.
# Number of outputs is 3 (one logit per pen state) plus 6 per mixture
# component: mean_x, stdev_x, mean_y, stdev_y, correlation_xy, and the
# mixture weight/probability (Pi_k)
n_out = (3 + self.num_mixture * 6)

with tf.variable_scope('RNN'):
  output_w = tf.get_variable('output_w', [self.hps.dec_rnn_size, n_out])
  output_b = tf.get_variable('output_b', [n_out])

# decoder module of sketch-rnn is below
output, last_state = tf.nn.dynamic_rnn(
```

```
        cell,
        actual_input_x,
        initial_state=self.initial_state,
        time_major=False,
        swap_memory=True,
        dtype=tf.float32,
        scope='RNN')

    output = tf.reshape(output, [-1, hps.dec_rnn_size])
    output = tf.nn.xw_plus_b(output, output_w, output_b)
    self.final_state = last_state

    # NB: the below are inner functions, not methods of Model
    def tf_2d_normal(x1, x2, mu1, mu2, s1, s2, rho):
      """Returns result of eq # 24 of http://arxiv.org/abs/1308.0850."""
      norm1 = tf.subtract(x1, mu1)
      norm2 = tf.subtract(x2, mu2)
      s1s2 = tf.multiply(s1, s2)
      # eq 25
      z = (tf.square(tf.div(norm1, s1)) + tf.square(tf.div(norm2, s2)) -
           2 * tf.div(tf.multiply(rho, tf.multiply(norm1, norm2)), s1s2))
      neg_rho = 1 - tf.square(rho)
      result = tf.exp(tf.div(-z, 2 * neg_rho))
      denom = 2 * np.pi * tf.multiply(s1s2, tf.sqrt(neg_rho))
      result = tf.div(result, denom)
      return result

    def get_lossfunc(z_pi, z_mu1, z_mu2, z_sigma1, z_sigma2, z_corr,
                     z_pen_logits, x1_data, x2_data, pen_data):
      """Returns a loss fn based on eq #26 of http://arxiv.org/abs/1308.0850."""
      # This represents the L_R only (i.e. does not include the KL loss term).

      result0 = tf_2d_normal(x1_data, x2_data, z_mu1, z_mu2, z_sigma1, z_sigma2,
                             z_corr)
      epsilon = 1e-6
      # result1 is the loss wrt pen offset (L_s in equation 9 of
      # https://arxiv.org/pdf/1704.03477.pdf)
      result1 = tf.multiply(result0, z_pi)
      result1 = tf.reduce_sum(result1, 1, keepdims=True)
      result1 = -tf.log(result1 + epsilon)  # avoid log(0)

      fs = 1.0 - pen_data[:, 2]  # use training data for this
      fs = tf.reshape(fs, [-1, 1])
      # Zero out loss terms beyond N_s, the last actual stroke
      result1 = tf.multiply(result1, fs)
```

```
  # result2: loss wrt pen state, (L_p in equation 9)
  result2 = tf.nn.softmax_cross_entropy_with_logits(
      labels=pen_data, logits=z_pen_logits)
  result2 = tf.reshape(result2, [-1, 1])
  if not self.hps.is_training:  # eval mode, mask eos columns
    result2 = tf.multiply(result2, fs)

  result = result1 + result2
  return result

# below is where we need to do MDN (Mixture Density Network) splitting of
# distribution params
def get_mixture_coef(output):
  """Returns the tf slices containing mdn dist params."""
  # This uses eqns 18 -> 23 of http://arxiv.org/abs/1308.0850.
  z = output
  z_pen_logits = z[:, 0:3]  # pen states
  z_pi, z_mu1, z_mu2, z_sigma1, z_sigma2, z_corr = tf.split(z[:, 3:], 6, 1)

  # process output z's into MDN parameters

  # softmax all the pi's and pen states:
  z_pi = tf.nn.softmax(z_pi)
  z_pen = tf.nn.softmax(z_pen_logits)

  # exponentiate the sigmas and also make corr between -1 and 1.
  z_sigma1 = tf.exp(z_sigma1)
  z_sigma2 = tf.exp(z_sigma2)
  z_corr = tf.tanh(z_corr)

  r = [z_pi, z_mu1, z_mu2, z_sigma1, z_sigma2, z_corr, z_pen, z_pen_logits]
  return r

out = get_mixture_coef(output)
[o_pi, o_mu1, o_mu2, o_sigma1, o_sigma2, o_corr, o_pen, o_pen_logits] = out

self.pi = o_pi
self.mu1 = o_mu1
self.mu2 = o_mu2
self.sigma1 = o_sigma1
self.sigma2 = o_sigma2
self.corr = o_corr
self.pen_logits = o_pen_logits
# pen state probabilities (result of applying softmax to self.pen_logits)
self.pen = o_pen
```

```
    # reshape target data so that it is compatible with prediction shape
    target = tf.reshape(self.output_x, [-1, 5])
    [x1_data, x2_data, eos_data, eoc_data, cont_data] = tf.split(target, 5, 1)
    pen_data = tf.concat([eos_data, eoc_data, cont_data], 1)

    lossfunc = get_lossfunc(o_pi, o_mu1, o_mu2, o_sigma1, o_sigma2, o_corr,
                            o_pen_logits, x1_data, x2_data, pen_data)

    self.r_cost = tf.reduce_mean(lossfunc)

    if self.hps.is_training:
      self.lr = tf.Variable(self.hps.learning_rate, trainable=False)
      optimizer = tf.train.AdamOptimizer(self.lr)

      self.kl_weight = tf.Variable(self.hps.kl_weight_start, trainable=False)
      self.cost = self.r_cost + self.kl_cost * self.kl_weight

      gvs = optimizer.compute_gradients(self.cost)
      g = self.hps.grad_clip
      capped_gvs = [(tf.clip_by_value(grad, -g, g), var) for grad, var in gvs]
      self.train_op = optimizer.apply_gradients(
          capped_gvs, global_step=self.global_step, name='train_step')


def sample(sess, model, seq_len=250, temperature=1.0, greedy_mode=False,
           z=None, existing_strokes=None):
  """Samples a sequence from a pre-trained model."""

  def adjust_temp(pi_pdf, temp):
    pi_pdf = np.log(pi_pdf) / temp
    pi_pdf -= pi_pdf.max()
    pi_pdf = np.exp(pi_pdf)
    pi_pdf /= pi_pdf.sum()
    return pi_pdf

  def get_pi_idx(x, pdf, temp=1.0, greedy=False):
    """Samples from a pdf, optionally greedily."""
    if greedy:
      return np.argmax(pdf)
    pdf = adjust_temp(np.copy(pdf), temp)
    accumulate = 0
    for i in range(0, pdf.size):
      accumulate += pdf[i]
      if accumulate >= x:
        return i
    tf.logging.info('Error with sampling ensemble.')
```

```
  return -1

def sample_gaussian_2d(mu1, mu2, s1, s2, rho, temp=1.0, greedy=False):
  if greedy:
    return mu1, mu2
  mean = [mu1, mu2]
  s1 *= temp * temp
  s2 *= temp * temp
  cov = [[s1 * s1, rho * s1 * s2], [rho * s1 * s2, s2 * s2]]
  x = np.random.multivariate_normal(mean, cov, 1)
  return x[0][0], x[0][1]

prev_x = np.zeros((1, 1, 5), dtype=np.float32)
prev_x[0, 0, 2] = 1  # initially, we want to see beginning of new stroke

if z is None:
  z = np.random.randn(1, model.hps.z_size)  # not used if unconditional

if not model.hps.conditional:
  prev_state = sess.run(model.initial_state)
else:
  prev_state = sess.run(model.initial_state, feed_dict={model.batch_z: z})

strokes = np.zeros((seq_len, 5), dtype=np.float32)
mixture_params = []

greedy = greedy_mode
temp = temperature

if existing_strokes is not None:
  i = 0
  while i < existing_strokes.shape[0]:
    if np.array_equiv(existing_strokes[i], np.array([0, 0, 0, 0, 1])):
      break
    strokes[i] = existing_strokes[i]
    prev_x = np.expand_dims(existing_strokes[i:i+1], axis=0)
    if not model.hps.conditional:
      feed = {
        model.input_x: prev_x,
        model.sequence_lengths: [1],
        model.initial_state: prev_state
      }
    else:
      feed = {
        model.input_x: prev_x,
        model.sequence_lengths: [1],
```

```
        model.initial_state: prev_state,
        model.batch_z: z
      }

    params = sess.run([
      model.pi, model.mu1, model.mu2, model.sigma1, model.sigma2, model.corr,
      model.pen, model.final_state
    ], feed)

    prev_state = params[-1]
    i += 1
  start = i
  prev_x = np.zeros((1, 1, 5), dtype=np.float32)
else:
  start = 0

for i in range(start, seq_len):
  if not model.hps.conditional:
    feed = {
        model.input_x: prev_x,
        model.sequence_lengths: [1],
        model.initial_state: prev_state
    }
  else:
    feed = {
        model.input_x: prev_x,
        model.sequence_lengths: [1],
        model.initial_state: prev_state,
        model.batch_z: z
    }

  params = sess.run([
      model.pi, model.mu1, model.mu2, model.sigma1, model.sigma2, model.corr,
      model.pen, model.final_state
  ], feed)

  [o_pi, o_mu1, o_mu2, o_sigma1, o_sigma2, o_corr, o_pen, next_state] = params

  idx = get_pi_idx(random.random(), o_pi[0], temp, greedy)

  idx_eos = get_pi_idx(random.random(), o_pen[0], temp, greedy)
  eos = [0, 0, 0]
  eos[idx_eos] = 1

  next_x1, next_x2 = sample_gaussian_2d(o_mu1[0][idx], o_mu2[0][idx],
                                        o_sigma1[0][idx], o_sigma2[0][idx],
```

```
                                                o_corr[0][idx], np.sqrt(temp), greedy)

    strokes[i, :] = [next_x1, next_x2, eos[0], eos[1], eos[2]]

    params = [
        o_pi[0], o_mu1[0], o_mu2[0], o_sigma1[0], o_sigma2[0], o_corr[0],
        o_pen[0]
    ]

    mixture_params.append(params)

    prev_x = np.zeros((1, 1, 5), dtype=np.float32)
    prev_x[0][0] = np.array(
        [next_x1, next_x2, eos[0], eos[1], eos[2]], dtype=np.float32)
    prev_state = next_state

  return strokes, mixture_params
```

8.3.5 TransferLearning.py

```
# NOTICE
# This code is a modified version of code originally from
# the Sketch-RNN repository (https://github.com/magenta/magenta/tree/main/magenta/models/ske

from io import BytesIO

import tensorflow as tf
import numpy as np
import os
import json
import time
import requests
import sys
import svgwrite

import sketch_rnn_train
import model as sketch_rnn_model
import utils
from SketchToolUtilities import DataTweaker, DataUtilities

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.INFO)

FLAGS = tf.compat.v1.app.flags.FLAGS

PRETRAINED_MODELS_URL = ('http://download.magenta.tensorflow.org/models/'
                         'sketch_rnn.zip')
```

```
models_root_dir = '/tmp/sketch_rnn/models'
model_dir = '/tmp/sketch_rnn/models/aaron_sheep/lstm'

def load_environment(data_dir, model_dir, scale=False):
    """Loads environment for inference mode, used in jupyter notebook."""
    # modified https://github.com/tensorflow/magenta/blob/master/magenta/models/sketch_rnn/s
    # to work with depreciated tf.HParams functionality
    model_params = sketch_rnn_model.get_default_hparams()
    with tf.compat.v1.gfile.Open(
        os.path.join(model_dir, 'model_config.json'),
        'r'
    ) as f:
        data = json.load(f)
    fix_list = [
        'conditional',
        'is_training',
        'use_input_dropout',
        'use_output_dropout',
        'use_recurrent_dropout'
    ]
    for fix in fix_list:
        data[fix] = (data[fix] == 1)
    model_params.parse_json(json.dumps(data))
    return load_dataset(data_dir, model_params)


def load_dataset(data_dir, model_params):
    """Loads the .npz file, and splits the set into train/valid/test."""

    # normalizes the x and y columns using the training set.
    # applies same scaling factor to valid and test set.

    if isinstance(model_params.data_set, list):
        datasets = model_params.data_set
    else:
        datasets = [model_params.data_set]

    train_strokes = None
    valid_strokes = None
    test_strokes = None

    for dataset in datasets:
        if data_dir.startswith('http://') or data_dir.startswith('https://'):
            data_filepath = '/'.join([data_dir, dataset])
```

```
            tf.compat.v1.logging.info('Downloading %s', data_filepath)
            response = requests.get(data_filepath, allow_redirects=True)
            data = np.load(
                BytesIO(response.content),
                allow_pickle=True,
                encoding='latin1'
            )
        else:
            data_filepath = os.path.join(data_dir, dataset)
            data = np.load(data_filepath, encoding='latin1', allow_pickle=True)
        tf.compat.v1.logging.info('Loaded {}/{}/{} from {}'.format(
            len(data['train']), len(data['valid']), len(data['test']),
            dataset))
        if train_strokes is None:
            train_strokes = data['train']
            valid_strokes = data['valid']
            test_strokes = data['test']
        else:
            train_strokes = np.concatenate((train_strokes, data['train']))
            valid_strokes = np.concatenate((valid_strokes, data['valid']))
            test_strokes = np.concatenate((test_strokes, data['test']))

    # calculate the max strokes we need.
    all_strokes = np.concatenate((train_strokes, valid_strokes, test_strokes))
    max_seq_len = utils.get_max_len(all_strokes)
    # overwrite the hps with this calculation.
    model_params.max_seq_len = max_seq_len

    sample_model_params = sketch_rnn_model.copy_hparams(model_params)
    sample_model_params.use_input_dropout = 0
    sample_model_params.use_recurrent_dropout = 0
    sample_model_params.use_output_dropout = 0
    sample_model_params.is_training = 0
    sample_model_params.batch_size = 1
    sample_model_params.max_seq_len = 1

    train_set = DataTweaker(
        train_strokes,
        model_params.batch_size,
        max_seq_length=model_params.max_seq_len,
        random_scale_factor=model_params.random_scale_factor,
        augment_stroke_prob=model_params.augment_stroke_prob)
    normalizing_scale_factor = train_set.calculate_normalizing_scale_factor()
    train_set.normalize(normalizing_scale_factor)
    train_set.setScale(scale)
    train_set.tweak()
```

```
    valid_set = DataTweaker(
        valid_strokes,
        model_params.batch_size,
        max_seq_length=model_params.max_seq_len,
        random_scale_factor=0.0,
        augment_stroke_prob=0.0)
    valid_set.normalize(normalizing_scale_factor)
    valid_set.setScale(scale)
    valid_set.tweak()

    result = [
      train_set,
      valid_set,
      model_params,
      sample_model_params
    ]
    return result


def train(sess, model, sample_model, train_set, valid_set):
    """Train a sketch-rnn model."""
    # Setup summary writer.
    summary_writer = tf.compat.v1.summary.FileWriter(FLAGS.log_root)

    # Calculate trainable params.
    t_vars = tf.compat.v1.trainable_variables()
    count_t_vars = 0
    for var in t_vars:
        num_param = np.prod(var.get_shape().as_list())
        count_t_vars += num_param
    tf.compat.v1.logging.info('Total trainable variables %i.', count_t_vars)
    model_summ = tf.compat.v1.summary.Summary()
    model_summ.value.add(
        tag='Num_Trainable_Params', simple_value=float(count_t_vars))
    summary_writer.add_summary(model_summ, 0)
    summary_writer.flush()

    # main train loop

    hps = model.hps

    start = time.time()

    for step in range(hps.num_steps):
```

```
curr_learning_rate = (
    (hps.learning_rate - hps.min_learning_rate) *
    (hps.decay_rate)**step +
    hps.min_learning_rate
)
curr_kl_weight = (
    hps.kl_weight -
    (hps.kl_weight - hps.kl_weight_start) *
    (hps.kl_decay_rate)**step
)

_, x, s = train_set.random_batch()
feed = {
    model.input_data: x,
    model.sequence_lengths: s,
    model.lr: curr_learning_rate,
    model.kl_weight: curr_kl_weight,
}

(train_cost, r_cost, kl_cost, _, train_step, _) = sess.run([
    model.cost, model.r_cost, model.kl_cost, model.final_state,
    model.global_step, model.train_op
], feed)

if step % 20 == 0 and step > 0:

    end = time.time()
    time_taken = end - start

    cost_summ = tf.compat.v1.summary.Summary()
    cost_summ.value.add(
        tag='Train_Cost',
        simple_value=float(train_cost)
    )
    reconstr_summ = tf.compat.v1.summary.Summary()
    reconstr_summ.value.add(
        tag='Train_Reconstr_Cost', simple_value=float(r_cost))
    kl_summ = tf.compat.v1.summary.Summary()
    kl_summ.value.add(tag='Train_KL_Cost', simple_value=float(kl_cost))
    lr_summ = tf.compat.v1.summary.Summary()
    lr_summ.value.add(
        tag='Learning_Rate', simple_value=float(curr_learning_rate))
    kl_weight_summ = tf.compat.v1.summary.Summary()
    kl_weight_summ.value.add(
        tag='KL_Weight', simple_value=float(curr_kl_weight))
    time_summ = tf.compat.v1.summary.Summary()
```

```
time_summ.value.add(
    tag='Time_Taken_Train', simple_value=float(time_taken))

output_format = ('step: %d, lr: %.6f, klw: %0.4f, cost: %.4f, '
                 'recon: %.4f, kl: %.4f, train_time_taken: %.4f')
output_values = (
    step,
    curr_learning_rate,
    curr_kl_weight,
    train_cost,
    r_cost,
    kl_cost,
    time_taken
)
output_log = output_format % output_values

tf.compat.v1.logging.info(output_log)

summary_writer.add_summary(cost_summ, train_step)
summary_writer.add_summary(reconstr_summ, train_step)
summary_writer.add_summary(kl_summ, train_step)
summary_writer.add_summary(lr_summ, train_step)
summary_writer.add_summary(kl_weight_summ, train_step)
summary_writer.add_summary(time_summ, train_step)
summary_writer.flush()
start = time.time()

if not step % 100 and step > 0:
    print('*** Validation ***')
    (
        valid_cost,
        valid_r_cost,
        valid_kl_cost
    ) = sketch_rnn_train.evaluate_model(
        sess,
        model,
        valid_set
    )

    end = time.time()
    time_taken_valid = end - start

    sketch = sketch_rnn_model.sample(sess, sample_model)[0]
    DataUtilities.strokes_to_svg(
        sketch,
        filename=str(step)+'.svg'
```

```
                )

                start = time.time()

                valid_cost_summ = tf.compat.v1.summary.Summary()
                valid_cost_summ.value.add(
                    tag='Valid_Cost', simple_value=float(valid_cost))
                valid_reconstr_summ = tf.compat.v1.summary.Summary()
                valid_reconstr_summ.value.add(
                    tag='Valid_Reconstr_Cost',
                    simple_value=float(valid_r_cost)
                )
                valid_kl_summ = tf.compat.v1.summary.Summary()
                valid_kl_summ.value.add(
                    tag='Valid_KL_Cost', simple_value=float(valid_kl_cost))
                valid_time_summ = tf.compat.v1.summary.Summary()
                valid_time_summ.value.add(
                    tag='Time_Taken_Valid',
                    simple_value=float(time_taken_valid)
                )

                tf.compat.v1.logging.info(output_log)

                summary_writer.add_summary(valid_cost_summ, train_step)
                summary_writer.add_summary(valid_reconstr_summ, train_step)
                summary_writer.add_summary(valid_kl_summ, train_step)
                summary_writer.add_summary(valid_time_summ, train_step)
                summary_writer.flush()


if __name__ == '__main__':
    sketch_rnn_train.download_pretrained_models(models_root_dir)
    tf.compat.v1.disable_v2_behavior()

    # Dataset to be scaled before rotation?
    if 'scale' in sys.argv:
        print('scaling!')
        scale = True
    else:
        scale = False
        print('not scaling!')

    [
        train_set,
        valid_set,
        hps_model,
```

```
    sample_hps_model
] = load_environment(FLAGS.data_dir, model_dir, scale=scale)

sketch_rnn_train.reset_graph()

# Training new model or one from scratch?
if 'new' in sys.argv:
    print('new!')
    hps_model = sketch_rnn_model.get_default_hparams()
else:
    print('not new...')

model = sketch_rnn_model.Model(hps_model)
sample_model = sketch_rnn_model.Model(sample_hps_model, reuse=True)

sess = tf.compat.v1.InteractiveSession()
sess.run(tf.compat.v1.global_variables_initializer())

# Checkpoint must be loaded iff not new model, but after global
# variables are initialised
if 'new' not in sys.argv:
    sketch_rnn_train.load_checkpoint(sess, model_dir)

train(sess, model, sample_model, train_set, valid_set)
```

References

[1] D. Ha and D. Eck, “A neural representation of sketch drawings,” 2017.

[2] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, “Hierarchical text-conditional image generation with clip latents,” 2022.

[3] A. Graves, “Generating sequences with recurrent neural networks,” 2014.

[4] H. Kinsley, *Neural Networks from Scratch in Python*, 2020. [Online]. Available: https://nnfs.io

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” 2014.

[6] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders,” 2021.

[7] D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. [Online]. Available: https://doi.org/10.1561%2F2200000056

[8] A. Dhinakaran, “Understanding kl divergence,” 2023. [Online]. Available: https://towardsdatascience.com/understanding-kl-divergence-f3ddc8dff254

[9] S. Zhao, J. Song, and S. Ermon, “Towards deeper understanding of variational autoencoding models,” 2017.

[10] A. Sherstinsky, “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network,” *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, mar 2020. [Online]. Available: https://doi.org/10.1016%2Fj.physd.2019.132306

[11] R. C. Staudemeyer and E. R. Morris, “Understanding lstm – a tutorial into long short-term memory recurrent neural networks,” 2019.

[12] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, oct 2017. [Online]. Available: https://doi.org/10.1109%2Ftnnls.2016.2582924

[13] S. Alla, “A guide to bidirectional rnns with keras,” 2021. [Online]. Available: https://blog.paperspace.com/bidirectional-rnn-keras/

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

[15] S. Glen, “Bivariate normal distribution / multivariate normal (overview).” [Online]. Available: https://www.statisticshowto.com/bivariate-normal-distribution/

[16] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," 2020.

[17] G. C. Lab, "The quick, draw! dataset." [Online]. Available: https://github.com/googlecreativelab/quickdraw-dataset

[18] W. Chen and J. Hays, "Sketchygan: Towards diverse and realistic sketch to image synthesis," 2018.

[19] S.-Y. Wang, D. Bau, and J.-Y. Zhu, "Sketch your own gan," 2021.

[20] A. Sain, A. K. Bhunia, Y. Yang, T. Xiang, and Y.-Z. Song, "Stylemeup: Towards style-agnostic sketch-based image retrieval," 2021.

[21] D. Amos, "Python gui programming with tkinter." [Online]. Available: https://realpython.com/python-gui-tkinter/

[22] E. de Koning, "Polyline simplification," 2011. [Online]. Available: https://www.codeproject.com/Articles/114797/Polyline-Simplification#headingDP

[23] A. Koblin, "Aaron koblin sheep dataset," 2006. [Online]. Available: https://github.com/hardmaru/sketch-rnn-datasets/tree/master/aaron_sheep