University of Bergen
Department of informatics

Making a Drawing-Assistant System using Deep Learning

Author: Hans Martin Theigler Johansen
Supervisor: Troels Arnfred Bojesen

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2022

Abstract

Deep learning has been extensively employed for simplifying the process of generating text. Autocomplete is a ubiquitous tool on most mobile phones today, and the most widely used text processors all utilize autocomplete. Another medium where autocomplete might be helpful is sketching, which is great for quickly communicating ideas and not bound to any skill level. In this thesis, we seek to improve the free-hand sketching experience.

After discussing ways to enhance this domain, we create and experiment with a set of models based on a seminal paper in the deep learning sketch domain. Finally, we present a drawing application that utilizes the models we presented to create an autocompletion drawing experience.

Acknowledgements

First and foremost, I want to thank Troels for being a great supervisor and being patient with me throughout the work of this thesis. Secondly I'd like to thank my fellow master students Knut Holager, Mathias Madslien, Halvor Barndon, John Villanger, and Johanna Jøsang for creating a great environment to work in. Finally, thanks to Knut and Mathias for opening my eyes to the fantastic world of high-quality sandwiches.

`Hans Martin Theigler Johansen`
Wednesday 1st June, 2022

Contents

List of Figures

List of Acronyms and Abbreviations

AI	Artificial Intelligence.
ANN	Artificial Neural Network.
CNN	Convolutional Neural Network.
GCPS	Generation Conditioned on Prior Sequence.
GMM	Gaussian Mixture Model.
LSTM	Long Short-Term Memory.
MLP	Multilayer Perceptron.
RDP	Ramer-Douglas-Peucker.
ReLU	Rectified Linear Unit.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
SDG	Stochastic Gradient Descent.

Chapter 1

Introduction

In this chapter, we give an overview of the motivation for the project, followed by a review of the thesis structure. Finally, we discuss some ways that an artificial intelligence system could improve or speed up a drawing experience.

1.1 Motivation and Overview

Deep learning is a family of machine learning that employs neural networks. Like other machine learning models, neural networks are concerned with learning a task from a set of examples, which contrasts with how regular computer programs and algorithms work, where a human gives the instructions as code. Systems that can perform tasks that traditionally require humans to perform are called Artificial Intelligence (AI) systems. Deep learning has shown to be very effective and flexible in many domains. Consequently, modern AI systems are usually enabled through deep learning and neural networks.

Deep learning has, for example, been extensively applied to create tools that improve the experience of working with text. Word completion on phones has become ubiquitous. Language models such as GPT [30][31][3] can both create realistic text by themselves and be able to complete incomplete sentences. Translation models convert text from one language to another [32][45].

Free-hand sketching, which is in this thesis is also referred to as ”sketching” or ”drawing,” is a great tool both for expressing oneself artistically and for communicating ideas to other humans. These attributes make it, in some ways, similar to text. However unlike writing, sketching does not require knowing any language nor requires any practice to get started. Compared to writing, much less work has been done on augmenting this domain with deep learning.

Prior to 2017, most of the research on the topic was focused on making models that could classify sketches, e.g., recognize a drawing of a cat as a cat. In 2015, Yang and Hospedales [43] introduced the first model to beat humans on sketch classification. In 2017, Ha and Eck [17] with Google introduced a seminal paper where they released the largest sketching dataset to date and trained a generative model on the dataset. The model was capable of generating novel sketches in addition to completing unfinished sketches, which showcased the capabilities of deep learning in the domain. Several subsequent papers made improvements upon this architecture.

Inspired by these works, we seek to explore the possibilities of using deep learning to create an artificial drawing assistant that can interactively help a user draw. In the following section, we discuss broadly how such an assistant could work before deciding on the functionality we will work towards in this thesis.

1.2 Ways to improve the sketching experience

There are several ways an artificial drawing assistant system could aid a user when sketching. One way is to transform a finished sketch into one that is more visually appealing. For example, a circle drawn by a user will probably be somewhat irregular. A system could recognize that the user intended to draw a circle and replace it with a perfectly round one (see fig. 1.1).

Creating such a system introduces a few challenges. Firstly, recognizing what the user intended to draw is not a clear-cut challenge, which might make it a suitable task for deep learning. Secondly, a human artist must create the desired replacements for each object to support, which can be costly and time-consuming. It can also be problematic if the need

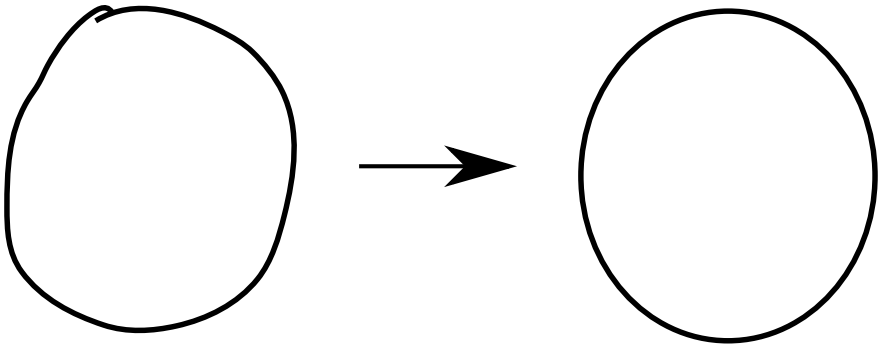

Figure 1.1: Transforming an imperfect oval into a perfect one

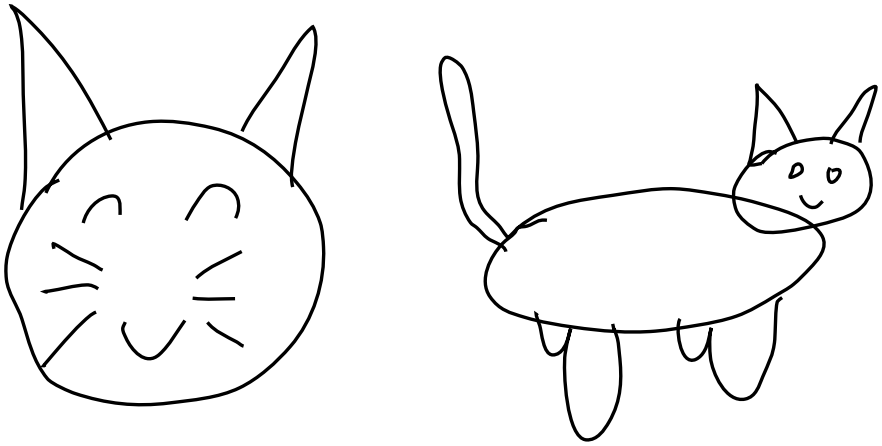

Figure 1.2: Cats drawn with and without a body

arises to extend the number of supported objects at a later time, e.g., the original artist could be unavailable.

Finally, one would have to decide how fine-grained the replacement should be. Should the system have a separates replacements for a face drawn with and without a neck, or should it ignore the neck and replace the whole drawing with a ”perfect” face? This question arises when objects are more complex than simple geometric shapes with precise target forms, which is not uncommon when dealing with sketches. Sketch drawings are highly subjective and very abstract; how an object or idea is represented in a sketch can vary from person to person [41]. For example, some people might choose to draw a cat with both the head and the body, while others may choose to omit the body (see Fig. 1.2).

Another way to aid the user could be to predict the next stroke the user would draw. In such an example, the user could draw only half of the circle, and the system would be able to

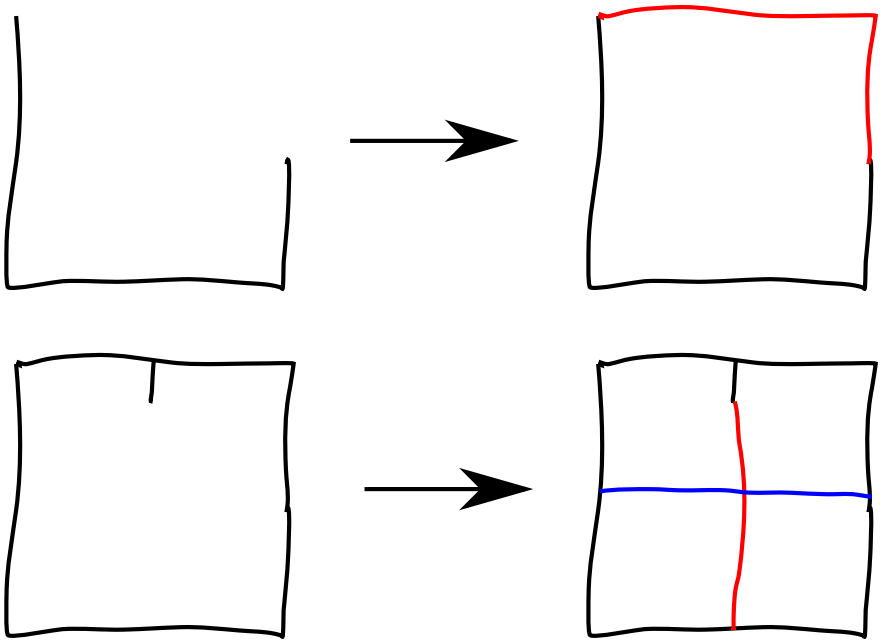

Figure 1.3: Top: a model completes the square with red color. Bottom: a model completes a window by filling in the frames; a small black stroke beginning the vertical frame ”hints” to the model a window is being drawn. In both cases, the model must use the prior sequence to deduce the target object.

fill in the next stroke to complete the circle. When drawing a window, the user could draw the start of the frames, and the frames would be completed by the system (see Fig. 1.3).

To achieve this with deep learning, we would want a model that can predict the next position of the pen given the sequence that is already drawn. Such a model can learn from pre-existing drawings, meaning it requires less human intervention to make the system. However, learning from humans also means learning all the human flaws and properties present in the drawings, like irregular strokes and varying degrees of drawing quality.

With this method, we are improving the drawing speed rather than the quality of the drawings. While there are several way to improve the sketching experience, it is specifically this type of ”assistance” we will be focusing on in this thesis.

1.2.1 Structure of the thesis

In chapter 2, we give a broad introduction to the different topics that are relevant to creating an artificial drawing assistant. Then, in chapter 3, we show how it is possible to draw on a

computer. Moreover, we discuss two common ways of representing sketches on a computer, including the advantages and challenges of each representation. In chapter 4, we review the field of using deep learning for sketch processing. We introduce several tasks within the domain as well as relevant research. The SketchRNN model is introduced, and we round out the chapter by discussing the difference between the goals of this thesis and previous work.

In chapter 5, we introduce a base "decoder-only" model that we use to achieve our goal of creating an artificial drawing assistant system. The details of the QuickDraw dataset are outlined, and we show how the model can be trained and used. An extension of the base model, as well as an "encoder-decoder" architecture is introduced in an effort to improve performance. In chapter 6, we train the model on different categories of drawings in the QuickDraw dataset. Several versions of the model are tested, and the performance of each version is tested both quantitatively and qualitatively. In chapter 7, we finally build a drawing application that employs the model for drawing assistance capabilities.

Chapter 2

Preliminaries

In this section, we will give an overview of the elementary elements important to this thesis. First, we explain how images and drawings can be represented on a computer, and we introduce the RDP algorithm. Second, we explain what machine learning is and introduce the concept of a model. Then we delve deeper into the topic by introducing neural networks, a family of models that is the main drive behind many modern systems. Next, we explain two variations of neural networks used in this thesis. Finally, we briefly explain Gaussian Mixture Models.

2.1 Computer Graphics

In a raster image, the image is described as a grid of discrete pixels where each pixel represents the color and brightness of the image at the pixel's position [6] (see fig. 2.1). With grayscale images, it suffices to keep a single value for each pixel (representing the "whiteness"), while colored images require one value for each primary color, red, green and blue (RGB). The number of values associated with each pixel is called the number of *channels*, e.g., an RGB image is said to have three channels.

A raster image can be saved on a computer as a 3-dimensional array where the first two dimensions represent the pixel's position, and the third dimension represents the channels.

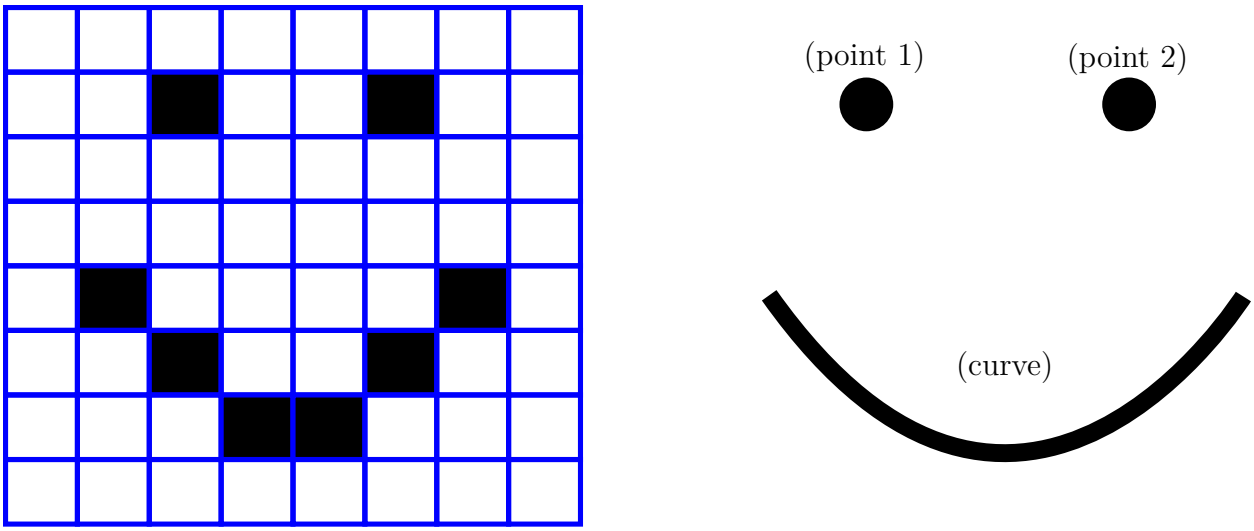

Figure 2.1: Left: example raster image with 8 by 8 resolution; the blue lines represents the border between the pixels. Right: the same object in a vector image consisting of two points and a curve. The curve is defined by two endpoints and a set of parameters controlling the curvature (not shown).

For example, a colored raster image with a resolution (i.e., grid size) of 256×256 can be represented as an array of shape (256,256,3). Raster images are generally good at representing objects with high fidelity and texture, like real-life photographs.

A vector image, on the other hand, is a way of describing objects with geometric shapes [6]. Instead of saving the pixel value of each position, the computer saves the parameters of geometric objects. Depending on the object to represent, a vector representation may present some advantages. For example, a uniformly colored rectangle can be represented with only seven values; four parameters for the corner positions and three parameters for the color. A raster representation may need significantly more parameters depending on the resolution; a 256×256 image requires $256 \times 256 \times 3 = 196608$ values. Another advantage is that vectorized objects do not lose their fidelity if resized. Vector images can also readily be converted to raster images, which is not easy the other way around. In fact, any time a vector image is displayed on a physical display (composed of pixels), it has to be converted to some form of a raster image.

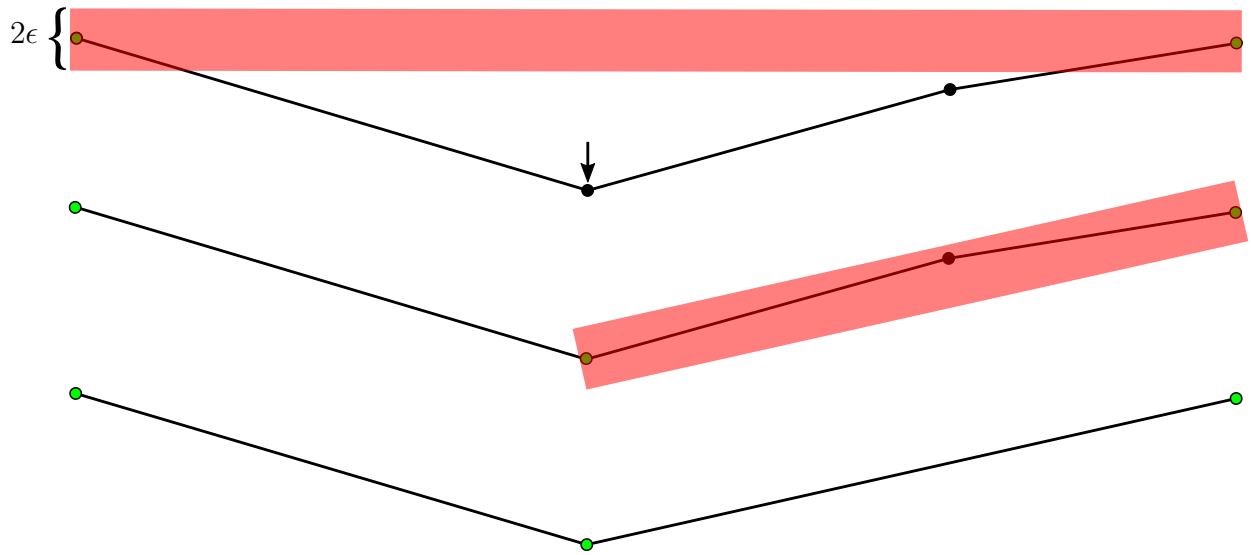

Figure 2.2: The RDP algorithm on a four-point stroke, resulting in one point being removed. At the first step, both endpoints are automatically marked to be kept. The second point from the left has the largest perpendicular distance to the red area, so it will be marked as kept. At the second step, we have split the challenge recursively into the left and right side of the previous chosen point. There is nothing to do on the left side, but on the right side the middle point is within the red area, and so will be discarded. Finally, since no more points can be discarded, the procedure ends.

2.1.1 Ramer-Douglas-Peucker

Ramer-Douglas-Peucker (RDP) [10] is an algorithm that simplifies lines. When given a curve consisting of line segments, the algorithm aims to find a similar curve with fewer segments.

The algorithm works by recursively dividing the line. On each divided segment, the first and last points are marked to be kept, then it finds the point p on the segment with the largest perpendicular distance to a straight line going through the start and endpoint. Suppose the perpendicular distance of p is larger than ϵ. In that case the point is marked to be kept, and the process is recursively applied to the line segments from the start point to p, and from p to the end point. If, however, the distance is smaller than ϵ, then p is discarded. When no more points can be discarded, the procedure ends. See figure 2.2 for a simple example of the algorithm applied to a four-point line.

2.2 Machine Learning

A machine learning model is an algorithm that can learn from data [15]. To learn from data involves using experience E to improve the performance measure P on a task T. To make the idea more concrete, consider predicting the amount of ice cream sold on a beach based on the temperature outside. The task T is to predict ice cream sales. From the experience E, one might sense that low temperatures lead to low sales, and high temperatures lead to high sales. A computer can likewise use this experience to create a model which predicts the number of sales given the temperature. To analyze the performance, the model is tested on some performance measure P, for example the average error between the actual value and the value from the model. A lower average error corresponds to a higher performance P, corresponding to a better model.

The task T can be quite abstract, like making drawings; however it is often easier to define the task indirectly through the performance measure, which we want to optimize over a set of *examples* [15]. An example encapsulates a part of the experience E, and it consists of a set of sample observations or *features* from the event we wish to model. If we observe m features from the event, then an example x is typically represented as an m-dimensional

vector. Usually, the experience E is represented as a dataset X, e.g., if we have n examples from our experience, then $X = \{x_1, x_2, ..., x_n\}$.

Many tasks boil down to solving regression or classification tasks [15]. In the case of regression, the algorithm is tasked to produce a function that inputs a vector of m observations and outputs a real-valued output (or a vector of real-valued numbers). The ice cream example above is a regression task with an $m = 1$ dimensional vector consisting of the observed temperature, and the output is the predicted amount of ice cream sales. In the case of classification, the model must output a predicted category, or express a probability distribution over the existing categories.

Machine Learning algorithms are broadly divided into supervised algorithms and unsupervised algorithms. In supervised learning, we wish to predict some value y given a set of input features x. Usually, both x and y are contained in the same dataset X where y is a subset of the m available features (and x contains the features not in y). Formally we assume that there is some underlying function f such that

$$y = f(x) + \epsilon$$

Where ϵ is an error term representing all the information about y that is not captured in x (including random noise). That is, f encodes the proper relation between x and y [20]. To approximate this relation, we want the learning algorithm to produce a model $\hat{f}$ such that $\hat{f}$ is similar to f. In addition to being a regression task, the ice cream example is also an example of a supervised task where the temperature is x, and the number of ice creams sold is y.

In unsupervised learning, there is no specific supervision signal y. If we assume that the underlying process from which the actual data originates forms a probability distribution $p(x)$ over the sampling process, then unsupervised learning can be described as either learning the distribution itself, or some of its properties [15]. By learning a sampleable approximation $\hat{p}(x)$ of the true probability distribution $p(x)$, we can generate novel samples that appear as if they came from the true distribution. For example, by learning a probability distribution over a dataset of drawings, we get a model that can be sampled for novel drawings.

There is generally no hard line between a supervised and unsupervised task; in fact many unsupervised tasks can be solved by reformulating them in a supervised manner. For

example, the chain rule of probability states that if we have an example $x \in \mathbb{R}^m$, then $p(x) = \prod_{i=1}^{m} p(x_i|x_1, ..., x_{i-1})$, which means that we can model $p(x)$ by splitting the task into modelling n conditional distributions, i.e. n supervised learning tasks [15].

In addition to supervised and unsupervised learning, there is also Reinforcement Learning (RL), where the model or the *agent* essentially needs to gather the experience "on its own" by interacting with an environment [36]. In chapter 4, we will see some efforts to take advantage of this machine learning paradigm.

2.3 Artificial Neural Networks

This section introduces the model family of Artificial Neural Networks (ANNs) and relevant models within this family.

2.3.1 Introduction - Feedforward Neural Networks

The quintessential architecture of an ANN is the *Feedforward Neural Network*. A feedforward network defines a mapping $\hat{y} = \hat{f}(x, \Theta)$ where Θ is a set of *learnable parameters*, which means that they can be adjusted or *learned* in a way that makes $\hat{f}$ approximate a (true) relation f. Such a mapping is often written as $\hat{f}(x)$ where the parameters Θ are implicit.

Following the example from Goodfellow et al. [15], a network might consist of 3 layers: $\hat{f}(x) = \hat{f}^{(3)}(\hat{f}^{(2)}(\hat{f}^{(1)}(x)))$ where the first layer $\hat{f}^{(1)}$ is called the input layer, and the last layer $\hat{f}^{(3)}$ is called the output layer. During training, examples show how the network should map inputs x to output y. The examples only specify what the final layer should do, i.e. produce predictions $\hat{y}$ that are close to y. The desired output of the other layers ($\hat{f}^{(1)}$ and $\hat{f}^{(2)}$) is unspecified in the training data, so the learning algorithm is free to decide how to use these layers in order to increase the performance of the last layer.

In general, layers that are only used internally are called *hidden layers* [15]. Hidden layers transform the input x into a new set of features called a *latent space representation*. Optimally, this representation makes it easy for the last layer to map the representation to

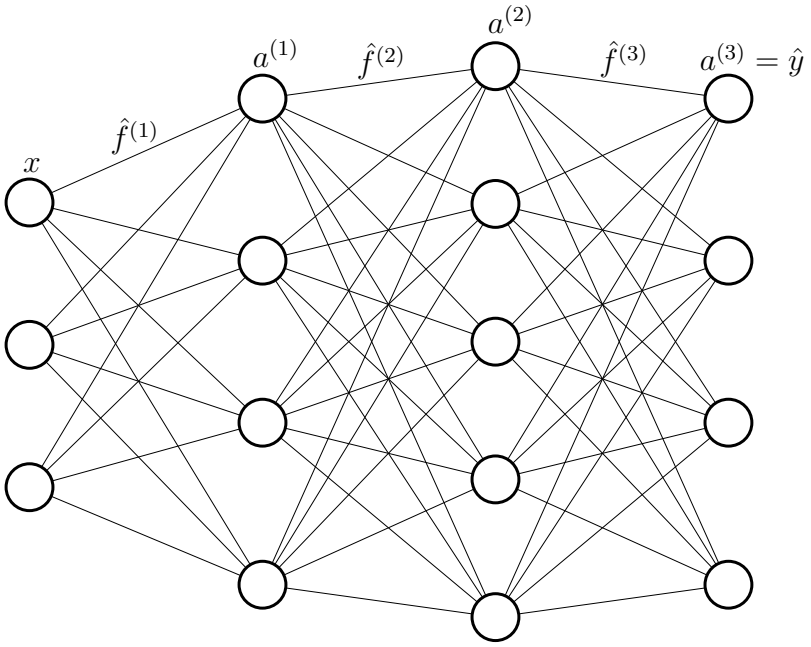

Figure 2.3: Figure of a 3-layer neural net with a 3-element input x and 4-element output y. The round nodes, called "neurons", are the individual values of each layer. The lines between the nodes represent the functions. All together, the structure of the model resembles a network of nodes, hence the name "neural network".

the correct output. We can think of the latent space representation as describing x in a way that is relevant for predicting y

In a simple case, each hidden layer in a network is an extension of a *linear* layer. A linear layer g with n-dimensional input x and m-dimensional output z is defined as an affine transformation:

$$z = g(x) = Wx + b$$

where $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. A major shortcoming of linear models is that they cannot model non-linear relationships. We can extend the linear model to a non-linear mapping by applying a non-linear *activation function*. If z is the input to an activation function σ, then $a = \sigma(z)$ are the *activations* of z. Note that an activation function is usually applied *element-wise* to the values in the input.

With this knowledge, the example above can be expanded to reflect how a simple feed-

forward neural network operates:

$$
\begin{aligned}
a^{(1)} &= \hat{f}^{(1)}(x) = \sigma^{(1)}(W^{(1)}x + b^{(1)}) \\
a^{(2)} &= \hat{f}^{(2)}(a^{(1)}) = \sigma^{(2)}(W^{(2)}a^{(1)} + b^{(2)}) \\
a^{(3)} &= \hat{f}^{(3)}(a^{(2)}) = \sigma^{(3)}(W^{(3)}a^{(2)} + b^{(3)})
\end{aligned}
$$

The predicted value $\hat{y}$ is then equal to the activations $a^{(3)}$ of the last layer. The parameters Θ of the network consists of the weights and biases (W and b) of each layer. Figure 2.3 is an example of a neural network using this structure. This neural network is called a *fully connected network*, since each value in a layer is computed using all values in the previous layer.

2.3.2 Training neural networks

To train neural networks, we define an minimization target called a *differentiable loss function*. Based on a true input x and output y from the training data and the parameters of the network Θ, the loss function $L(x, y, \Theta)$ defines an error between the true and predicted value [15]. The loss function should be designed such that minimizing it results in better performance on the actual task T. Since the loss function is made differentiable, the gradients of the loss with respect to the parameters of the final layer can be calculated. Moreover, the *backpropagation* algorithm can calculate the gradients of the loss with respect to prior hidden layers [15].

For example, in a fully connected feedforward network, the output of one layer only directly affects the output of the following layer. Hence the gradient of the loss w.r.t. the current layer is a function of the following layer gradient and the gradient of the following layer w.r.t. the current layer. Informally, the gradients are said to "flow" from the final layer backwards into the previous layer, which further flow into the layer preceding the previous layer, and so on.

In traditional Batch Gradient Descent, the gradient used to optimize the network is calculated as the mean over the gradients of all examples in the dataset. For large datasets, this method is too computationally expensive; instead, a better method is to calculate the

gradient only over a small sample of m examples at each training step. This method of optimization is called Stochastic Gradient Descent (SDG) [15]. When the number of examples in the sample is not one, it is also called Mini-batch Gradient Descent, where each sample set is called a mini-batch.

When the gradients $\boldsymbol{g}$ of a mini-batch have been calculated, the loss can be reduced by making a small optimization step in the opposite direction, i.e. $\Theta \leftarrow \Theta - \eta \boldsymbol{g}$ where η called the *learning rate*. In practice, a more intricate algorithm to calculate the gradient step is used, such as the ADAM [23] optimizer.

To analyze the performance of a trained machine learning model, it is essential to test it on *unseen* data [20]. It is common to partition the data into three independent parts: the training set, the validation set, and the test set. The training set contains the examples used for the learning algorithm, while the test set is used to test the model after training. The validation set is similar to the test set; however, it is used *during* training to estimate the performance on the test set. This performance estimate is useful for performing model selection and hyperparameter tuning.

2.3.3 Activation functions

In the context of deep learning, activation functions are non-linear functions that enable a neural network to model non-linear relations between the input x and the output y.

Three of the most common activation functions are the logistic function (σ), the hyperbolic tangent (tanh), and the Rectified Linear Unit (ReLU) (see fig. 2.4). These functions are defined in the following manner:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}, \quad \mathrm{ReLU}(x) = \max(0, x)$$

The hyperbolic tangent and the logistic function are in a class of functions called sigmoid functions. This class suffers from the problem of saturating gradients. Suppose the result of an affine transformation causes a neuron in the network to have very high or low values. In that case the derivatives of the activation function become extremely small (see fig. 2.4). When training with gradient descent, the small derivatives makes the gradient propagation through the network diminish, which causes slow convergence times [37].

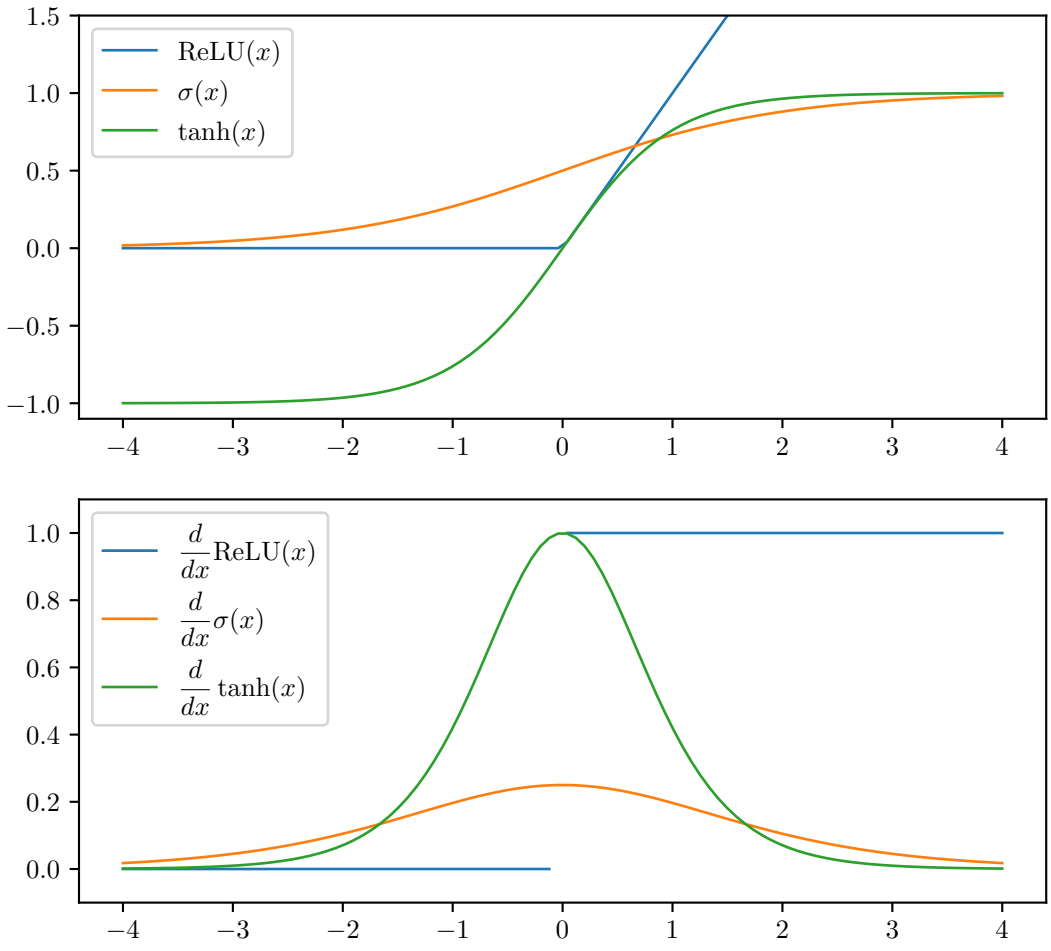

Figure 2.4: Top: Activation functions; bottom: derivatives of the activation functions

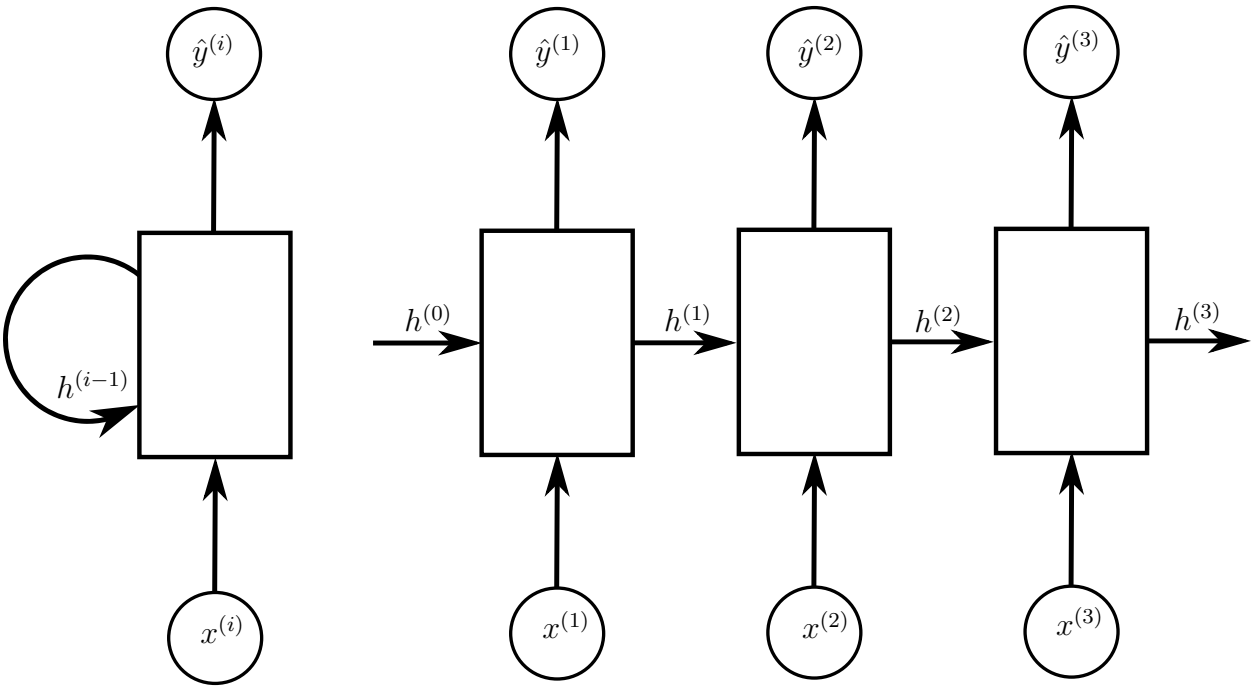

Figure 2.5: A recurrent neural network. Left: "Actual" architecture, note that the hidden state at the previous time step serves as input to the current time step. Right: "Unrolled" over the three first time-steps.

The ReLU function avoids this issue when the input is positive, in which case the derivative is always 1; the function is also highly computationally cheap. Hence, in newer feed-forward networks, the sigmoid and logistic functions have mostly been replaced by the ReLU function. However, we will see in the following section that they are still applied in recurrent neural network architectures.

The *softmax* activation function can be employed in classification tasks to ensure the output expresses a probability distribution over the categories. The softmax function ensures that all output values are between zero and one and that the sum over the elements equals one. If there are M elements in the input x, then the output of the softmax at index i is defined as

$$\text{softmax}_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{M} e^{x_j}}$$

2.3.4 Recurrent neural networks

Recurrent Neural Networks (RNNs) are a family of neural network architectures for processing sequential data [15]. They differ from feedforward neural networks in that output information is fed back into the model. A consequence of this architecture is that RNNs are capable of processing inputs of a variable length.

RNNs keep a hidden state h (i.e., a memory) which at each time step is updated as a function of the current input and the hidden state at the previous time step. Identical weights are used to process every point in the sequence regardless of the time step, enabling the model to more easily recognize patterns and information regardless of where they occur (assuming that similar patterns occur throughout the sequence).

For example, a typical RNN architecture (essentially an Elman network [12]) can be defined in the following manner:

$$
\begin{aligned}
z_x^{(t)} &= W_x x^{(t)} \\
z_h^{(t)} &= W_h h^{(t-1)} \\
z^{(t)} &= z_x^{(t)} + z_h^{(t)} + b \\
h^{(t)} &= \tanh(z^{(t)})
\end{aligned}
$$

At time t, the current input $x^{(t)}$ and the previous hidden state $h^{(t-1)}$ are turned into the intermediate values $z_x^{(t)}$ and $z_h^{(t)}$ respectively, which are then used to form the new hidden state $h^{(t)}$. A tanh non-linearity is common in RNNs; one reason for this is discussed in the following subparagraph.

There are several ways to use recurrent neural networks depending on the use case. One way is to use the final hidden state as a summary of a sequence, e.g., to classify sequences. Another use case is to employ it as an *autoregressive* model by making a prediction at each step. Autoregressive models are capable of using their own previous predicted values as the basis for a further prediction. For example, the hidden state at time t can be further processed (e.g., with an affine transformation) into a prediction of the following input at time $t+1$:

$$\hat{x}^{(t+1)} = \hat{y}^{(t)} = W_y h^{(t)} + b_y$$

Problems with RNNs While recurrent networks are great at modeling sequential data, the architecture presents some challenges. Since the hidden state is the only value passed on through time, an RNN needs to keep all the information of prior parts inside this state; updates of the hidden state may cause information from earlier parts of the sequence to be overwritten by newer information.

A related problem is *exploding* and *vanishing* gradients [5]. The network weights are equal at all time steps, so passing a long sequence through the network is comparable to repeatedly applying the same affine transformation to the hidden state. If this transformation tends to increase the magnitude of the hidden state, the repeated applications can quickly cause the magnitude of the hidden state (and the gradients) to "explode." A hyperbolic tangent as non-linearity strongly mitigates this problem because it ensures the elements in the hidden state are always between -1 and 1. Moreover, it is possible to "clip" the gradients, i.e., enforce that the magnitude of the gradients is not larger than some value.

On the other hand, suppose the affine transformation mainly reduces the size of the hidden state. Then the signal will decrease as we go further back, so changes to the later inputs will comparatively have much more effect on the current hidden state than earlier inputs. In other words, the gradients "vanish" when they flow back in time. Moreover, the use of a tanh function only accentuates this effect because of its saturating gradients property [5].

LSTM Long Short-Term Memory (LSTM) [19] is a specialized recurrent neural network architecture that can mitigate the adverse effects mentioned above. It is defined in the following manner:

$$
\begin{aligned}
f_t &= \sigma(W_{f,x}x^{(t)} + W_{f,h}h^{(t-1)} + b_f) \\
i_t &= \sigma(W_{i,x}x^{(t)} + W_{i,h}h^{(t-1)} + b_i) \\
\tilde{c}_t &= \tanh(W_{\tilde{c},x}x^{(t)} + W_{\tilde{c},h}h^{(t-1)} + b_{\tilde{c}}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c} \\
o_t &= \sigma(W_{o,x}x^{(t)} + W_{o,h}h^{(t-1)} + b_o) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Where $(\odot)$ denotes element-wise multiplication. The idea behind the architecture is to manage a *cell state* c, that can efficiently keep relevant old information. The updates to this

cell state are controlled through a set of operations which ensure the gradients can easily flow backward in time.

At a basic level, the architecture works by building a candidate vector $\tilde{c}$, with elements between -1 and 1. The cell state is updated by *adding* $\tilde{c}$ to c, hence for each element e in the cell state c, the model is restricted to incrementing or decrementing e by 1, or something in-between.

The architecture also employs several *gates* (f_t, i_t, and o_t) containing values between zero and one. A gate can be viewed as a weighting over the information of some vector a. At the limit (where the gate values are either 0 or 1) the element-wise multiplication of a gate with some vector a express a "choice" over the elements of a.

Using these gates, the network can ensure that only necessary information is passed to each component. The forget gate f_t "removes" the information in the cell state the model no longer finds useful. The input gate i_t weights the importance of elements in $\tilde{c}$, protecting the cell state from irrelevant modifications. The output gate o_t "extracts" the parts of the cell state which are relevant for the hidden state, and by extension relevant for the predicted value each time step.

2.3.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) is a specialized class of neural networks that has been successfully applied to model both visual and temporal data [15]. The name convolution indicates that the networks work by employing a mathematical operation called a *convolution*, analogous to sliding a weighted filter over the inputs and computing the sum of the values under the filter. We will only consider the discrete case, where the filter moves with a discrete step size instead of a continuous movement.

In the one-dimensional case, if we have an input x and a kernel of size M with weights w, then the convolution result $K(t)$ at position (e.g., time) t is defined in the following manner:

$$K(t) = \sum_{m=0}^{M-1} x^{(t-m)} w_m$$

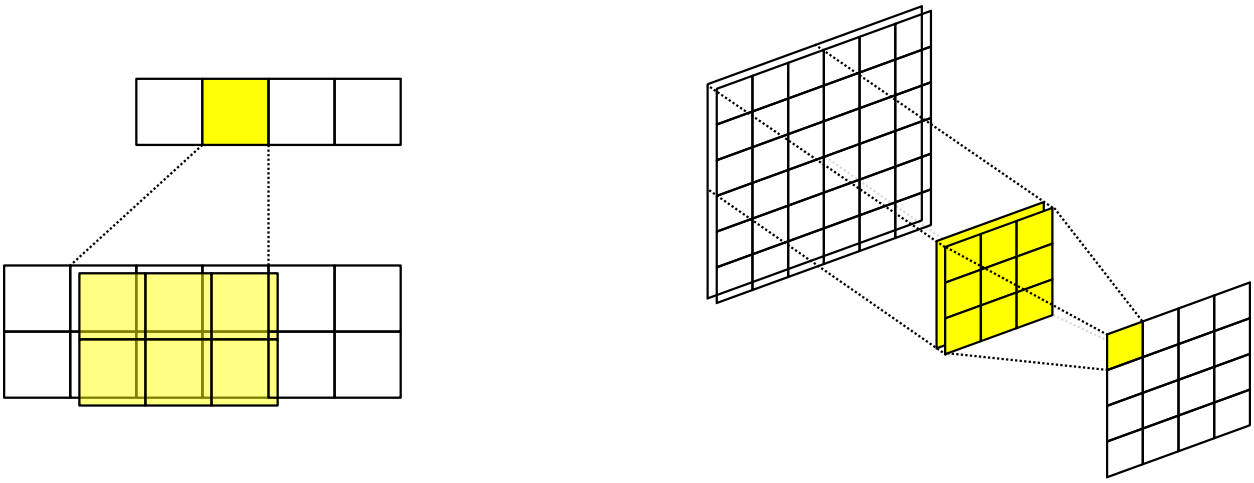

Figure 2.6: Convolutional operations with one kernel. Left: a 1D convolution over an input with two channels. Right: a 2D convolution over an input with two channels

In the two-dimensional case, we operate over two dimensions of the input (e.g., an image); the operation is defined as:

$$K(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{i-m,j-n} w_{m,n}$$

The convolutional operation essentially matches the local area under the kernel with the pattern defined by the weights of the kernel. If the structure of the weights are similar to the structure of the local area, then the operation will result in a large-magnitude output.

Sometimes the input contains several channels, in which case the number of channels in the weights needs to match the number of channels in the input. For example, colored images have three channels, which means that $x_{i,j} \in \mathbb{R}^3$, so $w_{m,n} \in \mathbb{R}^3$ as well. In general, if C_{in} is the number of input channels, then $w_m \in \mathbb{R}^{C_{\text{in}}}$ (1D case) or $w_{m,n} \in \mathbb{R}^{C_{\text{in}}}$ (2D case). Note that number of channels in a one-dimensional input is the number of values at each position (e.g., time-step).

Usually, several kernels are used in the same layer to increase the number of patterns to can match against. The result of each kernel is placed in an individual channel in the output, i.e., with C_{out} kernels, the number of channels in the output will be C_{out}.

A typical operation to use after a convolution is the *maxpool* operation. Like a convolution, the operation slides a kernel over the input. On each position, it calculates the

maximum value under the kernel, discarding the other values. A common way to use the maxpool operation on images is to use a 2×2 kernel with a stride of 2, i.e., moving it by two positions at each step. This operation has the effect of halving the image size while preserving the features that matched the kernels the most.

There are several reasons that convolutional networks can lead to improved performance compared to fully connected networks [15]. Like RNNs, they take advantage of parameter sharing. The kernel weights stay constant over the input, which means the operation will match the same patterns at all positions. Fully connected networks, on the other hand, might learn the exact position of the patterns, so moving them around can confound the networks. Kernels are usually much smaller than the inputs, making the operation computationally quick and requiring low memory. Changing the size of the kernel allows us to decide on how local the computations should be. To model photo images, for example, the trend has been to use small 3×3 kernels [35], which are excellent at picking up on texture patterns.

2.4 Gaussian Mixture Models

Sometimes we wish to be able to sample a prediction from a model as opposed to getting a directly estimated value. Networks can be designed to estimate the *parameters* of an underlying probability distribution $p(y|x)$ rather than the prediction $\hat{y}$ itself. For example, we can estimate parameters of a normal distribution and then use this distribution to sample a prediction:

$$\mu, \sigma = \hat{f}(x), \quad \hat{p}(y|x) = \mathcal{N}(y|\mu, \sigma)$$

$$\hat{y} \sim \mathcal{N}(\mu, \sigma)$$

There might be causes when the real conditional $p(y|x)$ has several peaks in y-space, i.e., given x, one can expect there to be several distinct possibilities for the true value y. A single normal distribution is unfit for this task because it is unimodal, i.e., it has only one peak.

One way to overcome this challenge is to combine several normal distributions into a single mixture distribution. These distributions are called Gaussian Mixture Models (GMMs) [15]. A GMM is defined as a weighted sum over M distinct normal distributions called the model's

components. Mathematically this can be expressed as:

$$p_{\text{GMM}}(y|\Pi, \mu, \sigma) = \sum_{m=1}^{M} \Pi_m \mathcal{N}(y|\mu_m, \sigma_m)$$

Where $\Pi \in \mathbb{R}^m$ is a categorical distribution over the components, i.e. $\sum_m \Pi_m = 1$

Chapter 3

Representing a sketch on a computer

This chapter will look at how we can enable sketching on a computer. Two natural ways that sketch drawing can be represented in this medium are presented before discussing the advantages and disadvantages of each method.

3.1 Creating and representing a sketch in a computer setting

An intuitive way of drawing on a computer involves treating the cursor as a pen; holding down the left-mouse button represents the pen being ”pressed” or ”touching the canvas”. Unlike drawing on real paper, where the pen-ink or pencil material is continuously applied to the canvas, a computer needs to operate discretely. For example, to record the movement of a cursor controlled by a mouse, the computer samples the mouse's position at short intervals of time.

When using the method described above, we need to specify how the computer should handle each mouse event, i.e., how we should save the event and visualize the effect for the user. A raster-image way to handle the event is to fill in the pixels around an area of the pen position, directly writing to the canvas grid shown on the monitor, which is somewhat analogous to how a real pencil physically writes to a paper. However, the discreteness of

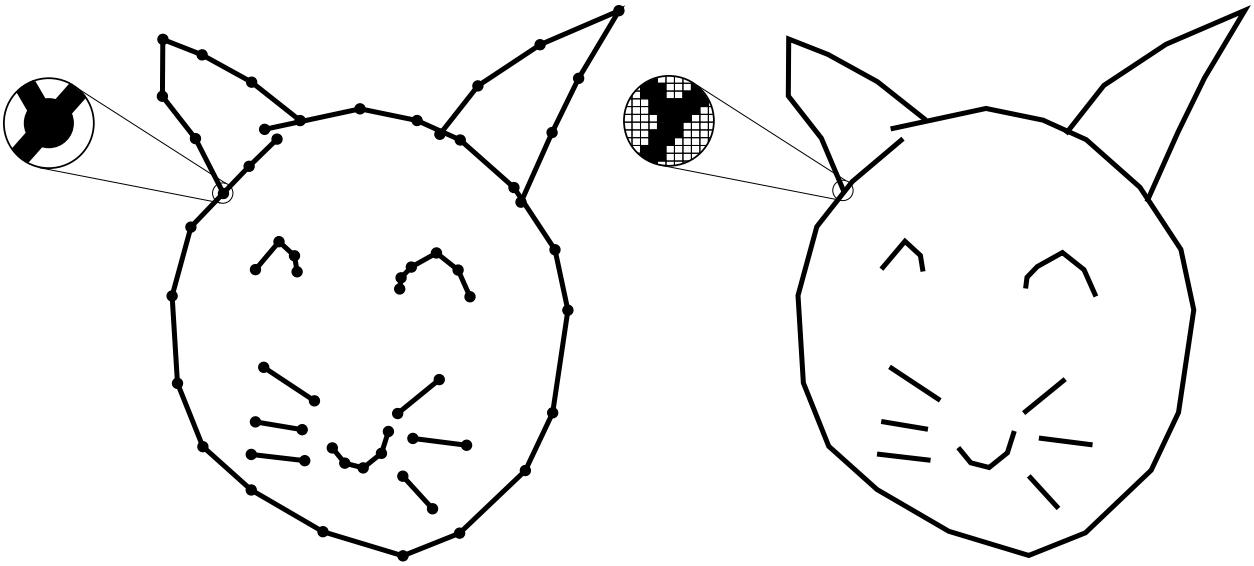

Figure 3.1: Drawing of cat in vector and raster format. The vector drawing consists of a set of straight line segments that are defined by a set of anchor points shown as the black dots on the drawing.

computers may lead to visual side-effects in this approach. For example, an inadequate sampling rate will lead to "gaps" between the points as if the pen was lifted between each point; however the problem can be mitigated by interpolating between the lines.

A vectorized approach is to save the pen positions in a list, visualizing each stroke by drawing straight lines between any two consecutive points. In many ways, this approach is more flexible for drawings; the stroke width and color can be changed dynamically before visualizing, and each point can easily be removed or manipulated. Moreover, the drawing can be scaled to any size without losing fidelity, which is useful when working with applications of differing canvas sizes.

Vector drawings are sequences of strokes, each containing a sequence of pen positions. How should the end of one stroke and the beginning of another be encoded? In the Quick-Draw [17] dataset, each pen position sample contains a binary value encoding whether the stroke ended, i.e., the value is 1 if the stroke ended and 0 otherwise. In the next chapter, we will showcase other directions that have been made to encode the relation between strokes and pen positions.

3.2 Representing a sketch for deep learning

Sketches mostly lack the details of the objects they represent, they have a sequential nature, and the background of a sketch is often empty or sparse [41]. These unique properties need to be taken into considerations when used in machine learning.

Representing sketches as raster images makes it possible to employ traditional raster image nets like CNNs to create interpretations that preserves the spatial information in the sketch. Humans perceive sketches by looking at them, so it makes sense that the spatial representation is important. However, using rasterized representation introduces side effects, like making the stroke width of the sketch and the size of the canvas affect the representation [41]. The sparseness and lack of texture of sketches are also challenges tied to a visual representation.

When representing drawings as a vector images, a helpful transformation is to reinterpret the drawing as differences in pen-movement between each pen-point and the next. This transformation makes objects have equal representation regardless of their position in the image, and thus makes movement patterns in each drawing position invariant.

From this perspective, sequential representations can be viewed as a set of instructions. For example, imagine asking what object is obtained by moving the pen 5 units to the right, 5 up, 5 left, and 5 down. One can reason about the instructions to deduce that the object is a perfect square; however, a visual representation would be much preferred. While humans perceive drawing visually, they draw sequentially. Imaging sketching a rough circle. Most people would not stop after each point to analyze the state of the circle, nor would they fill out arbitrary sides of the circle; instead, they already know how to move the hand sequentially to create the circle in one single movement.

Consider the task of predicting the next point in a drawing represented as a sequence (i.e., autoregressively drawing). In a raster representation, the order of the sequence and the current pen position is not naturally encoded in the image. It would have to be given supplementary information or manually encoded onto the image. Moreover, passing the image through a convolutional network at each step is potentially computationally expensive, and since most sketches are sparse, a large part of the computation is wasted on processing empty patches. With a sequential representation, the drawing can be encoded as a list of

pen-points ordered by the time they were sampled. The last pen position is then already encoded as the last point of the list. Moreover, a sequential network only processes the signal in the sketch, so no computation is wasted.

Sequential networks also have drawbacks. RNNs need to summarize the drawing in their hidden state, making it hard to preserve the information of the early parts of the drawing. This problem can be somewhat mitigated with LSTMs; however, even these networks struggle if the sequence is long enough [5]. Perhaps the biggest drawback to a sequential representation is losing spatial accuracy and precision (since spatial errors will accumulate). Without this information, the model might not capture the importance of the placement of the strokes. For example, windows should be contained within the walls of houses; however, a sequential model might not learn this relation sufficiently well.

Chapter 4

Background to sketch processing with deep learning

We will now produce an overview of the deep-learning sketch domain. Xu [41] published an excellent survey of the state of the field, which outlines the different tasks that have been of focus. First, we will look at datasets and several different tasks before moving on to the main topic of sketch generation.

In sketch generation, the SketchRNN model is introduced, followed by several efforts made to improve its elements. Then we review a sequence of other exciting directions, including papers employing reinforcement learning. Finally, we discuss the difference between previous work and the focus of this thesis.

4.1 Datasets

The TU-Berlin [11] dataset was released in 2012, and it is one of the first important datasets used in the field. It consists of 20,000 drawings distributed over 250 categories, and the drawings in the dataset are generally of high quality. In addition to releasing the dataset, the researchers explored several aspects of human drawings. One of the findings was that humans could recognize the correct object in 73.1% of the drawings, while a pre-deep-learning model

Figure 4.1: Left: cat from the TU-Berlin dataset; right: cat from the QuickDraw dataset.

could only successfully classify 56%. Many of the earlier deep-learning sketch-recognition models were based on this dataset.

In 2017 google released the QuickDraw [17] dataset , an ever-growing set of more than 50 million sketches across 345 categories. They achieved this by "gamifying" the data collection. A website was created where a user would be tasked to draw a given object in 20 seconds or less, and to encourage the user, an AI would continuously attempt to classify the drawing.

The dataset was a great addition to the field due to several reasons [41]. Firstly the size of the dataset enables training models of much higher capacity, whereas previous models had to be strongly regularized to work. Secondly, the objects in the dataset more accurately reflect how most real sketches look, i.e., very rough and highly abstract, which contrasts with the TU Berlin dataset of comparably much higher-quality drawings. Thirdly the dataset is very diverse as it has been collected from people of different cultures across the world.

4.2 Sketch recognition

Sketch recognition is the task of labeling drawings with their respective classes. As the name suggests, we want the computer to recognize the class of objects a drawing belongs to, refer to fig. 4.2 for an example. Deep learning research targeting this task is one of the

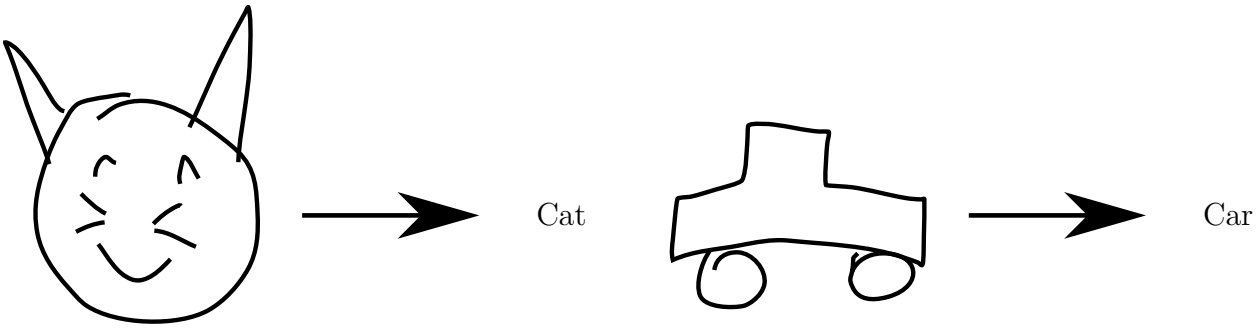

Figure 4.2: Sketch recognition

earliest in the field, and a significant variation of models and sketch representations have been conceived to optimize the task.

Yang and Hospedales [43] proposed "sketch-a-net", the first model to beat humans in the sketch recognition task. Their model was a Convolutional Neural Network that included several specific implementation details for modeling sketch data. They most notably used a large (15×15) kernel size for the first layer to better capture local stroke-structure. The model was trained and evaluated on the TU Berlin dataset, where it achieved a 74.9% accuracy, surpassing the 73.1% accuracy of humans. He et al. [18] split the drawing into 60%, 80%, and 100% completion, then passed the parts into a visual and sequential branch in parallel. Each branch would be fused in a later stage and processed by an MLP for a final classification. The trained model achieved a 79.6% accuracy on TU Berlin. Sarvadevabhatla et al. [34] treated a sketch as a series of accumulated strokes and developed a model consisting of both a CNN and an RNN to create useful representations of the sketch's temporal and spacial parts. The trained model achieved an 85.1% accuracy on a 160 category subset of TU Berlin. In addition, the architecture enabled the model to make on-the-fly predictions, i.e., predictions after each new stroke.

4.2.1 Sketch retrieval and hashing

Taking advantage of the groundbreaking size of the QuickDraw dataset, Xu et al. [42] proposed SketchMate, a deep hashing framework for sketch retrieval. Sketch retrieval is the task of querying a database for similar images to the query sketch (see fig. 4.3). Such a task requires a way to quantify the similarity between two sketches. Sketch hashing transforms a query sketch into a fixed-length sequence of bits called a hash; the goal is that sketches

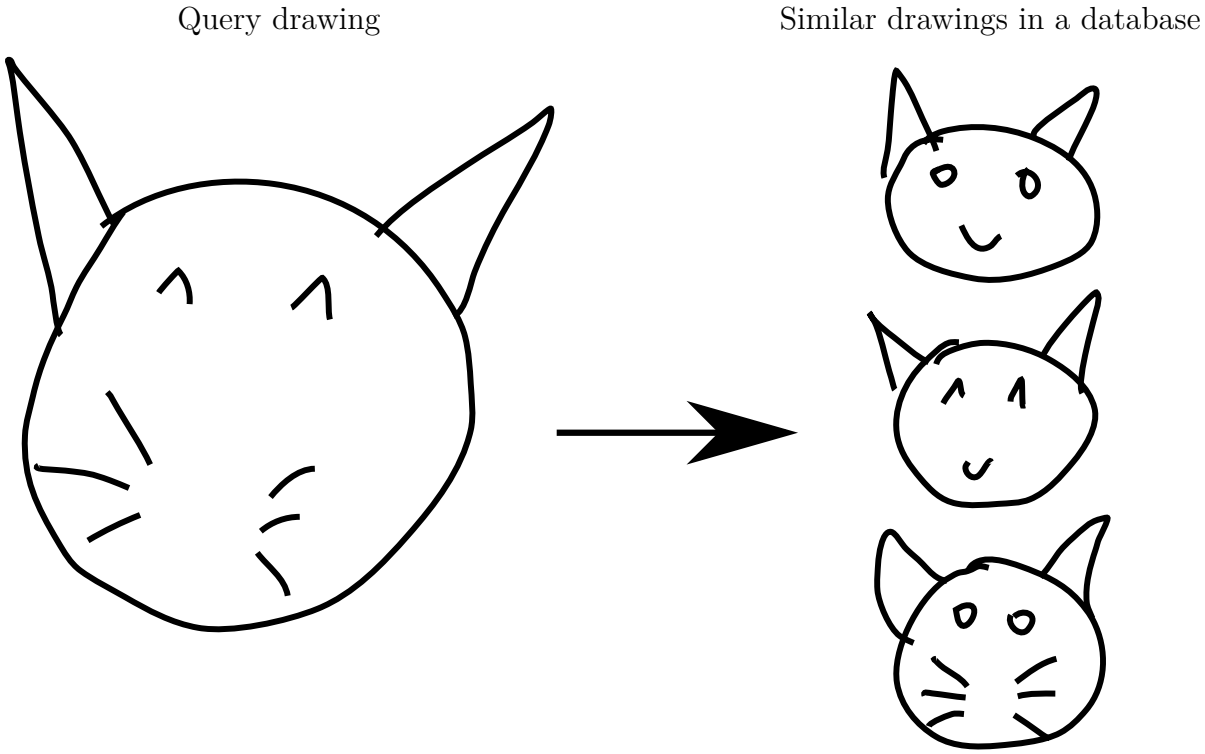

Figure 4.3: Sketch retrieval: retrieving 3 sketches that are similar to the one on the left

of similar semantics have similar hashes. Xu et al. devised a method of learning a neural mapping from sketches to hashes. A model consisting of a visual and sequential branch would process each sketch into a hash, and a specialized loss function was devised to ensure the closeness of similar drawings.

Building upon the idea of deep sketch hashing, Choi et al. [7] proposed SketchHelper, an application that provides real-time stroke guidance. The incomplete drawing is passed to the model when the user finishes a stroke, transforming it into a binary hash to retrieve the closest matching subsequent strokes. These matches are then proposed as gray shadows on the drawing, aiding the user by giving non-intrusive continuation suggestions which can be traced over. SketchHelper showcased how sketch recognition can be applied to create an artificial drawing assistant system.

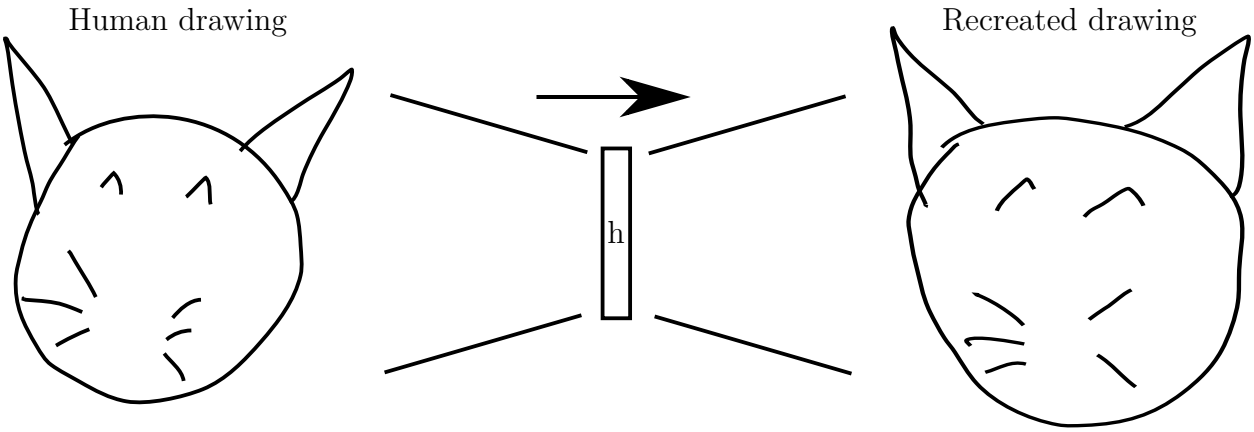

Figure 4.4: Sketch reconstruction. h is the latent space representation (encoding) of the sketch used by a decoder to create a reconstruction.

4.3 Sketch generation

Within the category of sketch generation, there are several possible sub-tasks. The goal functionality we set out to achieve in this thesis we will refer to as Generation Conditioned on Prior Sequence (GCPS), i.e. a model is conditioned on the unfinished drawing and tasked to predict the continuation (see fig. 4.6).

Sketch *reconstruction* is an unsupervised task that involves taking a reference sketch, encoding it in a latent space with an encoder, and recreating it with a decoder. Models with this capability are called *autoencoders* [15]. A subfamily of autoencoders called *variational autoencoders* enables us to sample a latent space rather than using the encoder, effectively creating "novel" sketches not present in the original dataset. Such functionality is called *unconditional* sketch generation. Variational autoencoders are essentially trained for sketch reconstruction, while the actual objective can be unconditional generation.

4.3.1 SketchRNN and improvements

In addition to releasing the QuickDraw dataset, Ha and Eck proposed SketchRNN [17], a variational autoencoder network. The model is a sequence-to-sequence network with a recurrent encoder that encodes a sketch into the parameters for a normal distribution. The distribution is sampled and passed to a specialized recurrent decoder network that attempts

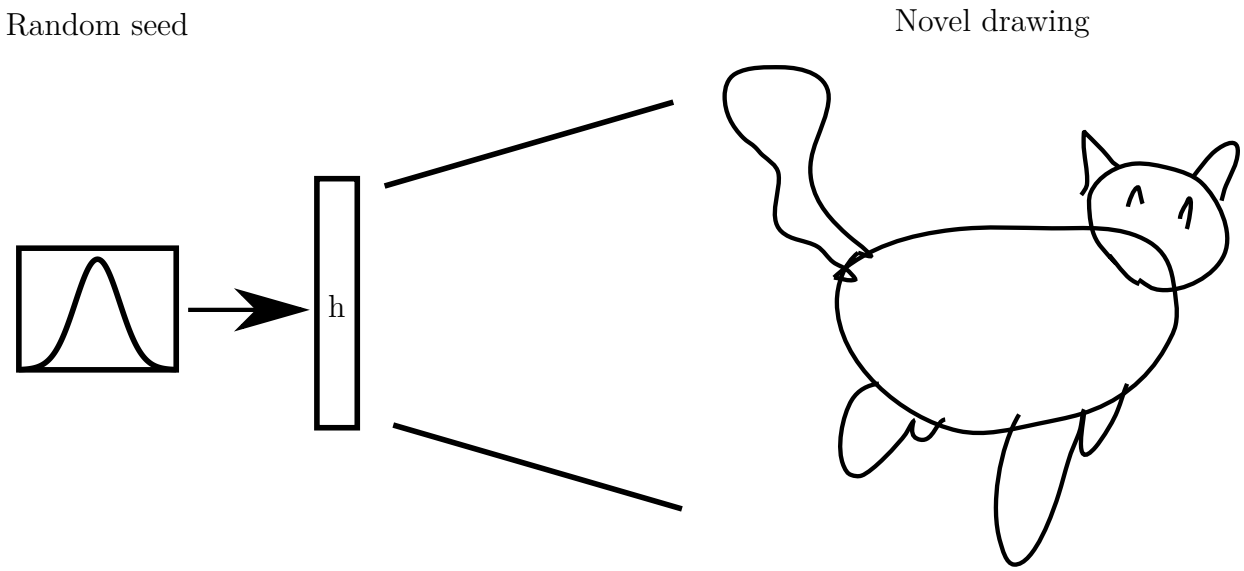

Figure 4.5: Unconditional sketch generation, a random seed is sampled from some distribution to form a latent space representation that a decoder can transform into a drawing.

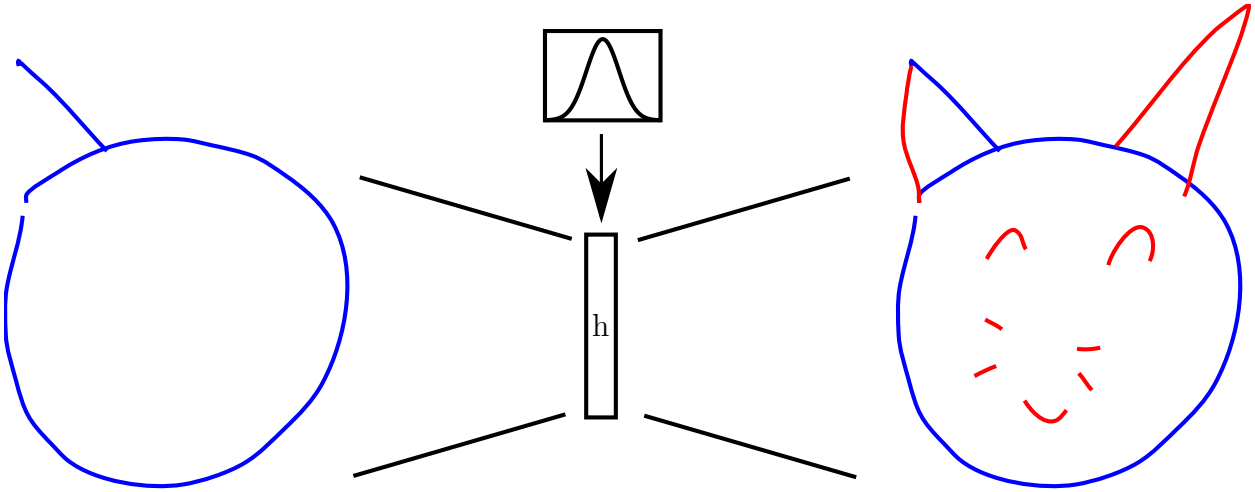

Figure 4.6: Generation Conditioned on Prior Sequence (GCPS). A random seed is combined with the encoded incomplete sequence (in blue) to create the prediction in red.

to recreate the reference image. The outputs of the decoder are parameters for a Gaussian Mixture Model, which is sampled for the final output coordinates. The model can generate sketches either unconditionally (i.e., without an input drawing) by directly sampling from the latent space of the encoder; or conditioned on a prior sequence by running the unfinished sketch through the decoder and letting the decoder generate a continuation. SkethRNN inspired a wave of works using sequential neural networks as the primary method for modeling sketch generation.

A recurring theme in the following research has been improving or swapping out components in SketchRNN. V et al. [39] introduced SkeGAN, the first Generative Adversarial Network (GAN) [14] to be used for the task of sketch generation in vector format. It consisted of an LSTM generator and discriminator, and it produced novel sketches "almost at par" with the real QuickDraw drawings. Moreover, the authors claimed that employing a discriminator loss mitigated a "scribble effect" evident in SketchRNN, i.e., a state where the model would create incoherent scribbles.

Cao et al. [4] proposed AI-Sketcher, a model architecture that seeks to deal with the "problems" in SketchRNN, referred to as a lack of quality in the reconstructed images and an inability to handle multiple classes efficiently. A convolutional branch was employed to capture spatial information, and an attention branch was added to help the model focus on parts of the image. Both branches would then be concatenated with the standard output of a SketchRNN encoder to form the input to a SketchRNN decoder.

Ribeiro et al. [33] introduced the first transformer [40] made for sketch encoding, essentially replacing the recurrent networks in the original SketchRNN with self-attention components. Lin et al. [25] continued the work using transformers and introduced an extension to the BERT [9] architecture.

Efforts have also been made to improve the representation of sketches. In addition to introducing transformers for sketches, Ribeiro et al. [33] explored other ways to represent the sketch before passing it through the model. In one experiment, they created a dictionary of codewords where similar movements, i.e., changes from one point to another, are mapped to the same entry in the dictionary. In another experiment, they assigned each point to a codeword based on its absolute position, grouping nearby points to the same token. They found that the dictionary representation outperformed the other methods, including the original SketchRNN representation.

Further exploring sketch representation, Das et al. [8] learned a Bézier embedding of the strokes. Each stroke is condensed into a fixed or variable-sized set of control points, which can reduce the input length and prevent the model from using capacity on low-level details. To model strokes defined by a varying amount of control points, the model can sequentially input each control point. Moreover, forcing all strokes to contain the same amount of control points enables representing *each stroke* as a data point, drastically reducing the input length.

Similar to the work of Das et al., Aksan et al. [1] trained an encoder to transform each stroke into a fixed-size representation. Then, a relational model would predict the next stroke based on previous strokes *without* modeling their sequential relations. The authors hypothesized that the order the strokes were created is not essential information; the sequential nature is only crucial in the context of individual strokes. The model excelled at modeling complex drawings such as flow charts.

4.4 Other interesting directions

Jaques et al. [21] proposed an interesting method of improving the quality of generated sketches. A camera would record human facial reactions to sketches generated from a SketchRNN-like model to assign a quality measurement to the sketch. With this information, a GAN component tweaked the model to favor generating sketches expected to result in positive feedbacks from humans.

Muhammad et al. [27] presented a system where a model is trained to learn which strokes in a drawing can be removed without a large reduction in recognizability. Such a model is then usable for (1) modeling the saliency of a stroke and understanding the decision of a recognition model, (2) for synthesizing sketches with variable levels of abstraction, and finally (3) for training fine-grained sketch-based image retrieval. Removing unneeded strokes in a sketch may be similar to how humans mentally sketch; a human has a more detailed mental model of the object-to-draw, but they figure out what details can be ignored without losing the most salient parts.

Reinforcement learning has also been employed for sketch generation. Ganin et al. [13] proposed SPIRAL, a system capable of recreating images by giving instructions to a painting environment. To achieve this functionality, an agent was trained to generate high-level code

for a graphics engine. This method enables the model to focus on the high-level qualities of the image, while the graphics library deals with the low-level details of each command. Mellor et al. [26] improved the model by calculating the loss at each prediction rather than after each completed drawing. Zhou et al. [44] changed the output of the SPIRAL model. The original agent created predictions as control points for Bézier curves, while the agent proposed by Zhou et al. predicted the next pen position.

4.5 Difference between previous work and the work of this thesis

Most of the sketch generation models in the field are concerned with the task of sketch *reconstruction* rather than GCPS (Generation Conditioned on Prior Sequence), a notable exception being the work of Aksan et al. [1]. While some papers mention GCPS as a valuable side-effect, we find it interesting that few known papers keep this as the primary focus.

Sketch reconstruction may pose some similar challenges as GCPS, which may cause an overlap in functionality. For example, the SketchRNN model is mainly described as an Autoencoder, even though GCPS can be achieved by excluding the encoder (this is also tested in the paper). The decoder of the model is built in a way that is not dependent on the encoder to function. Instead, the encoder acts as a guide that conditions the decoder to make the appropriate drawing. Like much of the research done, we use some elements of the SketchRNN model as a basis for our work.

However, it is not the case that all the research mentioned can be applied to GCPS. AI-Sketcher is only an improvement to the encoder-part of SketchRNN; Jaques et al. [21] focus only on improving unconditional sketch generation, and the SPIRAL models can only reconstruct drawings. While Ribeiro et al. [33] make no attempt at performing GCPS, transformers should also apply to such a task.

Chapter 5

Methodology

In this chapter, we introduce the models we will train for the task of GCPS. First, we introduce the dataset we train the models with and what preprocessing is applied to the drawings. Then, we introduce a baseline, encoder-only model. Based on the mechanics of this model, we introduce an appropriate way to train the model that enables it to achieve the task at hand. After this, an extension of the encoder-only model is presented by adding a 1D convolutional layer to model sequentially local variations in the drawing. Finally, an encoder-decoder architecture is presented to capture spatial relations in drawings.

5.1 The Data

The data used to train the models stem from the QuickDraw [43] dataset. We choose to represent the data in the same way as was done in the SketchRNN paper. If a drawing consists of T pen samples, then $S = (S^{(1)}, S^{(2)}, ..., S^{(T)})$ denotes the list of ordered data points that make up the drawing. Each data point is a 5-element vector: $S^{(t)} = (\Delta x^{(t)}, \Delta y^{(t)}, p_1^{(t)}, p_2^{(t)}, p_3^{(t)})$. Δx and Δy denotes the difference in x and y coordinates between the last and the current pen position. p_1, p_2, and p_3 denote the "pen-state" of the sample. The first value, p_1, indicates that the virtual pen was still "down" after the sample, visually meaning that a line should be drawn between the current and next point. Conversely, p_2 indicates that the pen was "lifted" after the current point and that no line should be drawn. Finally, p_3 indicates whether the drawing is finished, i.e., only the last element $S^{(T)}$ has $p_3 = 1$. Note that one and only one of p_1, p_2, or p_3 can be 1 at any time, while the other values are 0.

5.1.1 Preprocessing

We use the preprocessed "Simplified Drawing files" from the official repositories [16] of the QuickDraw dataset and further scale the sequence with the same method as the SketchRNN paper. In summary, the data has been preprocessed in the following manner:

- Each drawing is shifted such that the minimum values of each axis is 0, and uniformly scaled to have a maximum value of 255.
- The RDP algorithm is applied to each sketch with $\epsilon = 2.0$ (see subsection 2.1.1)
- A single standard deviation $\sigma_{\Delta_{x,y}}$ is calculated over all pen movements $\Delta x, \Delta y$ in the dataset, and the data is normalized by the value, i.e. $\Delta x \leftarrow \frac{\Delta x}{\sigma_{\Delta_{x,y}}}$, $\Delta y \leftarrow \frac{\Delta y}{\sigma_{\Delta_{x,y}}}$, where $(\leftarrow)$ means the variables are reassigned as the newly computed values.

Note that the first two points are applied *before* the drawings are transformed to represent the difference in pen coordinates (Δx and Δy).

Applying RDP ensures that each point encodes a notable change in the pen's position, ensuring that all points carry significant information. A consequence of applying RDP, is that the temporal information implicit in the distance between each point is removed. However, it is advantageous to RNNs that the lengths of sequences are reduced. Normalizing the data ensures that the relative distances generally have a reasonable magnitude (i.e., close to 1), which can greatly reduce convergence time [24].

5.2 The decoder-only model

The first model we introduce consists of an RNN R and a fully connected layer F (see fig. 5.1). This model mimics the decoder in the SketchRNN [17] model and will be referred to as the *decoder-only* model.

At a time step t, the model receives the current data-point $S^{(t)}$, which, together with the last hidden state $h^{(t-1)}$, is processed into a new hidden state $h^{(t)}$. The fully connected

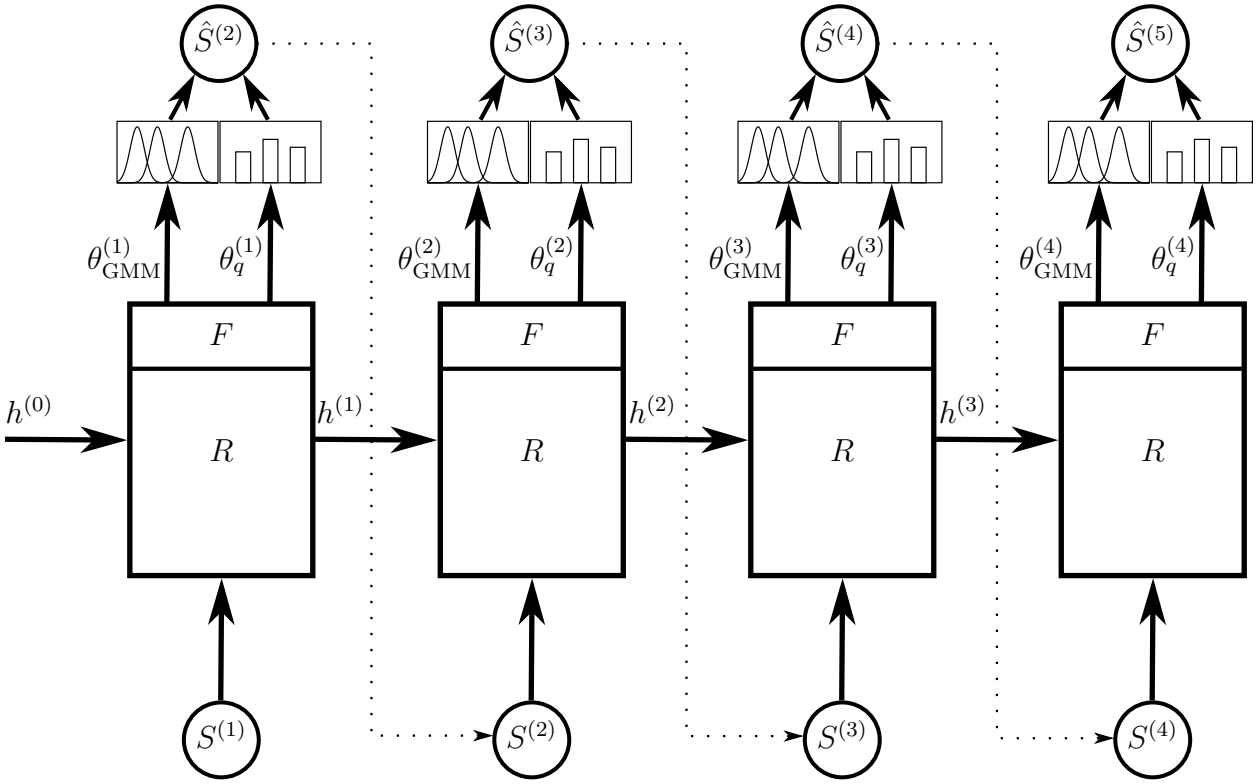

Figure 5.1: Base encoder-only model unrolled over 4 steps. The recurrent unit R transforms the previous hidden state and the current input into a new hidden state $h^{(t)}$. The fully connected layer F then transforms $h^{(t)}$ into a set of parameters used to sample the next pen-point. The dotted lines show that the output can be used as next input.

layer further transforms the hidden state into a set of parameters for a Bivariate Gaussian Mixture Model, as well as parameters for the pen-state, mathematically:

$$h^{(t)} = R(S^{(t)}, h^{(t-1)})$$
$$\tilde{\theta}^{(t)} = \{\tilde{\theta}^{(t)}_{\text{GMM}}, \tilde{\theta}^{(t)}_{p}\} = F(h^{(t)})$$

where $\tilde{\theta}_{\text{GMM}} = (\mu_x, \mu_y, \tilde{\sigma}_x, \tilde{\sigma}_y, \tilde{\rho}, \tilde{\Pi})$ are the parameters for the Gaussian Mixture, and $\tilde{\theta}_p = (\tilde{p}_1, \tilde{p}_2, \tilde{p}_3)$ are the parameters for the categorical distribution over the pen-state. Note that with M mixture components, since each components requires six parameters and there are three parameters for the pen-state, the total size of the output for each data-point is $6M+3$.

The model's outputs need to be further transformed to work as probability distribution parameters. A standard deviation only makes sense if its positive and nonzero, and a correlation must be between -1 and 1. The final pen parameters $\hat{p}_1, \hat{p}_2, \hat{p}_3$ and the component parameters Π must behave like the parameters of categorical distributions, i.e. $\hat{p}_1+\hat{p}_2+\hat{p}_3 = 1$ and $\sum_{i=1}^{M} \Pi_i = 1$. To achieve this we can use the exponential function, the tanh function, and the softmax function in the following manner:

$$\sigma_x = e^{\tilde{\sigma}_x} \quad \sigma_y = e^{\tilde{\sigma}_y} \quad \rho = \tanh(\tilde{\rho}) \quad \Pi_i = \frac{e^{\tilde{\Pi}_i}}{\sum_{k=1}^{M} e^{\tilde{\Pi}_k}} \quad \hat{p}_i = \frac{e^{\tilde{p}_i}}{\sum_{j=1}^{3} e^{\tilde{p}_j}}$$

From this we get the final parameters $\theta_{\text{GMM}} = (\mu_x, \mu_y, \sigma_x, \sigma_y, \rho, \Pi)$ and $\theta_p = (\hat{p}_1, \hat{p}_2, \hat{p}_3)$, and we can now sample the next point. The index $k \in \{1, 2, 3\}$ is sampled from the categorical distribution with parameters $\theta_p^{(t)}$, and determines which part of the pen-state should be 1, while the other elements are implicitly 0. Note that a caret over S_t, i.e., $\hat{S}_t$, represents that the point is a *prediction* rather than a true point from the dataset.

$$(\Delta x, \Delta y) \sim \text{GMM}(\theta^{(t)}_{\text{GMM}})$$
$$k \sim \text{Categorical}(\theta^{(t)}_{p}) \quad p_k = 1$$
$$\hat{S}^{(t+1)} = (\Delta x, \Delta y, p_1, p_2, p_3)$$

Now $\hat{S}^{(t+1)}$ can be fed back into the model to create $\hat{S}^{(t+2)}$ (and so on), which showcases that the model can work autoregressively. The initial data-point $S^{(0)}$ denotes the "Start of Sequence" (SOS) and is set to (0,0,1,0,0), which enables sampling from the model without any prior sequence. Since $p_3 = 1$ indicates that the drawing has finished, we may stop the process when such a value is sampled.

Following the SketchRNN [17] paper, we adjust the randomness of the sample by introducing a temperature parameter $\tau \in \mathbb{R}^+$ that changes the sample parameters in the following manner:

$$\sigma_x \leftarrow \sigma_x\sqrt{\tau}, \quad \sigma_y \leftarrow \sigma_y\sqrt{\tau}, \quad \tilde{\Pi} \leftarrow \frac{\tilde{\Pi}}{\tau}, \quad \tilde{p} \leftarrow \frac{\tilde{p}}{\tau}$$

Note that the standard deviation is adjusted after the exponential function, while the categorical values are adjusted before the softmax.

When $\tau < 1$, the standard deviations shrink, causing the mixture components to become "sharper." Similarly, dividing $\tilde{\Pi}$ and $\tilde{p}$ by a small τ has the effect of accentuating the differences between the classes, effectively pushing the probability of the most likely class up. When $\tau > 1$, the exact opposite happens, making the sampling broader. Note that by using a temperature different than 1, we are essentially sampling from a different distribution than the one learned during training.

5.2.1 Training the model

We can use the maximum likelihood principle to train the model [15]. In other words, we want to maximize the probability the model gives a human drawing. More formally, given a sequence S from the dataset, we want to maximize:

$$p_{\text{model}}(S) = \prod_{i=t}^{T} p_{\text{model}}(S^{(t)}|S^{(t-1)}, S^{(t-2)}, ..., S^{(1)})$$

Fixing t, since the model is an RNN, the estimated probability of $S^{(t)}$ is a function of $S^{(t)}$ itself, the previous input $S^{(t-1)}$ and the hidden state $h^{(t-2)}$ after feeding the model $(S^{(1)}, S^{(2)}, ..., S^{(t-2)})$. Mathematically:

$$p_{\text{model}}(S^{(t)}|S^{(t-1)}, S^{(t-2)}, ..., S^{(1)}) = p_{\text{model}}(S^{(t)}|S^{(t-1)}, h^{(t-2)})$$

The probability assigned to the true point is then the product of the estimated pen-movement probability $P_S^{(t)}$ and the estimated pen-state probability $P_P^{(t)}$. We have:

$$\begin{aligned}
\{\theta_{\text{GMM}}^{(t-1)}, \theta_p^{(t-1)}\} &= \text{model}(S^{(t-1)}, h^{(t-2)}) \\
P_S^{(t)} = p_{\text{model},S}(S^{(t)}) &= \text{GMM}(\Delta x, \Delta y|\theta_{\text{GMM}}^{(t-1)}) \\
P_P^{(t)} = p_{\text{model},P}(S^{(t)}) &= \text{Categorical}(k|\theta_p^{(t-1)})
\end{aligned}$$

Where $(\Delta x, \Delta y)$ is the coordinate change at time t, and k is the index of the true pen-state (at time t). Thus for the whole sequence, we want to maximize:

$$p_{\text{model}}(S) = \prod_{t=1}^{T} P_S^{(t)} \times P_P^{(t)}$$

We would like to convert this target into a loss function such that gradient descent can be applied. Note that maximizing this probability is the same as maximizing the log-probability. By the rules of logarithms, we have:

$$\begin{aligned}
\log(p_{\text{model}}(S)) &= \log(\prod_{t=1}^{T} P_S^{(t)} \times P_P^{(t)}) \\
&= \sum_{t=1}^{T} (\log P_S^{(t)} + \log P_P^{(t)}) \\
&= \sum_{t=1}^{T} \log P_S^{(t)} + \sum_{t=1}^{T} \log P_P^{(t)}
\end{aligned}$$

Thus maximizing the likelihood given to the sequence by the model is the same as individually maximizing the sum of log pen-state probability and log pen-movement probability. We want to reformulate the target as a loss function we can *minimize*. To do this we can negate the equations; denote L_S as the loss for the pen-movement and L_P as the loss for the pen-state, then

$$\begin{aligned}
L_S &= -\sum_{t=1}^{T} \log P_S^{(t)} \\
L_P &= -\sum_{t=1}^{T} \log P_P^{(t)} \\
L &= L_S + L_P
\end{aligned}$$

Since both equations are differentiable, they can be used as the optimization target for gradient descent. If a mini-batch has size n, then the cost function J with respect to the parameters of the model Θ is defined as the mean loss over the drawings in the mini-batch:

$$J(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L_i = \frac{1}{n} \sum_{i=1}^{n} \left(-\sum_{t=1}^{T_i} \log p_{\text{model,S}}(S_i^{(t)}|\Theta) - \sum_{t=1}^{T_i} \log p_{\text{model,P}}(S_i^{(t)}|\Theta) \right)$$

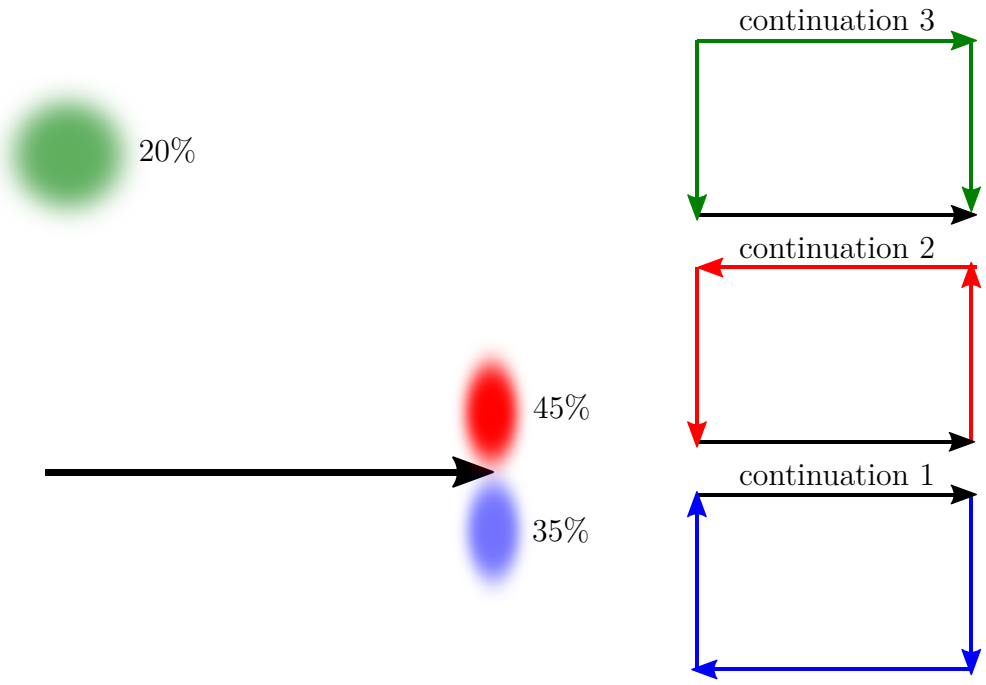

Figure 5.2: Left: Example of how a GMM with M=3 might estimate the next point given the prior sequence in blue. Each color is the estimation of one individual component. Right: how the sequence might continue given the choice of component during sampling.

To improve the model we make an optimization step using the gradient of the cost function with respect to the parameters, $\boldsymbol{g} = \nabla_{\Theta} J(\Theta)$. This thesis uses the ADAM [23] algorithm with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ to make the optimization step.

5.2.2 Why use a Gaussian Mixture Model

We will now briefly motivate why GMMs are common in sketch modelling applications. Say the task is to draw a square, and one begins the sequence with a straight line to the right. From an outside perspective, there are several possible ways this sequence could continue. Perhaps one chooses to continue straight up then left and down, or one chooses to go down then left and up to create another equally valid square, or perhaps one chooses to lift the pen and create the parallel lines first (see fig 5.2).

This example showcases that the continuation of a drawing is not necessarily deterministic. Humans perceive drawing visually; a sketch appears identical regardless of the order in which it was drawn. Hence the stroke order of a sketch is either a result of drawing style or

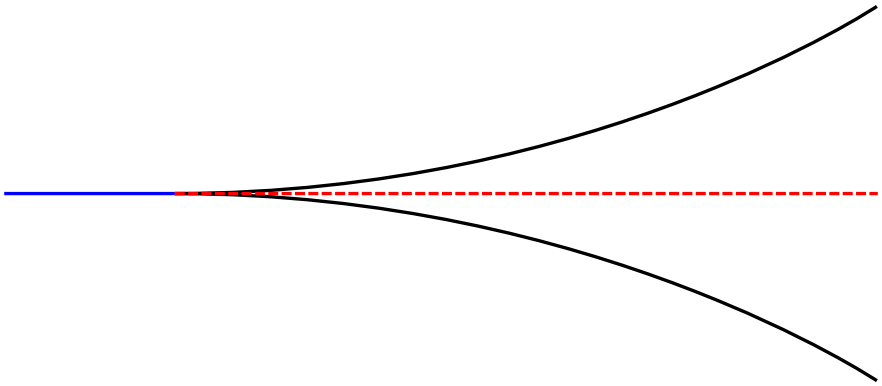

Figure 5.3: The red line minimizes the MSE loss between the two black sequences, a single-output model might converge to this line if trained on only these two sequences, while a GMM can represent both at the same time.

simply a coincidence. Further, since many classes of objects might start identically or contain similar strokes, the continuation at a given point is heavily dependent on the unknown underlying class of objects being drawn.

Efforts can be made to reduce the amount of unknown information. The underlying class being drawn could, in theory, be given by the user. Drawing style could also be learned by training the model on individual users' drawings. However there will always be some uncertainty in the data.

Mixture models can naturally encode this uncertainty (see fig. 5.2). Single-output models, on the other hand, are deterministic; they will always output the same next point given a prior sequence. Moreover, to reduce the geometric loss common in these models, the model may learn a mean stroke over the possibilities, which is probably not desired (see fig. 5.3).

5.2.3 Extending the model

We propose extending the model by modeling the local variations of the data before passing the values to the recurrent unit. By doing this, the recurrent component might avoid "wasting" capacity on modeling local relations, enabling it to focus on the long-term structure of the drawing.

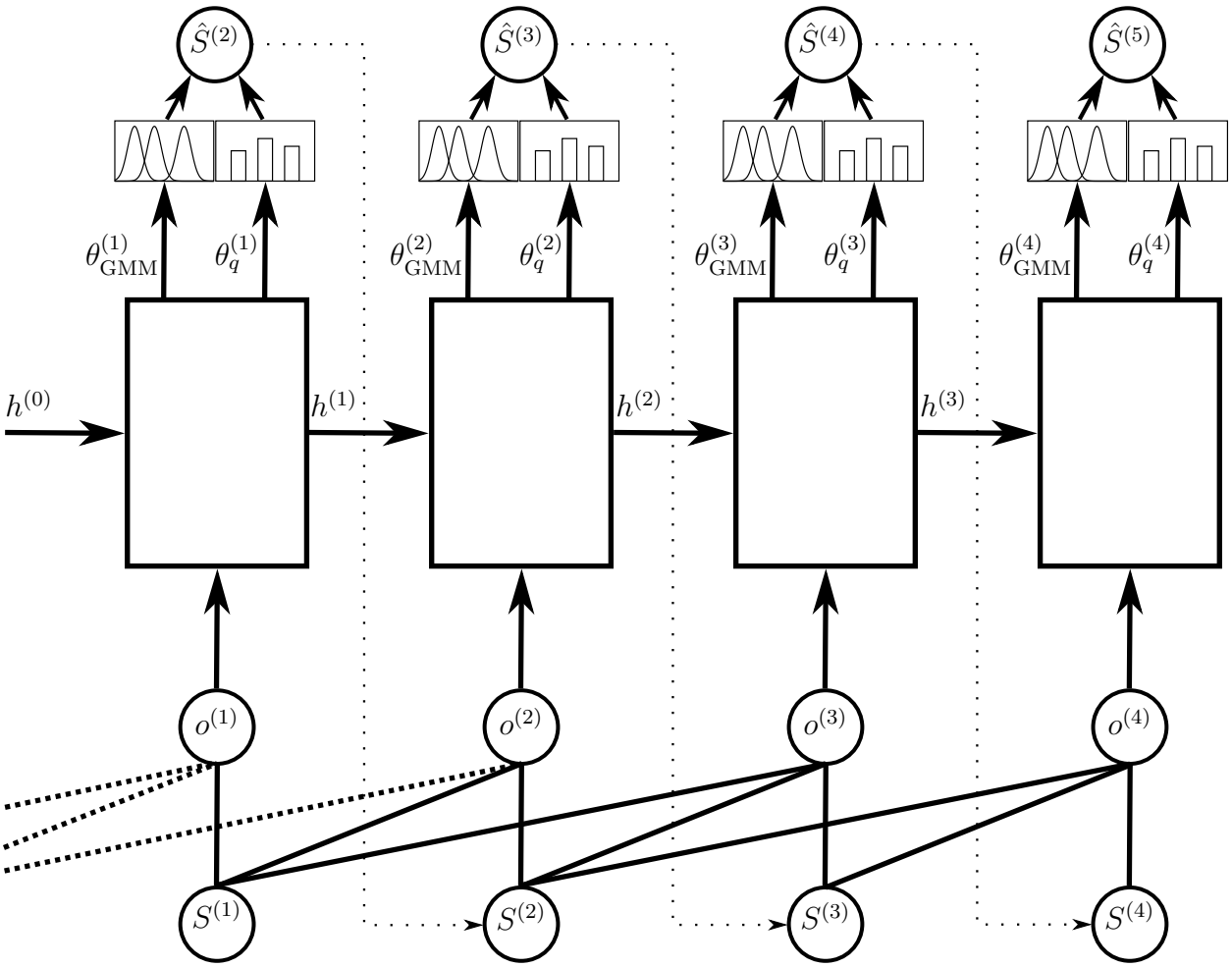

Figure 5.4: Extended model with kernel/window size $M = 3$. The intermediate values $o^{(t)}$ are computed using the M latest points in the drawing, which as represented as connected lines between the input points and the intermediate values. The thick dotted lines connected to $o^{(1)}$ and $o^{(2)}$ represent identical Start-of-Sequence input points.

To model such a local relation, a 1D convolution is applied over the input sequence. At each time step t, we compute an intermediate value $o(t)$ by using a convolutional operation:

$$o^{(t)} = K(t) = \sum_{m=0}^{M-1} S^{(t-m)} w_m, \quad w_m \in \mathbb{R}^5$$

The recurrent unit is now input the intermediate values rather than the points directly, i.e., $h^{(t)} = R(o^{(t)}, h^{(t-1)})$. If we use C_{out} kernels, then $o^{(t)} \in \mathbb{R}^{C_{\text{out}}}$, and as such the recurrent unit needs an input size of C_{out}.

Since the first position of a kernel with size M requires the values of $S_1, S_2, ..., S_M$ the first possible next point to predict is S_{M+1}, i.e., the model cannot predict the M earliest points in the drawing. We can remove this problem by padding the sequence with M start-of-sequence (SOS) points where each SOS denotes the vector $(0, 0, 1, 0, 0)$. For example, the first three points are sampled autoregressively from the model in the following manner:

$$\hat{S}^{(1)} \sim \text{model}(\text{SOS}^{(1)}, \text{SOS}^{(2)}, ..., \text{SOS}^{(M-1)}, \text{SOS}^{(M)})$$
$$\hat{S}^{(2)} \sim \text{model}(\text{SOS}^{(2)}, \text{SOS}^{(3)}, ..., \text{SOS}^{(M)}, \hat{S}^{(1)})$$
$$\hat{S}^{(3)} \sim \text{model}(\text{SOS}^{(3)}, \text{SOS}^{(4)}, ..., \hat{S}^{(1)}, \hat{S}^{(2)})$$

5.3 The encoder-decoder model

Until this point, we have only utilized the sequential nature of drawings; the spatial nature might still be a valuable unexploited aspect. To make use of such information, we propose an encoder-decoder architecture similar to the work of Cao et al. [4]. In this architecture, the encoder is tasked to process the current state of a drawing, while the decoder creates the suitable prediction.

By splitting the model into an encoder and decoder, we open up the possibility of using different architectures for each component. This thesis keeps the decoder as a recurrent model while using a CNN as the encoder. The choice is motivated by the discussion in chapter 3. Humans perceive sketches visually, while the drawing process is inherently sequential. Note that other encoder architectures are tested in the experiments to see whether this idea translates to deep learning.

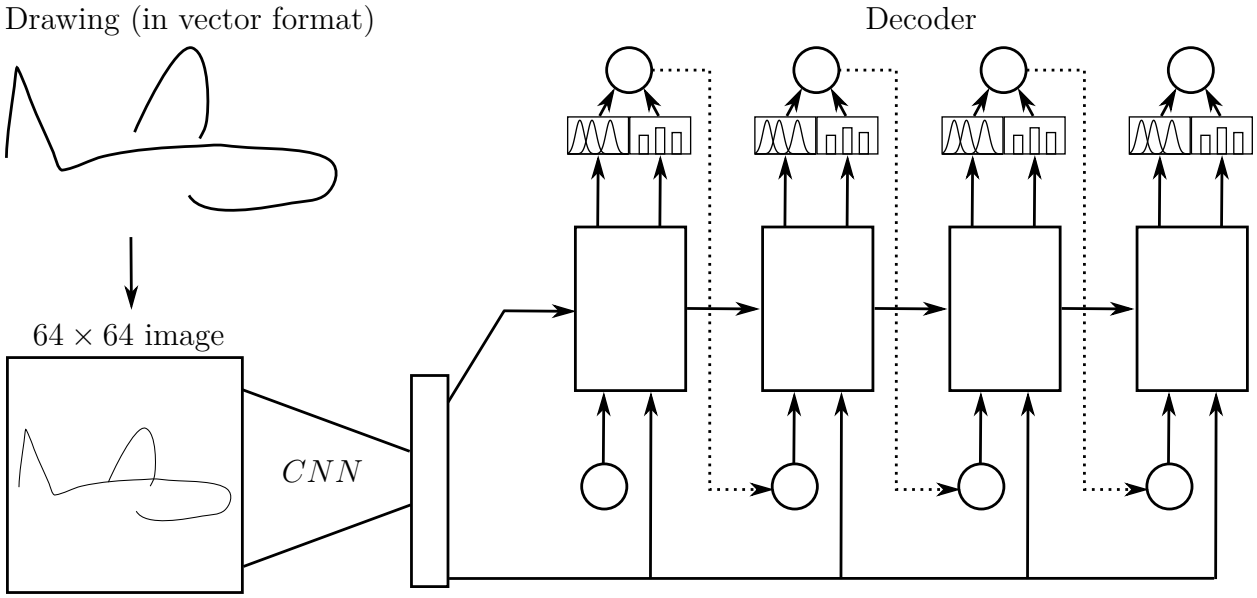

Figure 5.5: Encoder-Decoder architecture with a CNN encoder and RNN decoder, the drawing is transformed to a raster image before being passed through the CNN. The encoded image is transformed into the first hidden state of the decoder as well as being transformed into constant information that is fed to the decoder at each step. Other than the additional constant information from the encoder, the decoder works in the same fashion as described in section 5.2.

5.3.1 The architecture

The encoder is a CNN architecture consisting of 4 convolutional layers. Each consists of a convolutional operation (3×3 kernel sizes by default) followed by a 2×2 MaxPool operation with stride 2 and a ReLU non-linearity. Motivated by Yang and Hospedales [43], we use large 7×7 kernel sizes in the first-layer input to better capture the local structure in sketches. The first layer uses 32 kernels, which is doubled each following layer until the final number of output channels is 256. The output of the convolutional layers is then flattened and transformed with a linear layer into a 512 size latent space encoding of the image.

When an incomplete drawing is passed to the model, it is initially in vector format and needs to be transformed into a raster image to make it compatible with CNNs. This image is created by drawing the strokes onto a 64×64 image using a stroke size of 1. Moreover, we center the drawing on the image and ensure that the length of the drawing's longest axis covers 90% of the image. See figure 5.5 for a visualization of the process.

The encoded image is further concatenated with information not captured by the convolutional layer. This information consists of the absolute position of the pen, the pen state (lifted or down), and the smallest and largest absolute positions of points in the drawing. From these values, we create three parts using independent linear layers:

(1) The initial hidden state of the decoder. A tanh non-linearity is used to preserve the range of values expected by LSTMs.
(2) The initial cell state of the decoder.
(3) 512 values to be concatenated with each input point $S^{(i)}$. This method is inspired by how SketchRNN [17] passes information to the decoder. These values are constant information from the encoded drawing that the recurrent unit cannot overwrite. Note that as a result of this, the decoder now needs to process inputs of size $5 + 512$.

After initialization, the encoder is now ready to predict the continuation of the drawing. A Start-of-Sequence vector consisting of $512 + 5$ zeros is given as the first input to start the sampling process; following this, the decoder can work autoregressively.

5.3.2 Training the model

Since the decoder is a variant of the decoder-only model introduced in the previous section, training it follows the same procedure. Both the encoder and decoder are fully differentiable, so calculating the gradients of each component is straightforward with a modern deep learning library.

During the training process, a cutoff value must be chosen to decide how much of a drawing passed to the encoder. This cutoff value does not have to be constant throughout training; since a user probably would ask for prediction at random stages of a drawing, it may be more reasonable to sample this cutoff for each training step.

The encoded information from an incomplete drawing may become less and less valuable to the decoder as the prediction continues until the end. For example, suppose the half-complete body of a cat is given as input. Encoding this information useful for completing the body. However, when the decoder continues to create the head and facial features of the cat, then this information is essentially irrelevant. Such an information drop may cause

the decoder to ignore the values from the encoder when training the model. To prevent this effect, a decoder-length hyperparameter is introduced that decides how many points ahead in time (from the cutoff) the decoder is tasked to predict.

For example, consider a drawing consisting of 40 points, the cutoff is set to 20, and the decoder length is 10. In this case during training, the 20 first points of the drawing are passed to the encoder, and the decoder is tasked to predict the 10 following points (i.e. points 21 to 31).

Chapter 6

Experiments

In this chapter, we train the models presented in chapter 5. After training, we evaluate the models quantitatively by showing the loss metrics. Recall that the goal of the thesis is to make an artificial drawing assistant that can predict the continuations of an incomplete drawing. To rest such functionality, we visualize samples from the models at different stages of sketch completion.

6.1 Decoder-only model and 1D convolution extension

This section will experiment with the decoder-only model we introduced in the previous chapter, including models extended with a 1D convolution of different kernel sizes. Six categories from the QuickDraw dataset are selected for testing: Cat, Bicycle, Face, House, Car, and Airplane. First, we train the models on datasets consisting of one of these classes at a time; then, we increase the difficulty by making datasets consisting of three classes. Additionally, we review the results of using a pre-trained model and briefly discuss the results from omitting the recurrent unit.

6.1.1 Training setup and hyperparameters

Each dataset is split into a training, validation, and test set consisting of 60%, 20%, and 20% of the drawings. The model is fed the training set through mini-batches of size 100 (i.e., 100 drawings), and we train the model with 5000 min-batches batches for each experiment. Every 100 batches, we sample 500 drawings from the validation set to estimate the model's performance.

To qualitatively test the models on a drawing, we split it into five different stages of completion: 20%, 40%, 60%, and 80%. The models are then conditioned on each part and sampled with varying temperatures (0.2, 0.4, 0.6, 0.8, and 1.0). The models are sampled until the drawing is deemed complete (by the models) or until a maximum of 200 points. Sampling until completion might not be realistic behavior in an actual application, which is discussed in the following chapter. However, if the predictions are close to what a human would draw, then sampling until the end should result in a finished drawing (as if a human drew it).

From this perspective, a good model is then a model that "behaves" like a human when drawing. Moreover, behaviour that in unnatural and mechanic is viewed as negative, while human flaws learned by the model cannot be considered incorrect. Since we only sample the models, we cannot concretely say which model makes the best completions as some models might get more "lucky" than the others, i.e., some samples might look better than others by pure chance.

Following SketchRNN [17], we use $M = 20$ components in the GMM. We use a learning rate of $\eta = 10^{-3}$, and we reduce the value by multiplying it with $(1 - 10^{-4})$ for each mini-batch down to a minimum of 10^{-5}. The decoder has a hidden state and cell state size of $n_h = 512$. Gradients are clipped at 1.0, i.e., we prevent the norm of the gradient values from becoming larger than 1.0. Sequences of length less than 10 and more than 200 are removed as a simple way of dealing with outliers. When we use the model consisting of both a 1D convolution and an LSTM, the convolution layer contains 128 kernels; however if the convolution is used without the LSTM, it contains 512 kernels to match the output size of the LSTM.

Three kernel sizes are used to test the models extended with a one one-dimensional convolution: 5, 10, and 20. A size of 20 is only used on the convolutional-only model. For

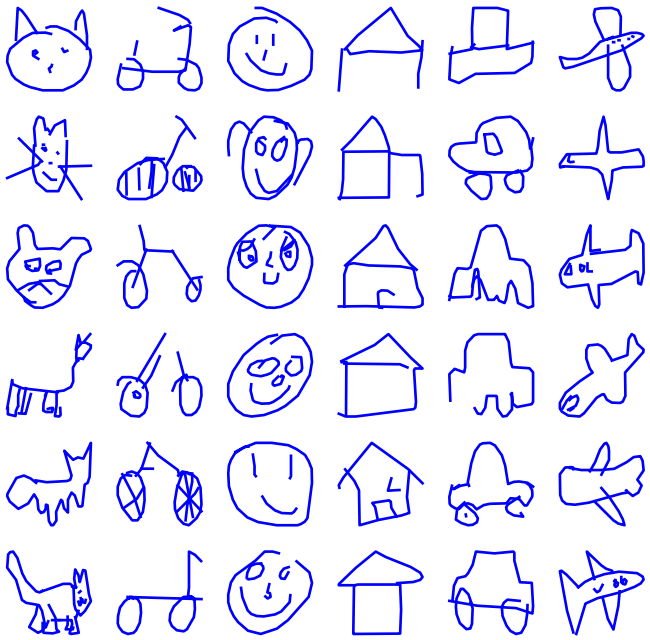

Figure 6.1: Samples of cats, bicycles, faces, houses, cars, and airplanes from the QuickDraw dataset

reference, note that the average stroke length in drawings of cats is 6.8, meaning that a kernel size of 20 covers around 3 average strokes.

6.1.2 Training on a singleton classes

Figure 6.2 shows the validation loss throughout the training process for three models. An LSTM-only model, shown as RNN, and two models extended with a 1D convolution with a kernel size of 5 and 10, shown as RNN+CNN(kernel size). Figure 6.3 shows the average loss over the test set for the same models.

Observe that the loss of the extended models stays lower than the LSTM-only model during training. The same trend can be seen in the final test loss, where the extended models reduced the loss by a noticeable amount. The difference in loss between the two

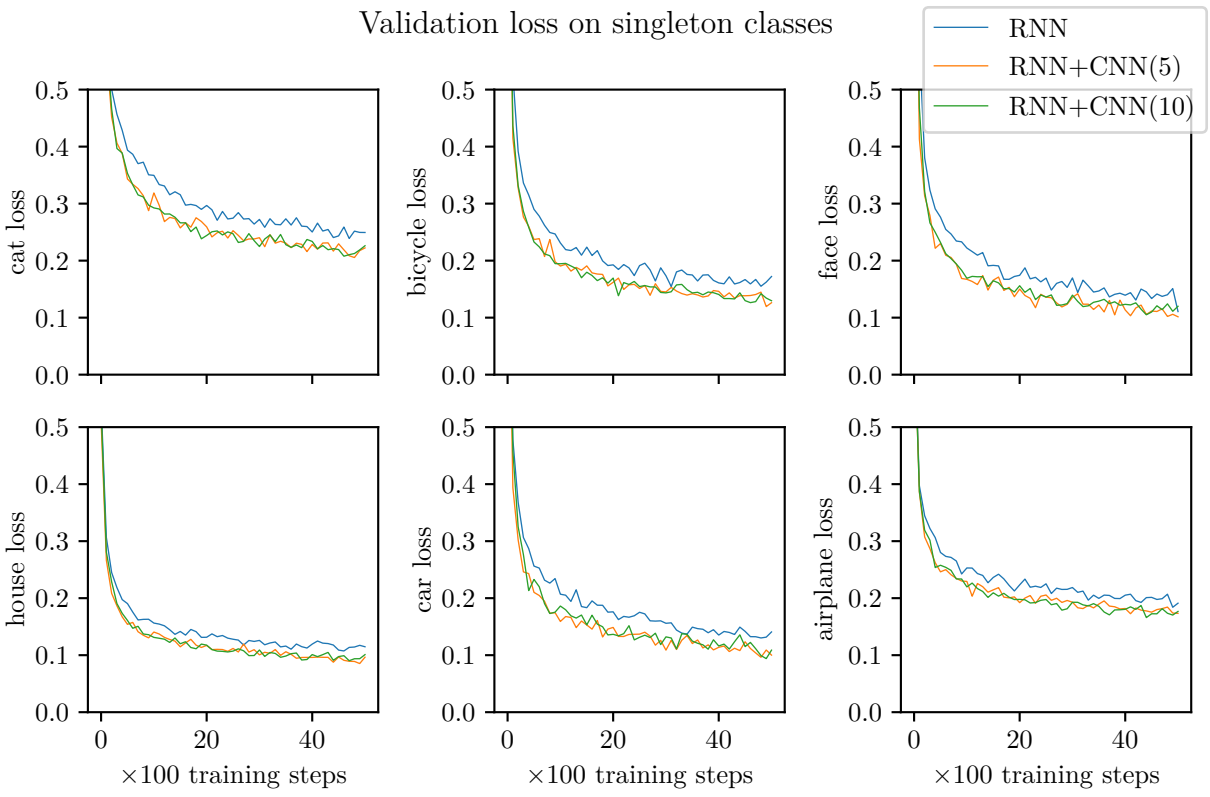

Figure 6.2: Validation loss of encoder-only model over the course of training on singleton classes.

kernel sizes is relatively insignificant. This finding might suggest that directly modeling the structure of the five latest points is helpful to the model. In comparison, information in the five points preceding these points is just as easily summarized in the hidden state.

Examples of cat sketches generated with different models are shown in 6.4. The bottom-right predictions stem from a ”pre-trained” model, discussed in the following section. All models seem to give predictions of similar quality. At lower temperatures (especially 0.2), the models often seem to get stuck in a state of repetition, especially when drawing eyes and noses. This state is referred to as the ”scribble state” by V et al. [39]. A temperature of 0.4 or 0.6 seems to give the best samples, as the higher temperature models tend to produce slightly more ”choppy” and incomplete samples. At the later stages of completion, the models seem reluctant to add missing facial features to the drawing. This reluctance might result from the high variability of drawing granularity in the dataset. For example, most cats are probably drawn with ears; however, not all cats contain finer details like whiskers.

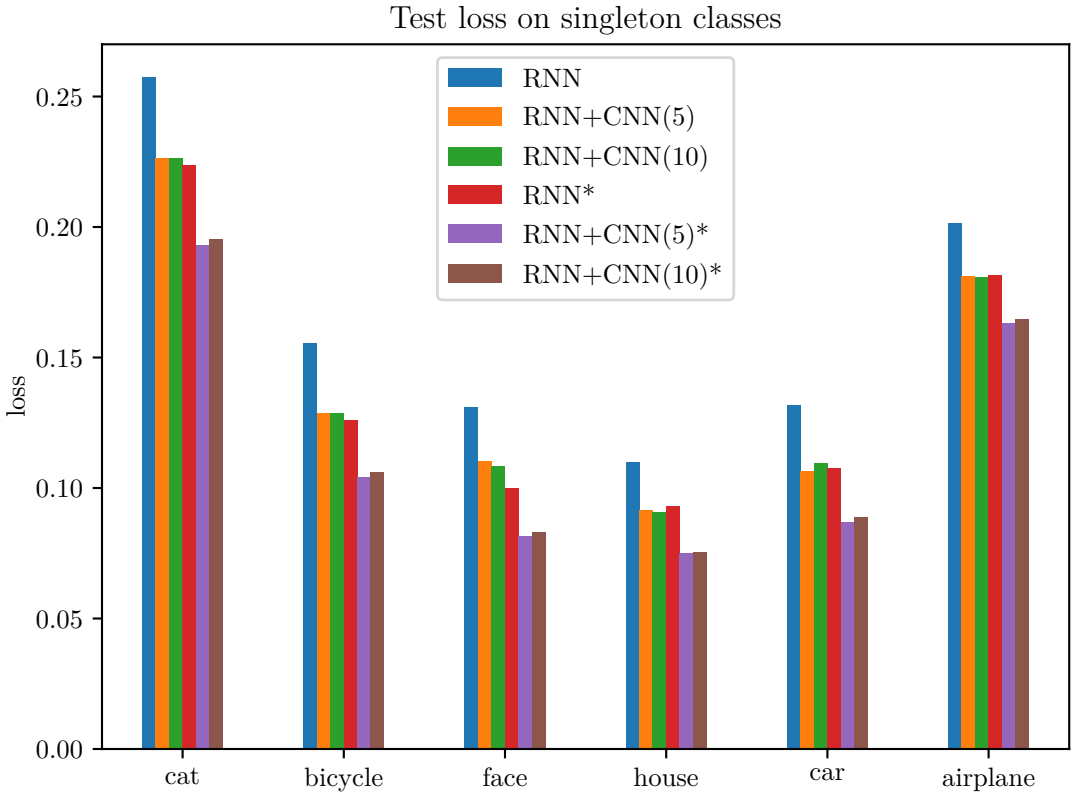

Figure 6.3: Test loss of encoder-only model on singleton classes, the star symbol ($*$) indicates the model was pre-trained.

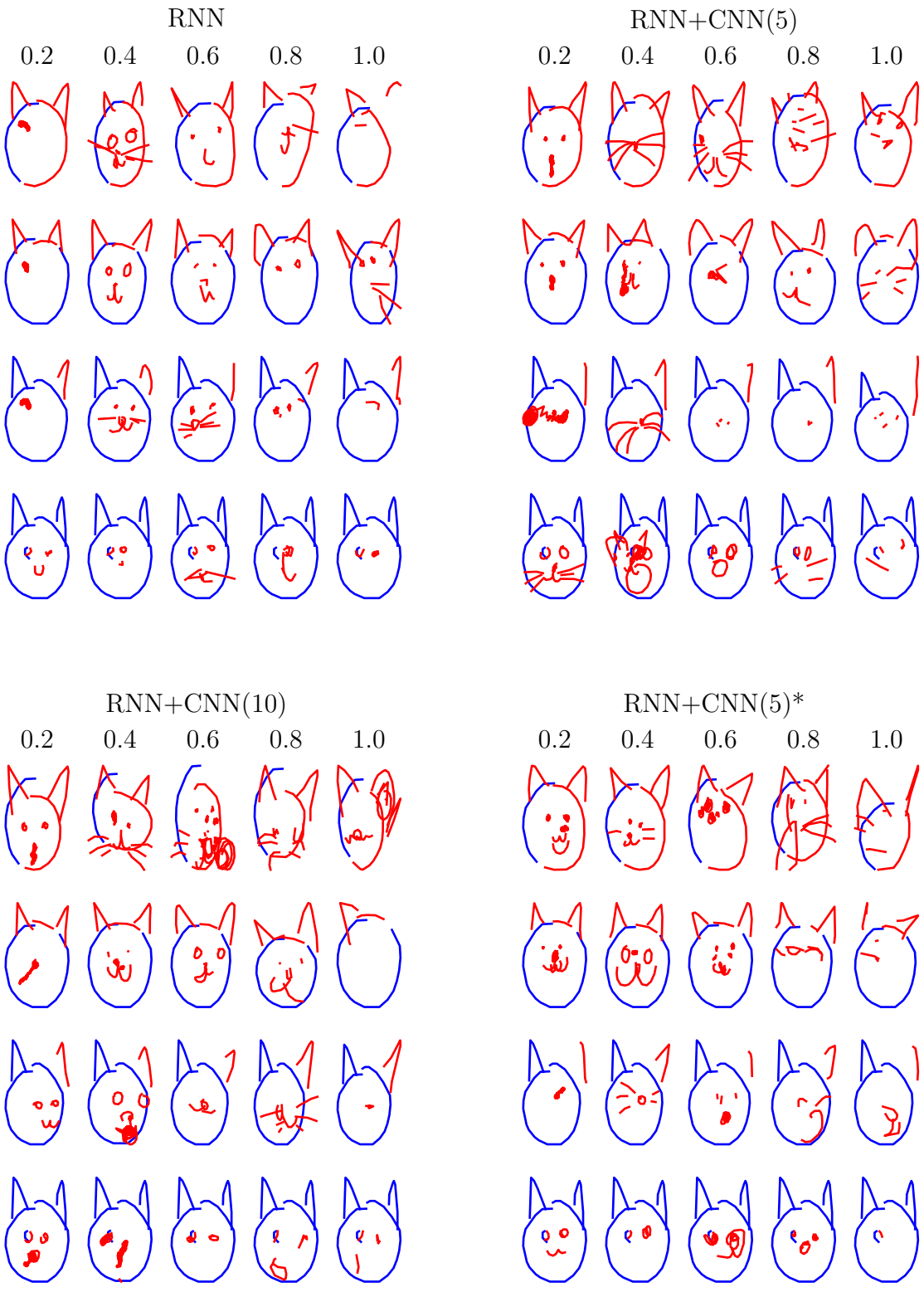

Figure 6.4: Sample completions of a cat with differing encoder-only models, the numbers above each column indicate the temperature used. The blue strokes are the "true" incomplete drawings from a human at different stages, while the red strokes are the predictions of the model.

6.1.3 Pretraining on other classes

We pre-trained models on all 339 classes not selected in the previous section before fine-tuning the model on every single class. Note that the pre-trained model was not alone capable of making good predictions, and the results of training this model are not shown. The result of the fine-tuned models are shown in figures 6.5, 6.2, and 6.4. The models converged faster than the others and reached a lower validation loss. This result was also primarily reflected in the test loss. Interestingly, the pre-trained models containing a convolutional layer were noticeably better than the LSTM-only pre-trained model, suggesting that the added capacity was practical. However, sampling these model did not lead to significantly better results.

One reason for the fast convergence speeds may be that the pre-trained model has learned general drawing patterns common between all classes. However, it could also be that some classes used to train the pre-trained model are sufficiently similar to the test classes. Even so, the fast convergence speeds suggest this method could be effective at quickly fine-tuning the model to fit a person's individual drawing style.

6.1.4 1D convolution only

We also attempted to remove the LSTM part of the model and only used the convolutional layer. In this model, the 1D convolutional layer is directly connected to the final linear layer, which transforms 512 feature maps from the convolution into the final parameters. This experiment may show whether the model can perform well using *only* highly local data.

We tested kernel sizes of 5,10, and 20, getting similar results. All configurations caused unstable training; the models had to be heavily regularized with a weight decay of 10^{-3} to prevent exploding weights. In general, the loss of these models was much higher than their LSTM counterparts, and sampling from the models gave incoherent predictions. As a result of the poor performance, we mainly omit the results from these models; however, refer to figure 6.5 to compare validation loss on cats.

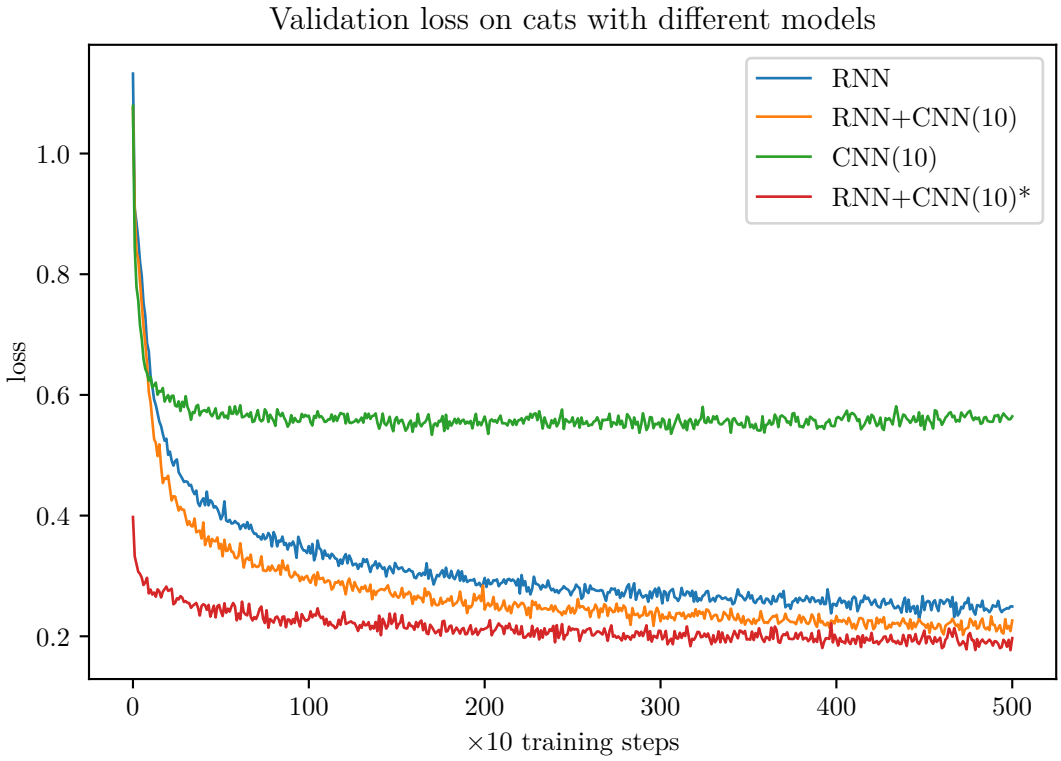

Figure 6.5: Encoder-only model performance on cats with (blue) only the LSTM, (orange) LSTM plus a CNN with a kernel size of 10, (green) only the CNN with a kernel size of 10, and finally (red) a pre-trained model with LSTM plus a CNN kernel size of 10.

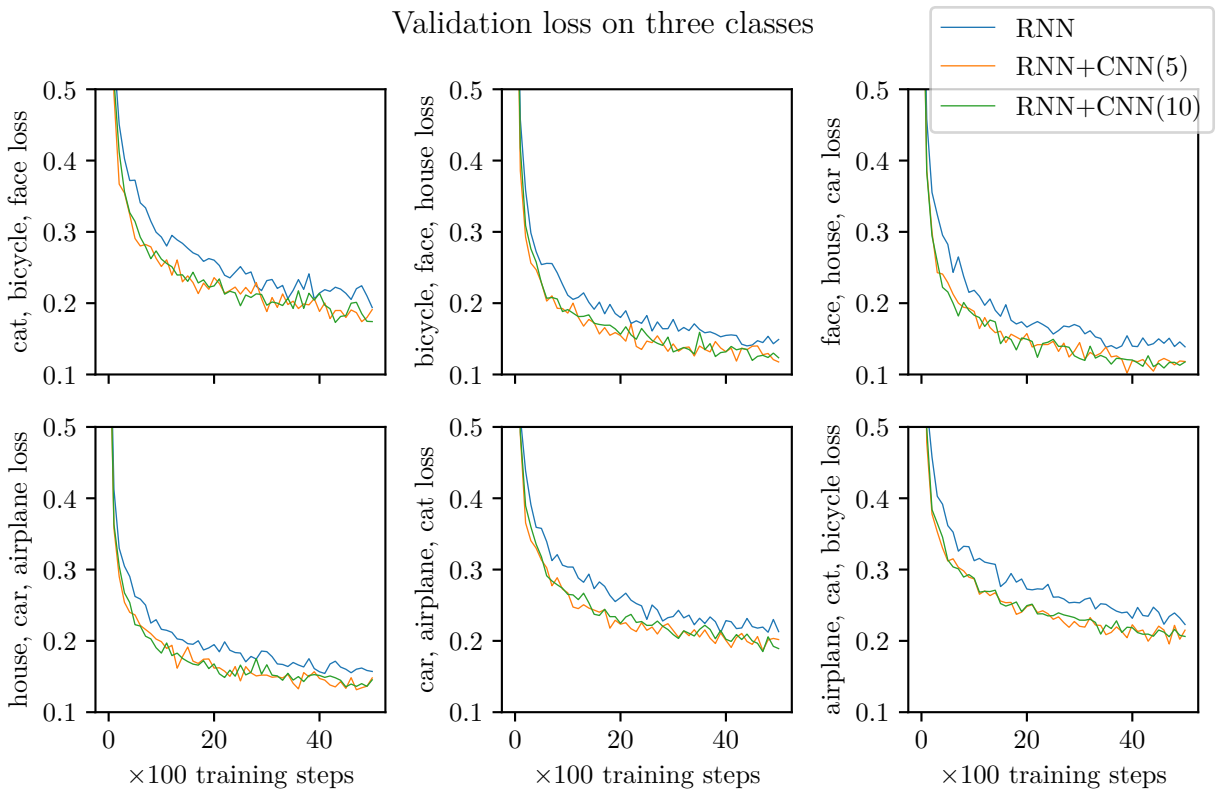

Figure 6.6: Validation loss of encoder-only model over the course of training on triplet classes.

6.1.5 Training on three classes at a time

We choose to train the model on three classes simultaneously by combining the classes chosen in the previous section. Figures 6.6 and 6.7 show the validation loss and test loss respectively. Training on several classes should be more difficult for the model as it now needs to account for inter-class variation. Even so, the model seems to converge to a loss comparable to that of using single classes. Again the models extended with a 1D convolution perform noticeably better than the LSTM-only model (w.r.t. loss).

Figure 6.8 shows three different models applied to a drawing of a car, an airplane, and a cat. Other than the earliest part of airplanes, which are sometimes mistaken for cars, the models rarely mistake what class is being completed. The classes are probably ”well separated,” i.e., the differences between how one draws each class are significant. Sampling with lower temperatures overall leads to better results. However, we can see several instances where this reduced temperature seems to cause the models to get stuck in the ”scribble state”.

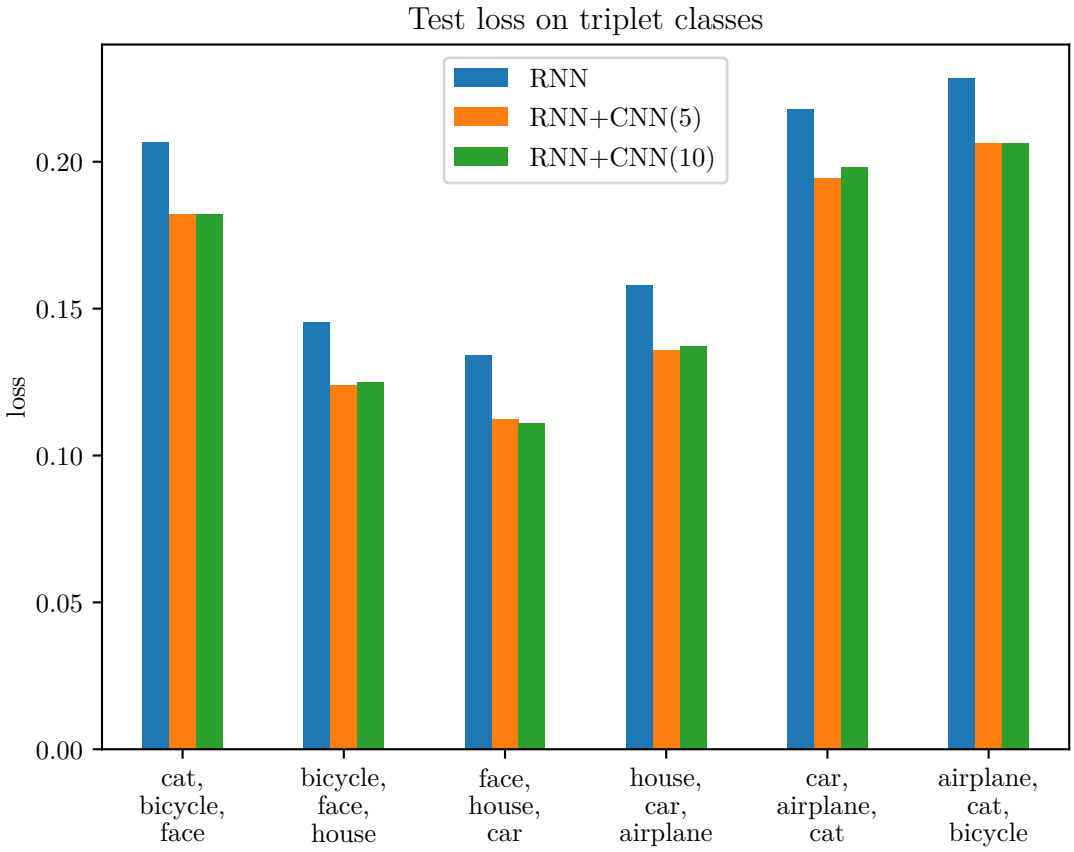

Figure 6.7: Test loss of encoder-only model on triplet classes.

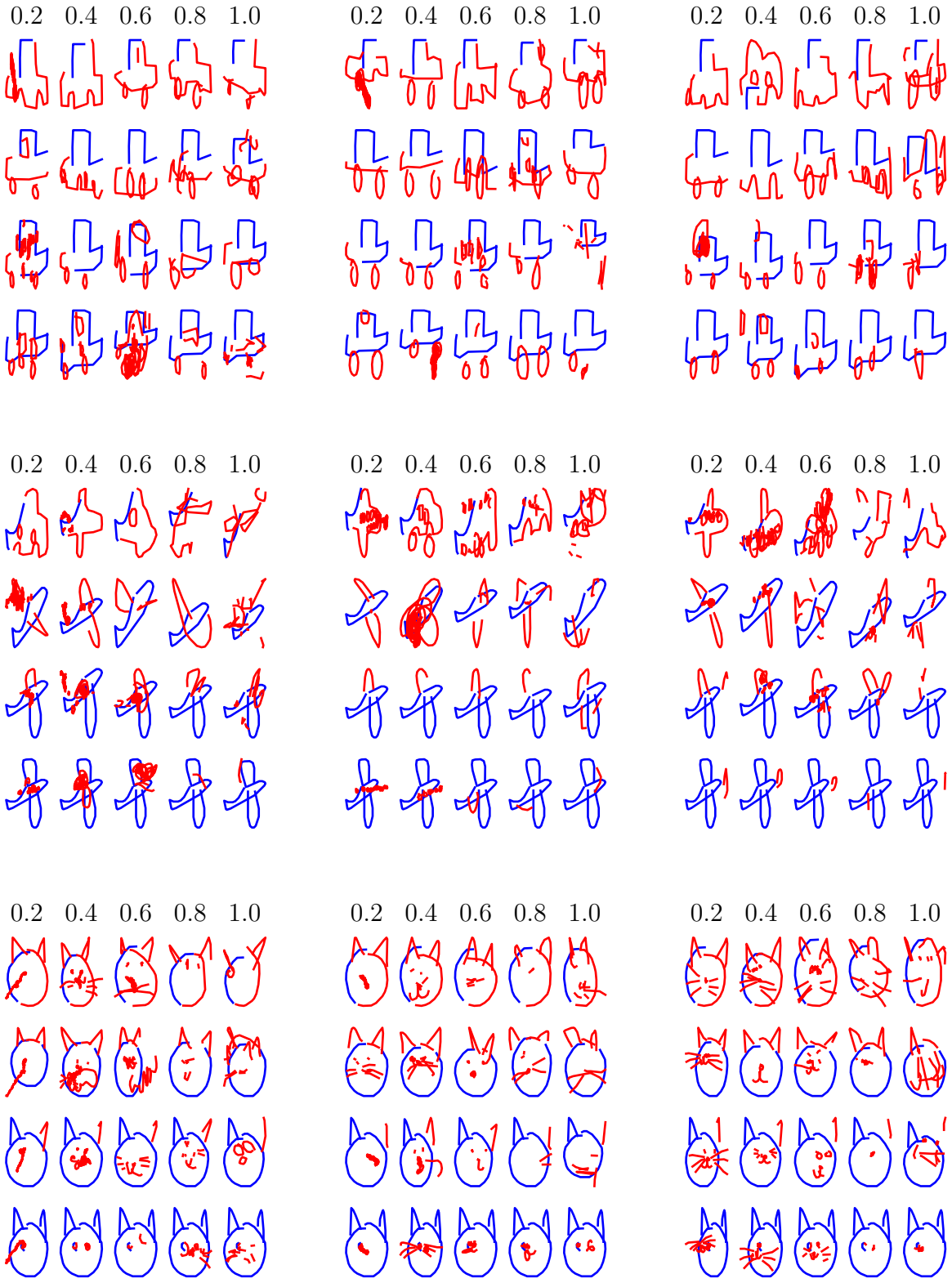

Figure 6.8: Encoder-only completions on cars, airplanes, and cats. The left column uses an LSTM-only model, the middle columns uses an LSTM model plus a CNN layer with a kernel size of 5, and the right column uses the same as the middle, only with a kernel size of 10. The blue strokes are the "true" incomplete drawings from a human at different stages, while the red strokes are the predictions of the model.

Sometimes this state is achieved when drawing the wheels of the car. However, it is most frequent in the airplane and especially present with the LSTM-only model. One thing to notice is that the models are not good at placing the windows of cars and airplanes inside the object. This error may result from a lack of spatial representations; however, it may also result from the dataset's quality.

6.2 Encoder-Decoder architecture

This section will experiment with the encoder-decoder architecture explained in the previous chapter. We use the same categories of drawing that we introduced in the previous sections. The experiments are set up in a similar fashion, where we train the models on singleton classes before extending the experiments to triplets of classes.

6.2.1 Training setup and hyperparameters

We test 4 different encoder architectures. The first architecture uses an LSTM as encoder, the second uses a CNN, and the third uses both an LSTM and a CNN simultaneously by concatenating the output from both components. The final fourth model is a baseline model that is only given some information about the current state of the drawing, i.e., the absolute coordinates of the pen, the pen state (down or up), and finally, the largest and smallest corners of the drawing.

The pen position might be an essential piece of information, and one may use this to guess what is drawn from the typical positional structure in a class. For example, if the class is cats and the pen is situated on the top-left of the drawing, then a model might complete a left ear. The baseline essentially indicates how much the decoder can learn with close to no information from the encoder. This information shows whether a more complex encoder is helpful in the model.

The model is fed the training set through mini-batches of size 100, and we train the model with 10000 training steps for each experiment. Every 100 batches, we sample 100 drawings from the validation set to estimate the model performance. Similar to the previous section,

the decoder uses $M = 20$ components in the GMM, the hidden state size is $h_n = 512$, and gradients are clipped at 1.0 for all recurrent units. We use a global learning rate of $5 \cdot 10^{-4}$, and at training steps 3000 and 6000 we reduce the learning rate by half.

Specific to this architecture, we need to set the cutoff value that decides what amount of a drawing should be processed by the encoder. To give the encoder a reasonable amount of information each step, we sample the cutoff uniformly from the first third of the drawing up to, but not including, the final point. In the limit, the encoder may be given the whole drawing except the last point; however, the average percentage of completion given to the encoder will be 66%. Even though the encoder may not receive less than a third of a drawing, it will still be given a wide variety of inputs due to the high variation of drawing lengths. For example, 30% of one cat might be 40 points while 30% of another might only be 5.

We set the cutoff to precisely 50% in the validation data to reduce "noise" when visualizing. The same procedure is done for the test loss to ensure each model is tested on the same data. Finally, the decoder length is set to 10, i.e., the decoder is tasked to predict the ten following points after the cutoff.

To test this model, we still sample the model until the end. When sampling the sequence further than 10 points, the original prior sequence is concatenated with the predictions to form a new input to the encoder, which is used to predict 10 points further, and so on. As before, if the model can make good (human-like) predictions at each cycle, then sampling several cycles should, in theory, lead to a coherent completed drawing. The cycle continues until an end of drawing value is sampled, or the maximum length is reached. Because of the decoder length, the decoder is much less likely to see an end-of-drawing sample during training. As a result, the model is less likely to predict an ending during sampling, so we set the maximum samples to 60 (rather than 200) to better visualize the predictions.

6.2.2 Single classes

Figures 6.9 and 6.10 show each model's validation and test loss metrics. Training these models causes quite large fluctuations in the validation loss; hence to make it easier to see the trend, we calculate the exponential moving average of the loss over the training steps with $\alpha = 0.95$.

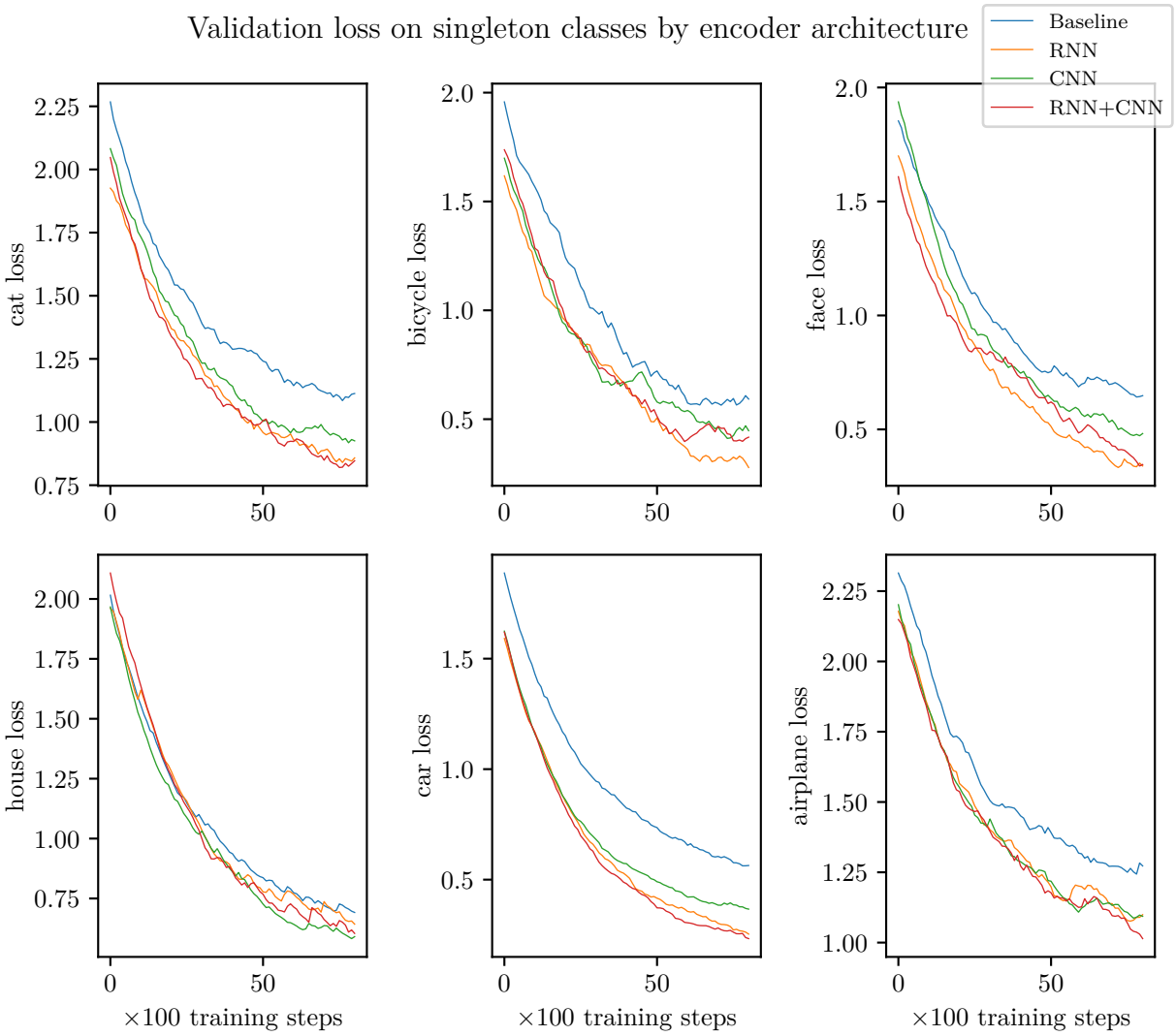

Figure 6.9: Validation loss of encoder-decoder model over the course of training on singleton classes. The displayed values are smoothed using an exponentially weighted average with $\alpha = 0.95$.

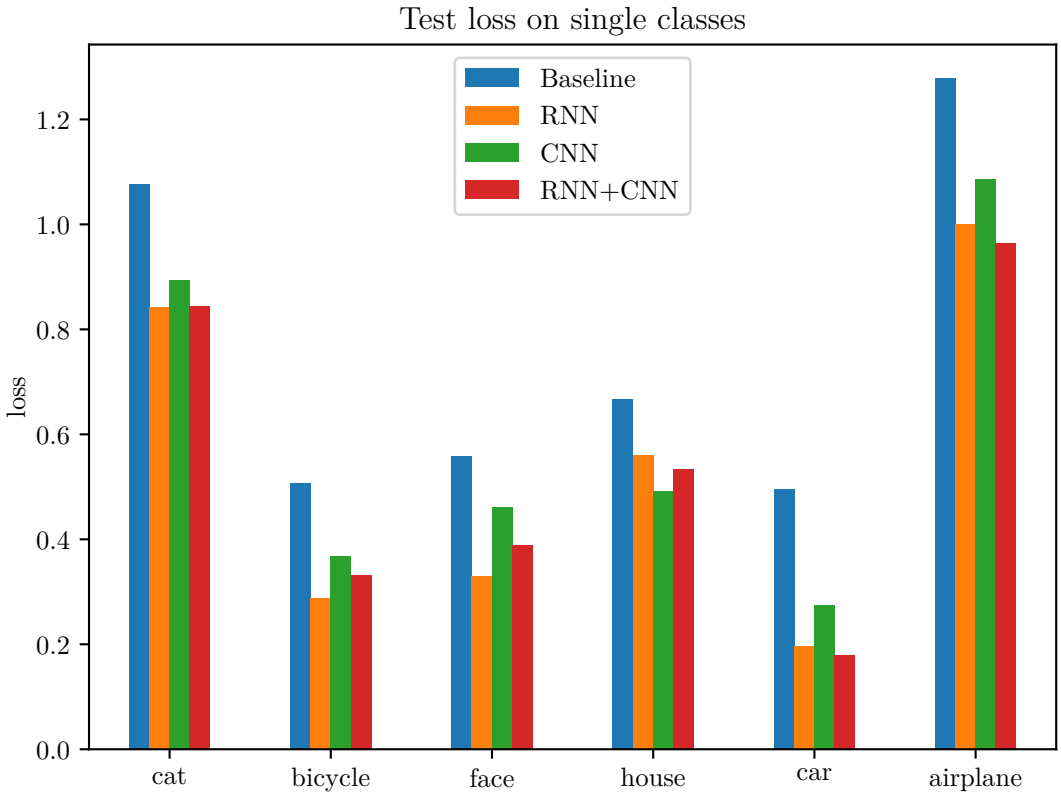

Figure 6.10: Test loss of encoder-decoder model on singleton classes.

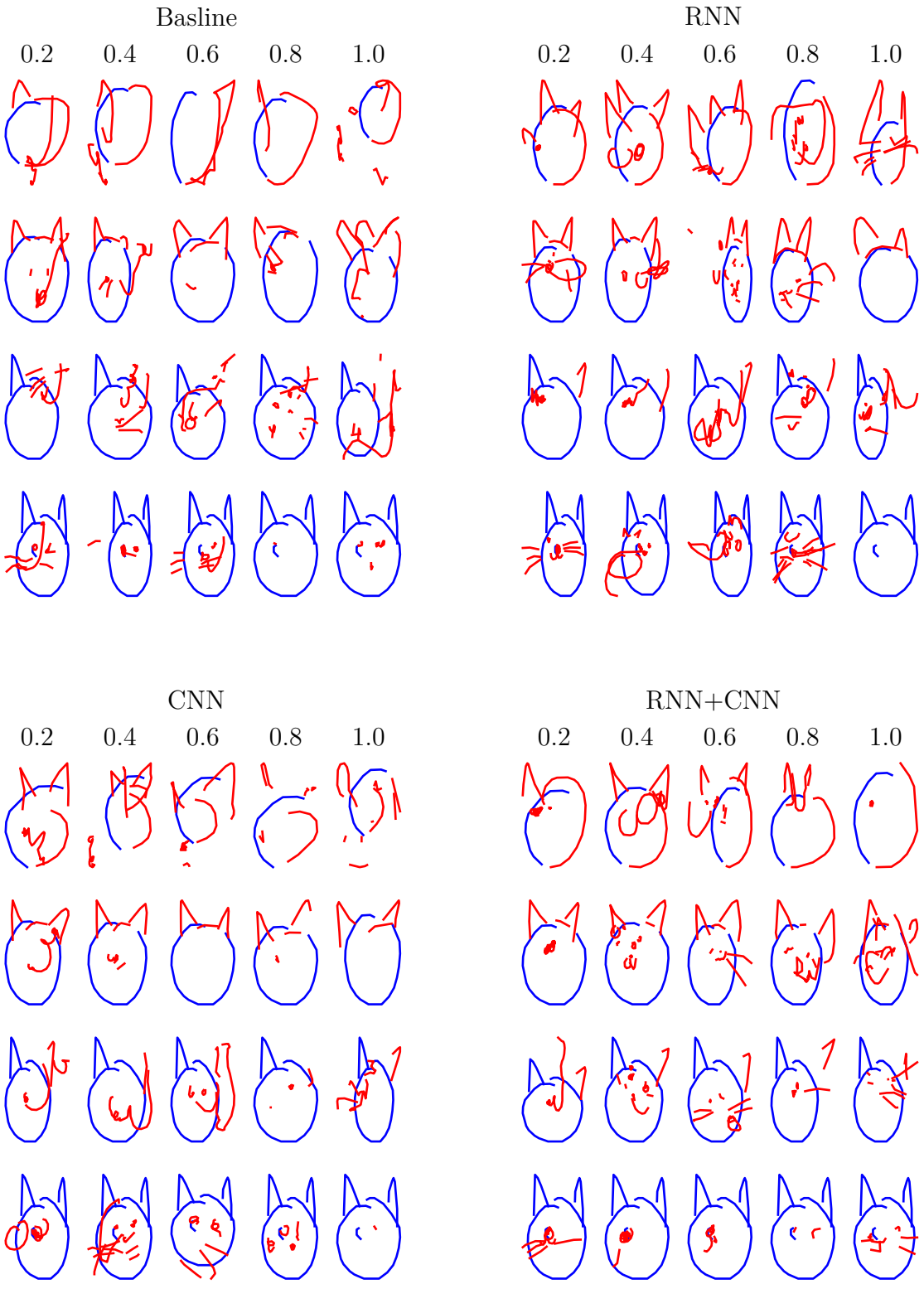

Figure 6.11: Sample completions of a cat with differing encoder-decoder models, the numbers above each column indicate the temperature used. The blue strokes are the "true" incomplete drawings from a human at different stages, while the red strokes are the predictions of the model.

The loss was generally much higher than with the encoder-only models. Theoretically, the loss should be pretty comparable if the performance is similar. The loss of the encoder-only model is the mean log-likelihood given to each point in the whole drawing, while the loss of the encoder-decoder is the mean log-likelihood given to the ten next points after the cutoff.

The loss of the baseline model generally stays higher than the rest of the models. However, the difference is insignificant, suggesting that the encoder is not as valuable as expected. The recurrent-only and dual-component models achieve approximately the same loss, while the convolution-only model performs slightly worse.

Figure 6.11 show the completions of a cat with each model. All models seem to make similar predictions. Compared to the encoder-only models, the samples are of lower quality, i.e., less human-like.

6.2.3 Triplet classes

Figures 6.12 and 6.13 show each model's validation and test loss metrics. Again, the baseline value is somewhat higher than the rest. The convolution-only encoder seems to perform closer to the recurrent models in the validation set; however, the test loss shows that the recurrent-unit models are still better. Both models containing a recurrent unit are quite close in performance, while the CNN+RNN model performed slightly better across all categories of test data.

Figure 6.11 show completions of a car, an airplane, and a cat with the baseline, recurrent, and convolutional model. The quality of the samples is lower than that of the encoder-only model, and even the non-baseline models seem to mistake some drawings of airplanes and cars for being cats. Even so, the baseline model is worse than the other models. This result is expected as the pen position cannot carry information about which class is being drawn.

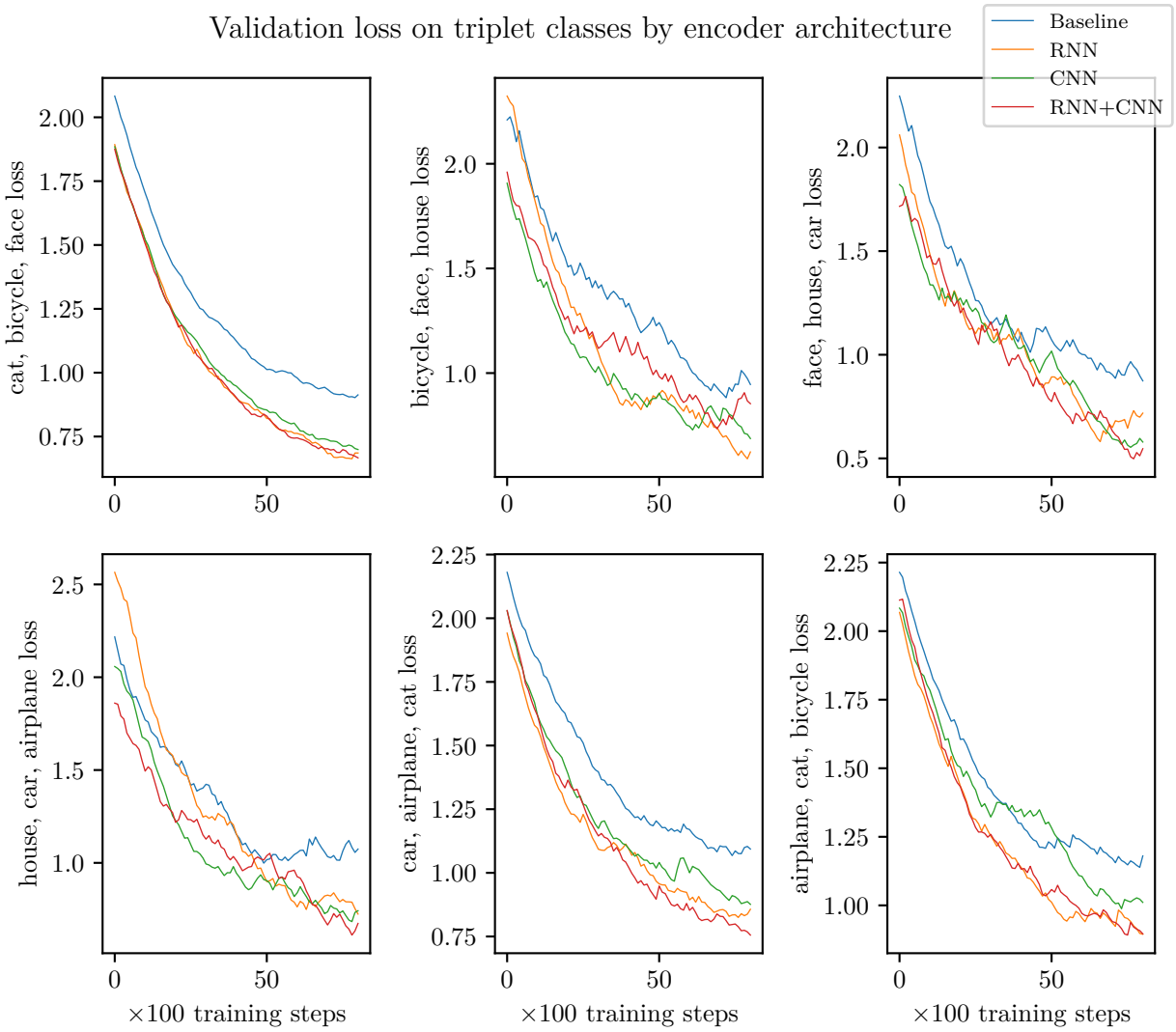

Figure 6.12: Validation loss of encoder-decoder model over the course of training on triplet classes. The displayed values are smoothed using an exponentially weighted average with $\alpha = 0.95$.

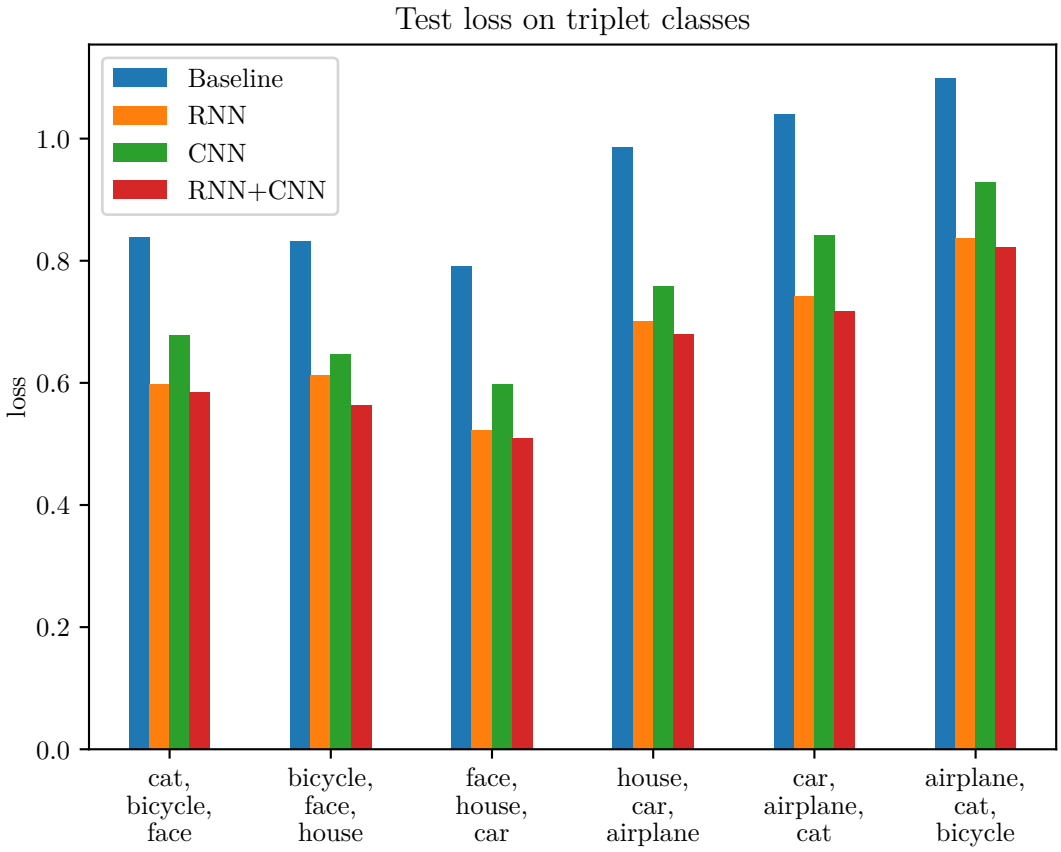

Figure 6.13: Test loss of encoder-decoder model on triplet classes.

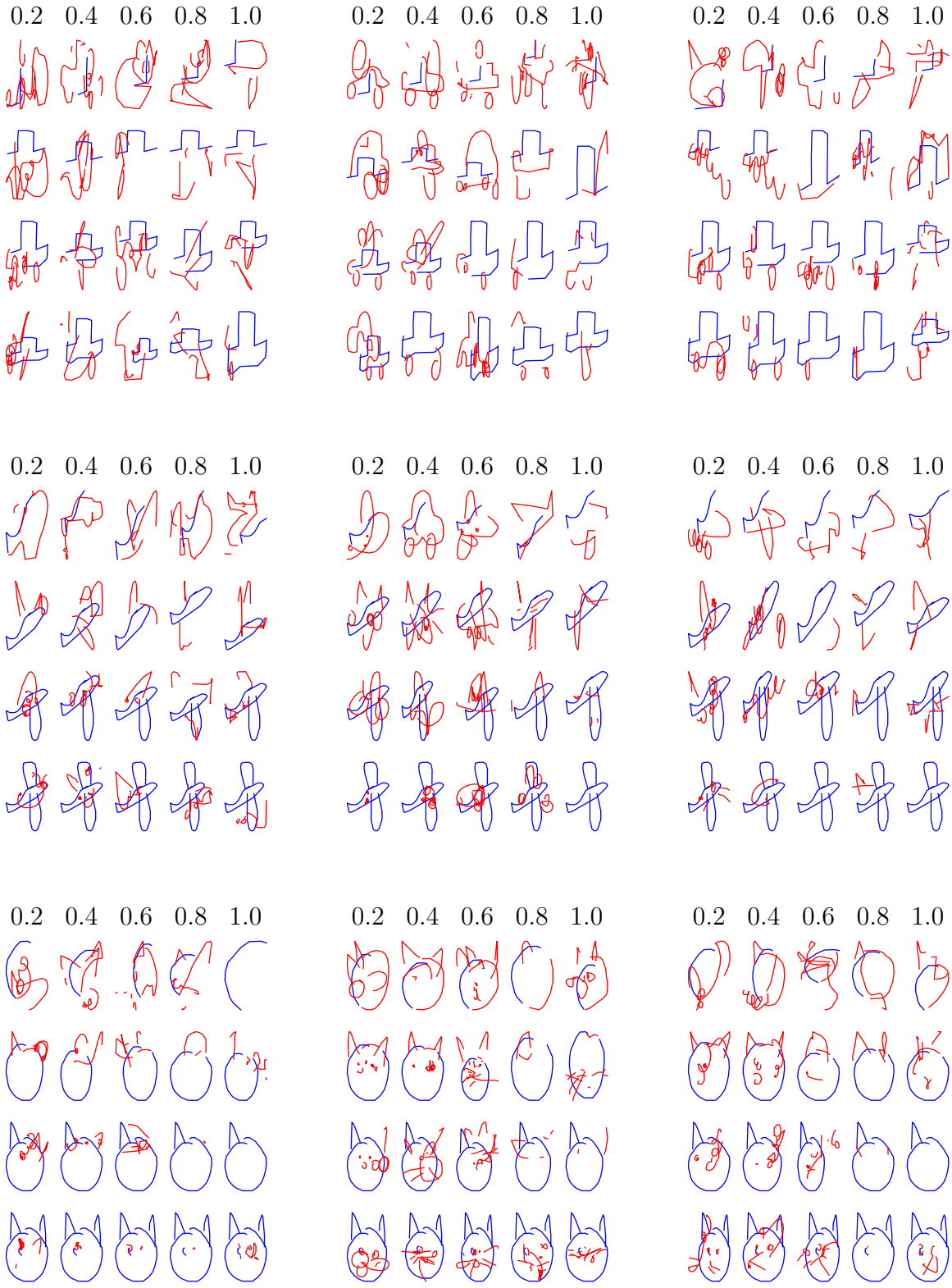

Figure 6.14: Encoder-decoder completions on cars, airplanes, and cats. Left column uses a baseline encoder, middle columns uses an LSTM encoder, and the right column uses a CNN encoder. The blue strokes are the "true" incomplete drawings from a human at different stages, while the red strokes are the predictions of the model.

6.3 Discussion and further work

While the visualizations probably give a good idea of what models perform best in a drawing assistant system, it would be even better to test each model interactively with a group of people.

Using a one-dimensional convolution reduced the loss of the encoder-only model by a noticeably amount, especially when fine-tuning a pre-trained model. However, when inspecting the samples we found no concrete improvement in quality with this method. The encoder-decoder models performed worse than the encoder-only models, and in general, the sample quality was relatively low with these models.

The fact that a purely sequential encoder performed about the same as the convolutional encoder suggests that the problem lies in the architecture or the training process instead of the spatial representation. Moreover, when both the encoder and the decoder are LSTMs, one would expect the model to have the same capabilities as the encoder-only model; however, this was not the case.

The cutoff value causes high variation in the samples given to the model; even with a fixed percentage cutoff, the variation in drawing lengths causes high variation. A constant cutoff could have been chosen; however this may make the encoder overfit on this configuration, making predictions only work at a certain point. Moreover, the usage of an encoder length attempts to make the decoder make use of the information from the encoder. However, the latest points after the cutoff are probably primarily dependent on the latest points right before the cutoff, which may make most of the information from the encoder of little use.

A different way to enable spatial representations is to use a purely convolutional model to predict the next point directly at each time step. Such a method could further be combined with the encoder-only model to enable both representations simultaneously. However, as discussed in chapter 3, predicting each time step with a convolutional model may potentially be computationally expensive.

It would be interesting to train the models on more than one or three classes, enabling them to be useful for a larger amount of drawings. Training the model on all classes was essentially attempted with the pre-trained models. However, since incomplete drawings can

look the same in several classes, such models would struggle to predict what class is being drawn, causing incoherent predictions most of the time. One way to fix this problem could be to condition the models on the class being drawn. This method would require telling the model of the class to draw. A user could do this manually; however, it may be cumbersome. Another method could be to use a separate sketch recognition system that informs the prediction model of the target class.

Efforts could also be made to increase the quality of samples. While a temperature was employed in this thesis, it could also be possible to use heuristics such as beam search [28] to create predictions the model deems probable. It would also be helpful to develop methods that reduce the occurrences of the "scribble effect" in the models.

Improvements could also be made to the data used to train the model. While it is essential to match the quality of the end-user, many drawings in the QuickDraw dataset are of lower quality as they had to be made in 20 seconds or less. Some drawings are not of the appropriate class, and some are incoherent "scribbles." A simple way to filter out "bad" drawings could be to choose only drawings with sequence lengths close to the mean of the class. A hand-crafted approach could be to analyze the properties of "good" (i.e., detailed) drawings in each class and primarily select drawings with these properties for training. Finally, it would be interesting to see how well a model could specialize to the drawings of each individual, e.g., the user's drawing style, and classes of drawings not present in the dataset.

Chapter 7

The Application

Finally, to achieve the goal we set out in the introduction, we create a drawing application capable of using the models trained in the previous section.

7.1 Sampling the model

There are several ways sampling could be integrated into the application. A model can theoretically create arbitrary long predictions (until an end-of-drawing point is sampled). If the model is not restrained, it could create unreasonably large sequences that clutter the screen. The frequency of the end-of-drawing state is low compared to the pen-lifted and pen-down states, which may cause the model to ignore it. There are methods to mitigate this problem if it is crucial to the task. In SketchRNN, for example, the task of reconstruction demands that the decoder ends the reconstruction at an appropriate time, so special efforts were made to encourage appropriate stopping. GCPS, on the other hand, avoids this challenge as it is the user's choice to end the drawing, as opposed to the model. However, an end-of-drawing token can still be viewed as helpful to encode that the model is unaccustomed to predicting beyond a certain point; hence the quality of further predictions may fall.

Even if the model gives appropriate end-of-drawing points, it does not mean the application *should* sample the model to the end. It is more reasonable to sample only a tiny continuation to make the interplay between the model and the user more natural. That way,

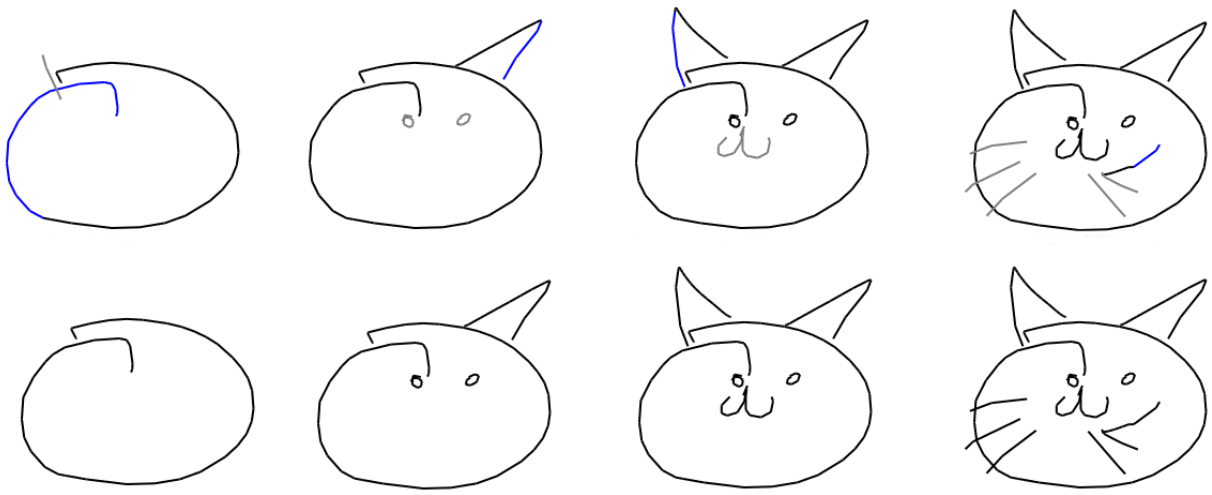

Figure 7.1: Real screenshots from drawing a cat using the application. The black strokes are from the user. The blue stroke is a prediction of the immediately following stroke, while the gray strokes are predicted strokes following the blue prediction. In the leftmost column, the user accepts only the blue stroke, while in the following column the user accepts all 20 predictions from the model

the human and the model go back and forth, where the human constantly updates the ”true” drawing, and the model uses this new information to create more relevant predictions. We can do this by restraining the sampling of the model. Two natural ways to restrain sampling are to employ a max sampling rate or to sample only the first stroke of the prediction.

7.2 Functionality

The application's default behavior is to sample 20 or fewer points from the model, regardless of how many individual strokes it creates in this length. The user is then given two possibilities: accept the first stroke in the prediction, or accept all its strokes.

A prediction is automatically sampled from the model after the user lifts the pen; this is a natural time to show predictions as the user might have lifted the pen to ponder the continuation. This behavior can optionally be turned off, in which case the user must manually request a prediction. The automatic prediction assumes that the pen was *not* lifted, which initially seems unintuitive. However, this functionality lets a user lift the pen

in the middle of a stroke to let the model complete the same stroke. Further, if the user did finish the stroke, the model can still predict that the following point should be a pen-lift.

The suggestions are visualized on the canvas as strokes of a different color than the user's. More specifically, the strokes from the user are colored black; the first stroke in a prediction is colored blue, while the following strokes are all colored gray (see fig. 7.1). A suggestion can be finalized and added to the user's drawing when desired, changing the color of the prediction to the same as the user.

Functionality to save drawings was added as an interesting further direction of training the model on drawings from a specific user. This functionality could make the model specialized on each user's drawing style.

7.3 Implementation

The application is written in the Julia [2] programming language using a GTK [38] backend. The canvas is a GtkCanvas with a grid size of 600×600, and the models, written in PyTorch [29], are loaded in Julia using the PyCall [22] library.

When a user draws, the pen position and pen state samples are continuously saved to a list of 3-element vectors, i.e., each sample is saved as (x, y, pen-lifted). Unless the pen is lifted, a line is drawn on the canvas between the previous and current sample position.

If a prediction is requested, the list of points is copied and preprocessed to match the QuickDraw training drawings. First, the coordinates are divided by the canvas height/width (i.e., 600) and multiplied by 255, then the RDP algorithm is applied with $\epsilon = 2.0$. Next, the coordinates are transformed into the representing the differences between each point and the next. Then, the differences are scaled to have a standard deviation of 1.0, while the original standard deviation is saved for rescaling the prediction. Finally, the pen state is represented as a 3-element state i.e., (p_1, p_2, p_3) as expected by the models,.

The sequence is then transformed to a PyTorch tensor and fed to the model using PyCall, which returns a 20-element prediction. This prediction is then "unscaled" (i.e., scaled back to the size of the user's drawing), shifted to the pen position, and drawn on the canvas with a blue/gray color. See figure 7.2 for a visualization of the application when prompted to predict a stroke.

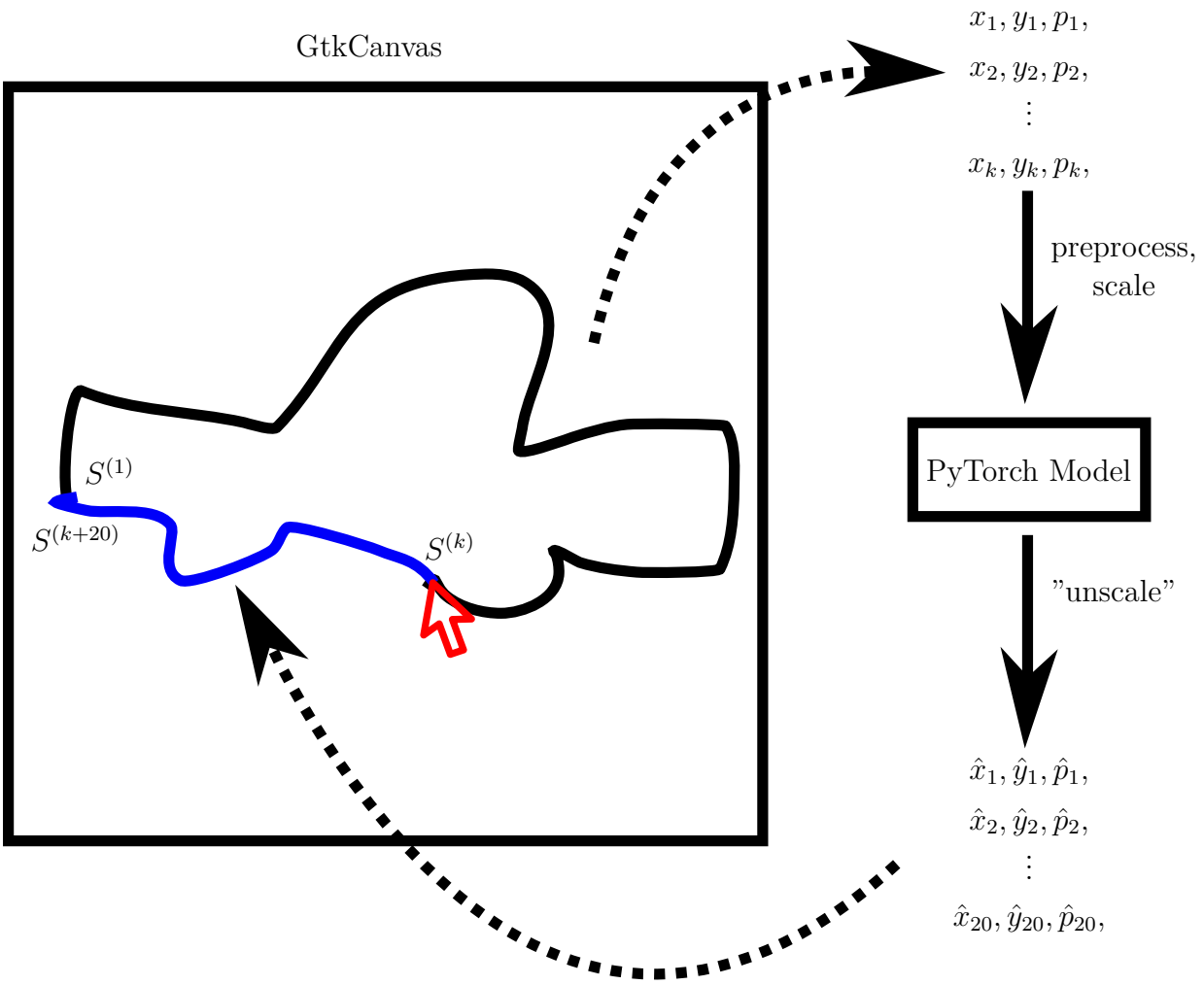

Figure 7.2: Figure of the application when a user requests a prediction. Internally, the drawing is saved as a list of points. This list is preprocessed and scaled before being passed to the PyTorch model. The output of the model is then scaled back to the canvas size and drawn on the canvas as a blue stroke.

Chapter 8

Conclusion

In this thesis, we introduced the concept of an artificial drawing assistant system, and we discussed two ways that such a system could work. Consequently, we decided to create a system capable of predicting the continuation of drawings at different stages of completion. To achieve such a system, we introduced the field of deep learning, which is capable of learning a wide variety of tasks through observing a dataset of examples.

We discussed the methods and challenges of representing a sketch on a computer and expanded the challenges to form a deep learning perspective. More specifically, we discussed two common ways to represent sketches: a list of points or a raster image. The spatial raster-image approach we theorized to be closer to how humans perceive sketches, while the sequential list-of-points representation we theorized to be closer to how humans draw sketches.

Before introducing our own set of models, we gave a thorough overview of the previous work in the sketch deep learning field. We introduced the seminal SketchRNN model, which paved the way for several models concerned with the task of sketch reconstruction. We extracted the encoder component from this model, which laid the groundwork for the models presented in this thesis. Then, we proposed a simple extension to this model by preprocessing the input with a 1D convolution layer. Motivated by how humans perceive and draw sketches, we proposed a novel encoder-decoder model which attempts to encode incomplete drawings with a CNN network before letting a recurrent decoder create predictions.

We chose six categories of drawings from the QuickDraw dataset in our experiments. First, we tried to model one class at a time before extending the challenge to modeling three classes. By sampling the models at different stages of completion of drawings, we visually estimated each model's capabilities in a drawing assistant system.

The 1D convolutional model loss was generally lower than the basic encoder-only model, and a pre-trained model resulted in an even lower loss. While the loss varied noticeably from model to model, we found no significant difference when sampling the models. Moreover, most samples were quite reasonable completions, especially with sampling temperatures of 0.4 and 0.6. However, a low temperature would sometimes lead to the model getting stuck in a scribble state.

In the encoder-decoder model, we tested several different encoder architectures: CNN, LSTM, CNN and LSTM, and finally, a baseline encoder. We found that the models containing a recurrent encoder component would perform best both on the loss and when sampling. Moreover, the baseline encoder was surprisingly close in performance to the other architectures. In general, the models' predictions were lower in quality than the encoder-only model.

Finally, we reached our goal by building a drawing application with an artificial drawing assistant system taking advantage of the models we trained in the experiments.

Bibliography

[1] Emre Aksan, Thomas Deselaers, Andrea Tagliasacchi, and Otmar Hilliges. Cose: Compositional stroke embeddings. *CoRR*, abs/2006.09930, 2020.
URL: `https://arxiv.org/abs/2006.09930`.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
URL: `https://doi.org/10.1137/141000671`.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.
URL: `https://arxiv.org/abs/2005.14165`.

[4] Nan Cao, Xin Yan, Yang Shi, and Chaoran Chen. Ai-sketcher : A deep generative model for producing high-quality sketches. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):2564–2571, Jul. 2019. doi: 10.1609/aaai.v33i01.33012564.
URL: `https://ojs.aaai.org/index.php/AAAI/article/view/4103`.

[5] Sarath Chandar, Chinnadhurai Sankar, Eugene Vorontsov, Samira Ebrahimi Kahou, and Yoshua Bengio. Towards non-saturating recurrent units for modelling long-term dependencies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3280–3287, 2019.

[6] Nigel Chapman and Jenny Chapman. *Digital Multimedia*. Wiley Publishing, 3rd edition, 2009. ISBN 0470512164.

[7] Jungwoo Choi, Heeryon Cho, Jinjoo Song, and Sang Min Yoon. Sketchhelper: Real-time stroke guidance for freehand sketch retrieval. *IEEE Transactions on Multimedia*, 21(8):2083–2092, 2019. doi: 10.1109/TMM.2019.2892301.

[8] Ayan Das, Yongxin Yang, Timothy M. Hospedales, Tao Xiang, and Yi-Zhe Song. Béziersketch: A generative model for scalable vector sketches. *CoRR*, abs/2007.02190, 2020.
URL: https://arxiv.org/abs/2007.02190.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
URL: http://arxiv.org/abs/1810.04805.

[10] DAVID H DOUGLAS and THOMAS K PEUCKER. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2): 112–122, 1973. doi: 10.3138/FM57-6770-U75U-7727.
URL: https://doi.org/10.3138/FM57-6770-U75U-7727.

[11] Mathias Eitz, James Hays, and Marc Alexa. How do humans sketch objects? *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):44:1–44:10, 2012.

[12] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. doi: https://doi.org/10.1207/s15516709cog1402_1.
URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.

[13] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *CoRR*, abs/1804.01118, 2018.
URL: http://arxiv.org/abs/1804.01118.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Google Creative Lab. Quickdraw github repository, 2022.
URL: https://github.com/googlecreativelab/quickdraw-dataset. [Online; accessed 20-May-2022].

[17] David Ha and Douglas Eck. A neural representation of sketch drawings. *CoRR*, abs/1704.03477, 2017.
URL: http://arxiv.org/abs/1704.03477.

[18] Jun-Yan He, Xiao Wu, Yu-Gang Jiang, Bo Zhao, and Qiang Peng. Sketch recognition with deep visual-sequential fusion model. In *Proceedings of the 25th ACM International Conference on Multimedia*, MM '17, page 448–456, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349062. doi: 10.1145/3123266.3123321.
URL: https://doi.org/10.1145/3123266.3123321.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[20] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.
URL: https://faculty.marshall.usc.edu/gareth-james/ISL/.

[21] Natasha Jaques, Jesse H. Engel, David Ha, Fred Bertsch, Rosalind W. Picard, and Douglas Eck. Learning via social awareness: improving sketch representations with facial feedback. *CoRR*, abs/1802.04877, 2018.
URL: http://arxiv.org/abs/1802.04877.

[22] Steven G. Johnson. Pycall.
URL: https://github.com/JuliaPy/PyCall.jl.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
URL: https://arxiv.org/abs/1412.6980.

[24] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2.
URL: https://doi.org/10.1007/3-540-49430-8_2.

[25] Hangyu Lin, Yanwei Fu, Yu-Gang Jiang, and Xiangyang Xue. Sketch-bert: Learning sketch bidirectional encoder representation from transformers by self-supervised learning of sketch gestalt. *CoRR*, abs/2005.09159, 2020.
URL: https://arxiv.org/abs/2005.09159.

[26] John F. J. Mellor, Eunbyung Park, Yaroslav Ganin, Igor Babuschkin, Tejas Kulkarni, Dan Rosenbaum, Andy Ballard, Theophane Weber, Oriol Vinyals, and S. M. Ali Eslami. Unsupervised doodling and painting with improved SPIRAL. *CoRR*, abs/1910.01007, 2019.
URL: http://arxiv.org/abs/1910.01007.

[27] Umar Riaz Muhammad, Yongxin Yang, Yi-Zhe Song, Tao Xiang, and Timothy M. Hospedales. Learning deep sketch abstraction. *CoRR*, abs/1804.04804, 2018.
URL: http://arxiv.org/abs/1804.04804.

[28] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601910.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[30] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[31] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.
URL: http://arxiv.org/abs/1910.10683.

[33] Leonardo Sampaio Ferraz Ribeiro, Tu Bui, John P. Collomosse, and Moacir Ponti. Sketchformer: Transformer-based representation for sketched structure. *CoRR*, abs/2002.10381, 2020.
URL: https://arxiv.org/abs/2002.10381.

[34] Ravi Kiran Sarvadevabhatla, Jogendra Kundu, and Radhakrishnan Venkatesh Babu. Enabling my robot to play pictionary : Recurrent neural networks for sketch recognition. *CoRR*, abs/1608.03369, 2016.
URL: http://arxiv.org/abs/1608.03369.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
URL: http://arxiv.org/abs/1409.1556.

[36] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
URL: http://incompleteideas.net/book/the-book-2nd.html.

[37] Tomasz Szandala. Review and comparison of commonly used activation functions for deep neural networks. *CoRR*, abs/2010.09458, 2020.
URL: https://arxiv.org/abs/2010.09458.

[38] The GTK Team. Gimp toolkit, gtk+.
URL: https://www.gtk.org/.

[39] Varshaneya V, S. Balasubramanian, and Vineeth N. Balasubramanian. Teaching gans to sketch in vector format. *CoRR*, abs/1904.03620, 2019.
URL: http://arxiv.org/abs/1904.03620.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[41] Peng Xu. Deep learning for free-hand sketch: A survey. *CoRR*, abs/2001.02600, 2020.
URL: http://arxiv.org/abs/2001.02600.

[42] Peng Xu, Yongye Huang, Tongtong Yuan, Kaiyue Pang, Yi-Zhe Song, Tao Xiang, Timothy M. Hospedales, Zhanyu Ma, and Jun Guo. Sketchmate: Deep hashing for million-scale human sketch retrieval. *CoRR*, abs/1804.01401, 2018.
URL: http://arxiv.org/abs/1804.01401.

[43] Yongxin Yang and Timothy M. Hospedales. Deep neural networks for sketch recognition. *CoRR*, abs/1501.07873, 2015.
URL: http://arxiv.org/abs/1501.07873.

[44] Tao Zhou, Chen Fang, Zhaowen Wang, Jimei Yang, Byungmoon Kim, Zhili Chen, Jonathan Brandt, and Demetri Terzopoulos. Learning to sketch with deep Q networks and demonstrated strokes. *CoRR*, abs/1810.05977, 2018.
URL: http://arxiv.org/abs/1810.05977.

[45] Robert Östling, Yves Scherrer, Jörg Tiedemann, Gongbo Tang, and Tommi Nieminen. The helsinki neural machine translation system, 2017.