Swarthmore College

Bachelors Thesis

Application of Neural Networks with Handwriting Samples

Authors:
Tom Wilmots
John Larkin

Advisor:
Matt Zucker

A thesis submitted in fulfillment of the requirements for the degree of Bachelors of Science

in the

Department of Engineering

May 2017

Declaration of Authorship

We, Tom Wilmots and John Larkin, declare that this thesis titled, 'Application of Neural Networks with Handwriting Samples' and the work presented in it is our own. We confirm that this work submitted for assessment is our own and is expressed in our own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Tom Wilmots

Date: 05.05.2017

"I propose to consider the question, 'Can machines think?"

Alan Turing

Abstract

We were originally inspired by a paper written by Alex Graves titled Generating Sequences With Recurrent Neural Networks. More specifically, we sought to replicate the portion that dealt with handwriting prediction.

Using the architecture of a cascading Long Short Term Memory Neural Network, we were able to get good results when prediction the next point in a sequence of handwriting. The entire system was trained on a batch size of 5, unrolled for 200 time steps and trained on sequences of (x, y, eos) offsets that, when plotted, represented handwriting. Prediction of real world values lends itself very well to probabilistic models so the output of the Neural Network was used to parameterize a Gaussian Mixture Model consisting of 3 components. This GMM would then output probabilistic predictions meaning all visualizations of results consisted of various heat maps.

Acknowledgements

We would like to profoundly thank Professor Matt Zucker for his endless passion, persistence, and patience with both authors of this paper. It cannot be overstated how fantastic Matt has been as a role model, both academically and socially, for both of us over the past four years.

We would also like to thank Alex Graves, for his brilliance behind the paper as well as his responses to our emails.

We would like to thank Studio Otoro, the Japanese design company who implemented a version of this for fun and wrote a nice writeup on their blog.

Finally, we want to thank Swarthmore College's Department of Engineering for the past four years. The friends and memories are unforgettable.

Contents

Abbreviations

ANN	**A**rtificial **N**eural **N**et
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory network
GMM	**G**aussian **M**ixture **M**odel
MDN	**M**ixture **D**ensity **N**etwork

Language of Neural Networks

batch - the number of samples that are propagated through the network at one time.

epoch - a full pass through the entire dataset.

perplexity - e^{loss}, this is a different representation of loss.

step size - the degree to which the LSTMs are unrolled for recurrent nature.

Chapter 1

Introduction to Long Short Term Memory Networks

1.1 Basic Review of Neural Networks

Artificial Neural Networks (ANNs) are large systems of connected neural units loosely modeled after the structure of the neurons and axons in the human brain. The core purpose of neural networks, or machine learning, is often framed as a function approximation given training data. The goal is to fit or learn a function f(x) such that y(t) is approximately equal to f(x(t)) for all values of t.

Neural networks are extremely useful in programming problem that are hard to code and in recent years have grown in popularity across many fields. They are successfully used to solve problems in robotics, computer vision, speech recognition, classification and many more.

1.2 Variations of Neural Networks

The core structure of a Neural Network consists of connections between all the neurons each carrying an activation signal of varying strength. If the incoming signal to a neuron is strong enough then the signal is permeated through the next stages of the network.

A simple neural network structure can be seen in figure 1.1. The red layer of neurons, or nodes, acts as the input layer feeding the data into the hidden layer, whose outputs are then fed into an output layer. Every connection between nodes carries a weight determining the amount of information that gets passed through.

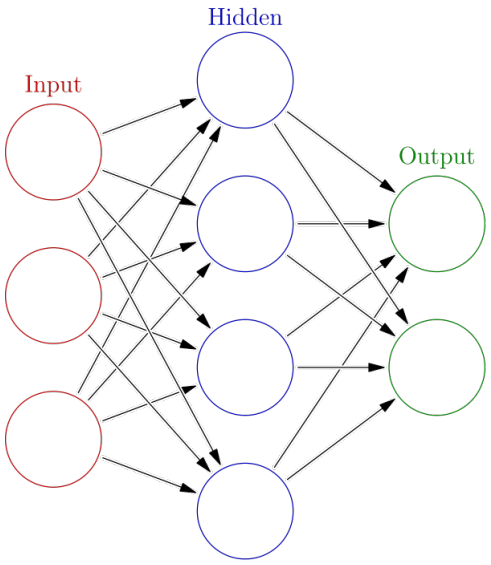

FIGURE 1.1: Basic Neural Network schematic

Neural Networks as a class are typically defined by the three following hyper parameters

- The pattern of connections between all neurons
- The weights of connections between neurons
- The activation functions of the neurons

The above parameters allow for a lot of variation and creativity based on parameters of the problem being tackled. During our project we only tackled feedforward and Recurrent Neural Networks (RNN) as well as a specific variation of RNN called Long Short Term Memory units (LSTMs). For those interested in exploring more variations, the Asimov Institute has an excellent blog post about the Zoo of Neural Networks.

1.2.1 Feedforward Neural Networks

Feedforward neural networks channel information or data in one direction. The structure shown in figure 1.1 is that of a feedforward neural network as the connections do not allow for the same input data to be seen multiple times by the same node. Because of this, these structures are often used for problems in classification that map raw data to categories. For example, the input data might be an image that the neural network could classify as face or not a face based on the inherent patterns within the raw data

that constitute face or not a face. The way that these decisions are made is through feedforward computation.

Every node outputs a numerical value that it then passes to all its successor nodes. This numerical output is

$$y_j = f(x_j) \tag{1.1}$$

where

$$x_j = \sum_{i \in P_j} w_{ij} y_i \tag{1.2}$$

and $f(x)$ is a smooth non-linear activation function that maps outputs to a reasonable domain. What activation function is used is up the designers discretion. Some common activation functions include $tanh(x)$ or the sigmoid function.

In order to train these networks we propagate the error back through the network starting at the output node, and minimize using stochastic gradient descent. There are many various definitions of error, as you will see later in the paper, but for now let us define our error simply as the sum of squared residuals between our k targets and the output of k output nodes of the network

$$E = \frac{1}{2} \sum_{k} (t_k - y_k)^2 \tag{1.3}$$

We then generate the set of all gradients of the weights with respect to the error and minimize these gradients

$$g_{ij} = -\frac{\delta E}{\delta w_{ij}} \tag{1.4}$$

Essentially, we are altering our weights in order to minimize their individual effect on the overall error of the outputs.

The largest hindrance of these types of neural networks is that they have no capacity to remember. This means they fall short when trying to predict sequences. For example, when trying to predict the last word in a sentence it is necessary to remember the previous words in the sentence. Context is necessary to make an accurate and relevant prediction. This is where Recurrent Neural Networks come in.

1.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs), unlike feedforward networks, have a capacity to remember. This memory stems from the fact their input is not only the current input vector but also a variation of what they output at previous time steps.

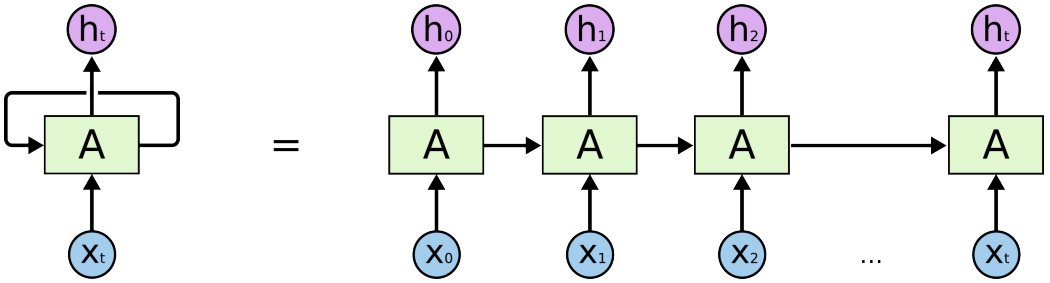

FIGURE 1.2: RNN Unrolling

In figure 1.2 we see a RNN module that is being unrolled for multiple time steps. Within each time step we see a connection between modules that is passing the information stored within the module at time step t, to the module in time step t+1. Once again it is this passing of information that provides RNNs with their memory and gives the network context.

An ideal RNN would theoretically be able to remember as far back as was necessary in order to to make an accurate prediction. However, as with many things, the theory does not carry over to reality. RNNs have trouble learning long term dependencies due to the vanishing gradient problem. An example of such a long term dependency might be if we are trying to predict the last word in the following sentence *"My family originally comes from Belgium so my native language is PREDICTION".* A normal RNN would possibly be able to recognize that the prediction should be a language but it would need the earlier context of Belgium to be able to accurately predict *DUTCH.*

We train RNNs through the same method of backpropagation outlined above, with the added wrinkle of doing it through time as well. The information that passes through RNNs undergo a lot of multiplactions and manipulations as it traverses through many layers. So when we try to propagate the error through these many layers we begin to push the derivaties, or gradients, to zero. These gradients can become so small that they cause underflow, and the networks are unable to effectively learn.

Fortunately, in 1997, Sepp Hochreiter and Juergen Schmidhuber developed the Long Short Term Memory (LSTM) unit that solved the vanishing gradient problem.

1.2.3 LSTMs

LSTMs are specifically designed to learn long term dependencies and have no problem doing so. Every form of RNN has repeating modules that pass information across timesteps, and LSTMs are no different. Where they differ is the inner structure of each module. A standard RNN might have a single neural network layer, but LSTMs have four.

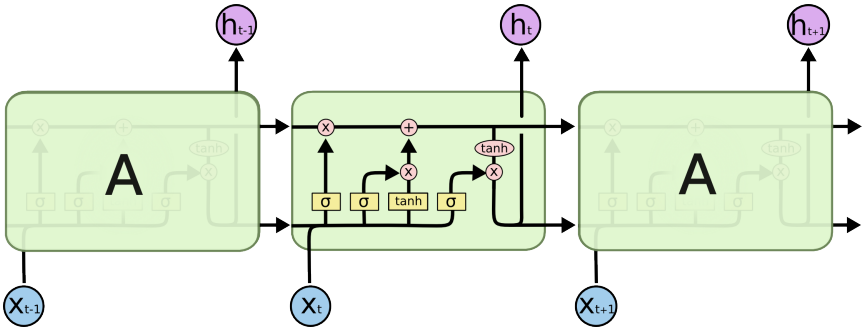

FIGURE 1.3: LSTM Module Structure

In order to understand how LSTMs learn long term dependencies let us walk through figure 1.3 step by step. The top line is key to the LSTM's ability to remember and it is called the cell state C_t.

The first neural network layer is a sigmoid and takes as input the concatenation between the current input x_t and the output of the previous module h_{t-1}. This is coined the forget gate as it determines what to forget in the current cell state. We piecewise multiply the output of the sigmoid layer, $f_t = \sigma(W_f \bullet [h_{t-1}, x_t] + b_f)$, with the cell state from the previous module C_{t-1}, forgetting the things that it deems no longer important.

The following two neural network layers constitute the update gate. First, $x_t \bullet h_{t-1}$ is pushed through both a sigmoid layer and a *tanh* layer. The output of the sigmoid layer, $i_t = \sigma(W_i \bullet [h_{t-1}, x_t] + b_i)$, determines which values to use to update, and the output of the *tanh* layer, $\hat{C}_t = \sigma(W_C \bullet [h_{t-1}, x_t] + b_C)$, proposes an entirely new cell state. These two results are piecewise multiplied and added to the current cell state (which we just edited using the forget layer) outputting the new cell state C_t.

The final neural network layer is called the output gate, as it determines the relevant portion of the cell state to output as h_t. Once again we feed $x_t \bullet h_{t-1}$ through a sigmoid layer whose output, $o_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o)$, we piecewise multiply with $tanh(C_t)$. The result of this multiplication determines the output of the LSTM module. Note that the purple *tanh* is not a neural network layer but a piecewise multiplication intended to push the current cell state into a reasonable domain.

Chapter 2

Background of Handwriting Prediction

2.1 Purpose

This section seeks to cover some of the pertinent theory behind our project. Many statistical models and deep learning modules are used throughout our project. Because of the lack of knowledge we both had when initially starting the project, we hope to carefully demonstrate and explain the theory behind such objects, so as to give the readers a clear view of how each piece of the project connects and the various algorithms that are going on behind the scenes.

Our idea with this paper is to demonstrate to someone with little deep learning or statistical background how this model works. While Graves leaves much of the implementation details up to the reader, we will hope to explicitly illustrate our design process and the various challenges that we overcame.

2.2 Project Overview and Interest

The initial idea of this project came from a paper written by renowned computer scientist, Alex Graves, who was obtaining his postdoc under Geoffrey Hinton at the University of Toronto when he published this paper. Alex Graves is now a research scientist at Google's DeepMind. His idea with the paper, *Generating Sequences with Recurrent Neural Networks*, was to learn the general patterns of some noticeable sequences probabilistically, and then be able to sample from the learned probabilistic structure, hence generating realistic sequences that were learned from. Our goals could be summarized

more simply than the generative portion of Graves's model. We wanted to build a model that reflected Graves's model and effectively predicted pen points. This project's scope was vast - including learning the theory behind basic machine learning and neural networks, probabilistic models, and a concrete grasp over the different dimensionality and spaces that we were working with.

In more detail, Alex Graves's paper, *Generating Sequences with Recurrent Neural Networks*, basically walks through the general applicability of Long-Short-Term-Memory networks. LSTMs are an effective solution to some of the basic issues that recurrent neural networks face, as described above in section 1.3. Graves then introduces his overall model, which is a type of mixture density network that predicts probabilistically based on the outputs of an LSTM cascade. With his constructed model, Graves then walks through several applications. Namely, text prediction using Wikipedia html as a basic input, handwriting prediction, and handwriting synthesis. The vast majority of our thesis will focus on handwriting prediction, and the ability to generate realistic looking handwriting.

2.3 Theory of Model Components

As this was our first introduction to deep learning and some of the various probabilistic models that Graves used casually in his paper, time was spent to fully understand and construct each of the modules used in Graves overall LSTM cascade model. In this section, the theory and basic abilities of each model will be explained more thoroughly.

2.3.1 Gaussian Mixture Models

Gaussian mixture models are an unsupervised technique for learning a probabilistic distribution of some unlabeled data. The concept of GMMs were formalized in 1977 by Arthur Dempster, Nan Laird, and Donald Rubin [Wikipedia, 2017b].

A few things about GMMs are important to note here:

- The data is not sequential
- The method is unsupervised
- There is no prediction here

At a high level, what happens with this algorithm is that a number of Gaussians are specified by the user, and the algorithm learns various parameters that represent

the data while maximizing the likelihood of seeing such data. If we have k components, then for a *multivariate* Gaussian mixture model, we will learn k means, k variances, k mixture weights, k correlations through expectation maximization.

Expectation maximization (EM) is at the heart of the algorithm. This process drives us to iteratively from our first randomly initialized means to the statistical best values for the means. In essence, it computes the probability for each point of being generated by each Gaussian component. The EM algorithm then tweaks our means, variances, mixture weights, and correlations so that the maximum likelihood is reached. The E-step of the EM algorithm stands for the expectation step, which is creating this log likelihood function that estimates the probabilities for the given parameters. The M-step is the maximization step, which includes taking derivatives of the likelihood function with respect to all known values. The implementation details will be discussed in Chapter 3. The key summary is that for given data, the means, variances, mixture weights and correlations are learned. This therefore lets you probabilistically represent your data with a discrete few parameters - in essence, estimating the underlying probability distribution of your data.

2.3.2 Mixture Density Networks

Mixture Density Networks (MDNs) are an extension of GMMs. The concept was developed in 1994 by Microsoft researcher Christopher Bishop in his paper, *Mixture Density Networks*. The idea is relatively simple - we take the output from a neural network and parametrize the learned parameters of the GMM. The result is that we can infer probabilistic prediction from our learned parameters. If our neural network is reasonably predicting where the next point might be, the GMM will then learn probabilistic parameters that model the distribution of the next point. This is different in a few key aspects. Namely, we now have target values because our data is sequential. Therefore, when we feed in our targets, we minimize the log likelihood based on those expectations, thus altering the GMM portion of the model to learn the predicted values.

The process of a mixture density network can be represented by the following figure:

There are specific activation functions that we force our learned network parameters through so that the results are reasonably accurate. Below we show out the output from the neural network parametrizes our vectors y_t.

$$y_t = (e_t, \{\pi_t^j, \mu_t^j, \sigma_t^j, \rho_t^j\}_{j=1}^M) \tag{2.1}$$

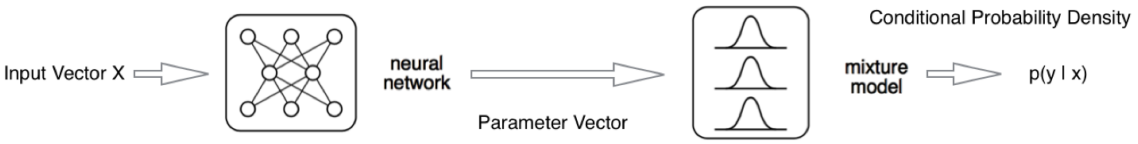

FIGURE 2.1: Process flow diagram for a MDN

$$e_t = \frac{1}{1+\exp(\hat{e}_t)} \qquad \Rightarrow e_t \in (0,1) \tag{2.2}$$

$$\pi_t^j = \frac{\exp(\hat{\pi}_t^j)}{\sum_{j'=1}^{M}\exp(\hat{\pi}_t^{j'})} \qquad \Rightarrow \pi_t^j \in (0,1), \quad \Sigma_j \pi_t^j = 1 \tag{2.3}$$

$$\mu_t^j = \hat{\mu}_j^t \qquad \Rightarrow \mu_t^j \in \mathbb{R} \tag{2.4}$$

$$\sigma_t^j = \exp(\hat{\sigma}_j^t) \qquad \Rightarrow \sigma_t^j > 0 \tag{2.5}$$

$$\rho_t^j = \tanh(\hat{\rho}_t^j) \qquad \Rightarrow \rho_t^j \in (-1,1) \tag{2.6}$$

Let's walk through each activation function. e_t is an end-of-stroke probability. This is specific to our model, but we want to push it into the range of (0,1) because that is the range of any given probability. Similarly, for our π's, these are our mixture weights. We therefore push them through the softmax activation function so that the weights collectively sum to 1. The means, μ, we do not enforce any activation function because we want to let them span any given scalar value. The variances σ we push through the exponential activation function so as to only allow positive variances (because variance cannot be negative). Finally, our correlations we force through the hyperbolic tangent function. This allows the range of the rhos to be contained in (-1,1) exclusive. As we will touch on later, this gave rise to a few different issues.

We want to predict what the next input is going to be given our output vector. Essentially, this is asking for $\Pr(x_{t+1} \mid y_t)$. There is a decent amount of statistical backing that goes into evaluating the probability distribution for this bivariate Gaussian mixture model. While the rigorous proof will not be shown, the equation for the conditional probability is shown below:

$$\Pr(x_{t+1} \mid y_t) = \sum_{j=1}^{M} \pi_j^t \, \mathcal{N}(x_{t+1} \mid \mu_j^t, \sigma_j^t, \rho_j^t) \tag{2.7}$$

where

$$\mathcal{N}(x \mid \mu, \sigma, \rho) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left[\frac{-Z}{2(1-\rho^2)}\right] \tag{2.8}$$

and

$$Z = \frac{(x_1-\mu_1)^2}{\sigma_1^2} + \frac{(x_2-\mu_2)^2}{\sigma_2^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2} \tag{2.9}$$

This equation is the generalized version of calculating the conditional probability for a bivariate Gaussian distribution. This mathematical construction and formula can be derived as shown on Wolfram's Mathworld. However, because we also have a handwriting specific end-of-stroke parameter, we modify our conditional probability formula to result in our final calculation of:

$$\Pr(x_{t+1} \mid y_t) = \sum_{j=1}^{M} \pi_j^t \, \mathcal{N}(x_{t+1} \mid \mu_j^t, \sigma_j^t, \rho_j^t) \begin{cases} e_t & \text{if } (x_{t+1})_3 = 1 \\ 1 - e_t & \text{otherwise} \end{cases} \tag{2.10}$$

This is our final conditional probability output from the MDN. Note, that once we have this, performing EM is simple as our loss function that we choose to minimize is just:

$$\mathcal{L}(\mathbf{x}) = -\sum_{t=1}^{T} \log \Pr(x_{t+1} \mid y_t) \tag{2.11}$$

2.3.3 Stacked LSTM and MDN

We are now ready to talk about the complete model that Graves creates in order to perform handwriting synthesis. We can use an MDN to generate a probablistic prediction - however now, we want our neural network to actually be a cascade of LSTMs. If there is any uncertainty about how an LSTM works, please refer to Section 1.2.3. Instead of just using a single LSTM for our neural network, we can actually use a cascade. The structure of the model can be seen in Figure 2.1, which is taken directly from Graves's paper.

The LSTM cascade buys us a few different things. As Graves aptly points out, it mitigates the vanishing gradient problem even more greatly than a single LSTM could. This is because of the skip-connections. All hidden layers have access to the input and all hidden layers are also directly connected to the output node. As a result, there are less processing steps from the bottom of the network to the top.

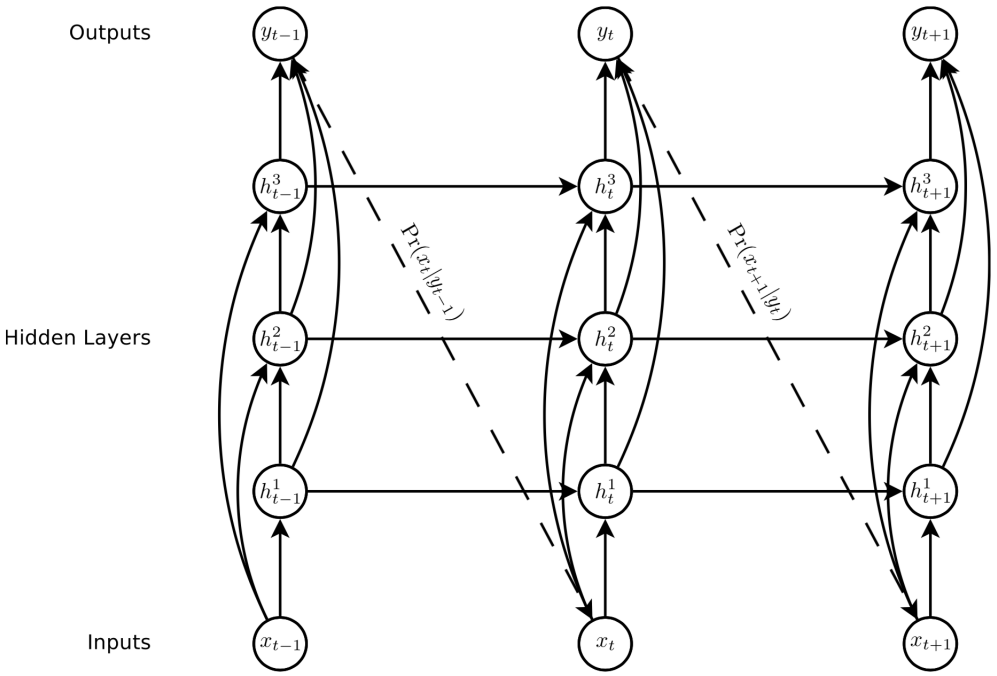

FIGURE 2.2: Architecture of the total model, where the hidden layers represent LSTM cells.

While Graves also does not note on this, the cascade of LSTMs also buys us a dimensionality increase. If we say we have $m = 3$ hidden layers, we can follow the skip sequences through to keep track of dimensionality.

Let's observe the x_{t-1} input. h^1_{t-1} only has x_{t-1} as its input which is in $\mathbb{R}^3$ because (x, y, eos). However, we also pass our input x_{t-1} into h^2_{t-1}. We assume that we simply concatenate the original input and the output of the first hidden layer. Because LSTMs do not scale dimensionality, we know the output is going to be in $\mathbb{R}^3$ as well. Therefore, after this concatenation, the input into the second hidden layer will be in $\mathbb{R}^6$. We can follow this process through and see that, the input to the third hidden layer will be in $\mathbb{R}^9$. Finally, we concatenate all of the LSTM cells (i.e. the hidden layers) together, thus getting a final dimension of $\mathbb{R}^{18}$ fed into our MDN. Note, this is for $m = 3$ hidden layers, but more generally, we can observe the relation as

$$\text{final dimension} = k\frac{m(m+1)}{2} \tag{2.12}$$

We note that this comes from a simple geometric sum multiplied by the original input of the first dimension. Here k is the dimension of x_{t-1} and m is the depth of our hidden layer. We can see that if we substitute in $k = 3$ and $m = 3$, we result in 18 as our final dimension.

The increase in dimensionality is beneficial for our model because it expands our representative space. $\mathbb{R}^3$ might not be a large enough space for our inputs to be expressed, but by combining them with the LSTM outputs we can recognize a larger combination of inputs more effectively.

Now, that we have increased the dimensionality, we can feed the output from our LSTM cascade into the GMM in order to build a probabilistic prediction model for the next stroke. With this in mind, the GMM will then be fed the actual next point, in order to create some idea of the deviation so that the loss can be properly minimized. In Figure 2.1, the dashed lines from the y_{t-1}'s to the x_t's represent the MDN at work. We are generating some conditional probability of the next input that we are going to see.

Chapter 3

Creating the Model

With both the theory of LSTMs carefully explained and the math behind the different building blocks of Graves's model explained, we are now ready to discuss the implementation details of our project. We arrived at our final model with a good project design after having built all of the individual components, we were easily able to link them together without much alteration to the actual code. Various hyperparameters were actually kept the same, with the occasional input dimension being altered to glue models together.

3.1 Gaussian Mixture Model

To visualize and ensure that we fully understood the Gaussian mixture models, we first solved the problem with some toy data - namely, the Iris flower dataset. We looked at two dimensional data in an effort to simulate the math behind a bivariate Gaussian distribution. We compared the sepal length to the sepal width and tried to see how best to visualize our data.

One thing that we did frequently take advantage of throughout of our program was broadcasting and playing with dimensionality. Broadcasting is a feature in both `numpy` and `tensorflow` that allows different sizes tensors to be broadcast to one similar shape. We therefore created our input variable x to be "pseudo"-3-dimensional. We added an extra dimension so that we could allow broadcasting to work effectively. Essentially, our input then has shape: $(?, numDim, 1)$, where ? refers to the number of input examples we are given.

We then randomly initialize all of our mixture parameters. We take careful note to let the pis be uniform at the start, so that we do not favor one distribution over the

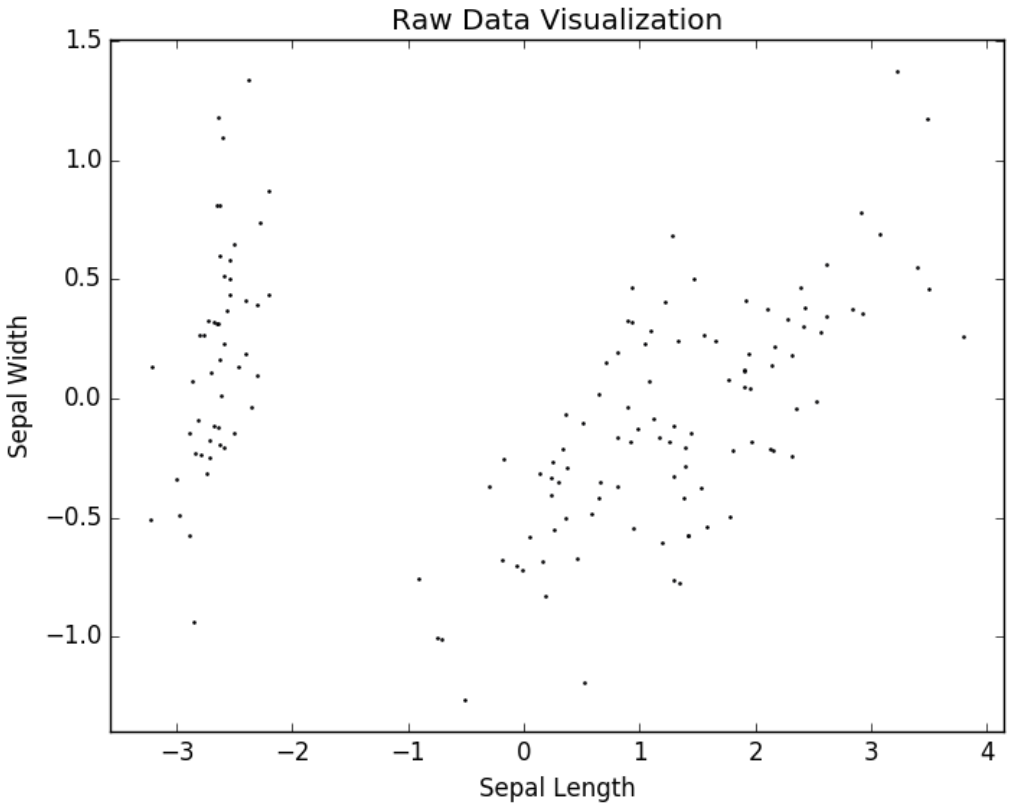

FIGURE 3.1: Raw data from the Iris flower data set. Note, the lengths and widths were mean shifted.

other. We perform the appropriate reductions and summations, all whilst keeping the data vectorized so as to not perform costly functions. Part of the large benefit of using Python is that things can be handed off to `numpy` and `tensorflow` which have much faster behind the scene methods written in C++ and C. This allows for both ease of coding, while still getting quick and efficient results.

Due to the lightness of the math and the little variables that we actually have to train and learn, we were able to compute train over the entire dataset for 10000 iterations. This is not possible in later models, simply because of the time that training would have required.

We then compared the results obtained by `tensorflow` to the results obtained by an `sklearn` example. The raw data that we were looking at is shown in Figure 3.1. The comparison between `sklearn` and `tensorflow` is also shown.

We can verify the correctness of the computation written in Tensorflow. We can see that one of the local minima that Tensorflow converges to in training also seen in

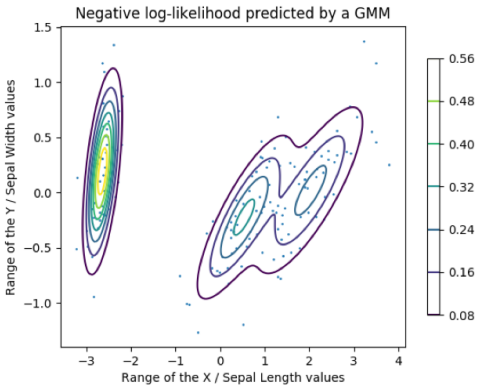

FIGURE 3.2: Sklearn Predicted GMM

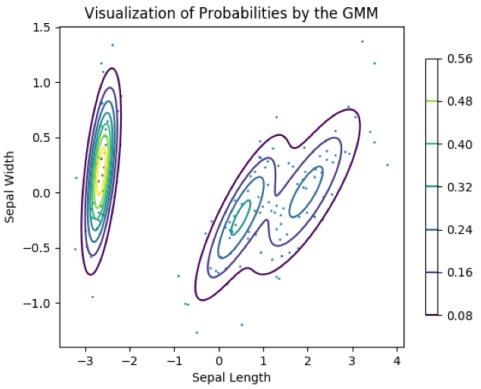

FIGURE 3.3: Tensorflow Predicted GMM

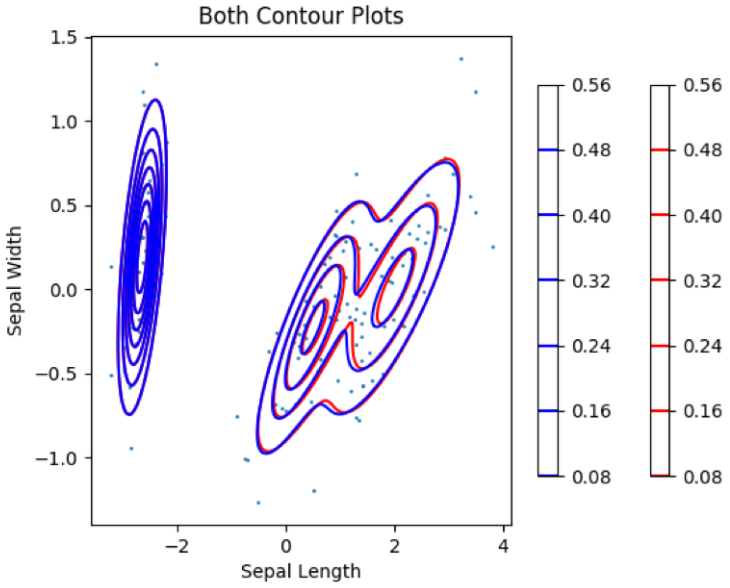

FIGURE 3.4: Overlaid dataset showing comparison between Tensorflow and Sklearn.

sklearn's version of the Gaussian mixture model through Figure 3.4. The result gave us strong confirmation that we had implemented the GMMs correctly.

It is also interesting to demonstrate the idea of multiple local minima. This is an optimization problem at heart, but there are also several local minima that the optimizer can fall into. Note, for this example, RMS prop was used to minimize the loss. If initially looking at the raw data seen in Figure 3.1, it seems intuitive that the clump of data near the left hand side of the graph could be represented by a single Gaussian. However, the clump of data on the right of the graph is a more challenging task. There are multiple ways that one could envision on drawing Gaussians to best represent the data. Similarly, our GMM program also encodes this logic.

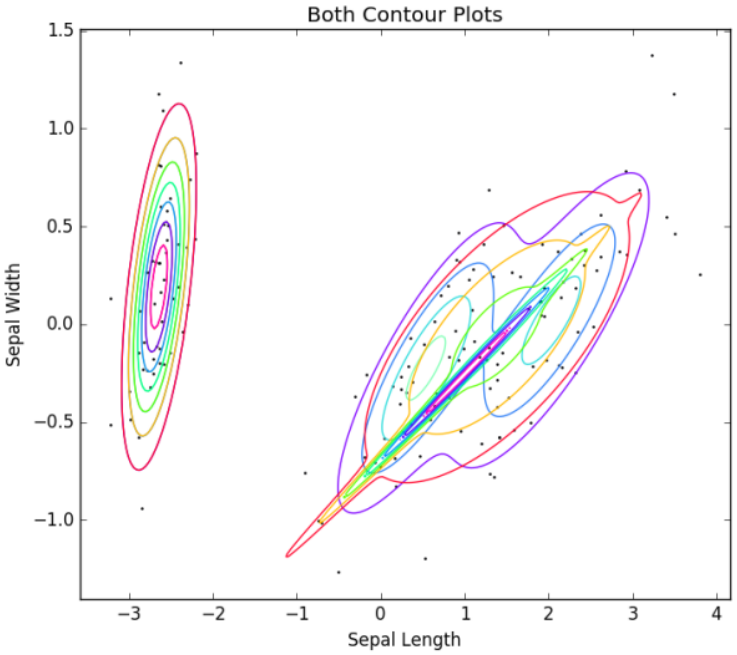

FIGURE 3.5: Local Minima 1

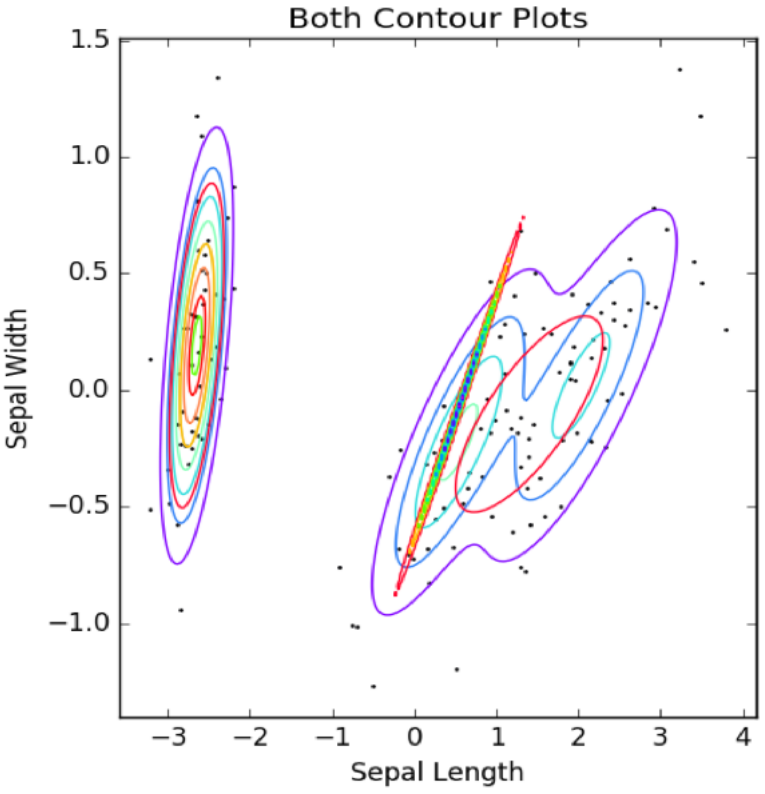

FIGURE 3.6: Local Minima 2

3.2 Mixture Density Network

The mixture density network, or MDN, was constructed at first using a basic neural network. We constructed the MDN using a hidden layer that scaled the input to $\mathbb{R}^{30}$, a space that we decided on almost arbitrarily. This was simple to implement in tensorflow, constructing simply a matrix and bias vector where the weights were learned and scaling accordingly. In other words, this could be accomplished in a single line of code (after instantiating the variables randomly) like such:

```
self.h = h = tf.nn.tanh(tf.add(tf.matmul(x, wh), bh))
```

Note the appropriate shapes. This can be represented as the nonlinear operation:

$$\underbrace{h}_{(?\times nhid)} = \underbrace{x}_{(?\times(nstep\cdot ndim)} \underbrace{wh}_{(nstep\cdot ndim)\times nhid} + \underbrace{bh}_{nhid} \tag{3.1}$$

And because of broadcasting the dimensionality of the addition operation works completely fine. So in other words, if we have a step of three, an input dimension of two, and a hidden size of 30, then the following operation would look like:

$$\underbrace{h}_{?\times 30} = \underbrace{x}_{?\times 6} \underbrace{wh}_{6\times 30} + \underbrace{bh}_{30} \tag{3.2}$$

And we can see that the dimensionality works out perfectly. We then apply the normal rules that are appropriate in order to learn the various parameters of the underlying distribution of the data. Recall, this was a softmax activation for the mixture weights, exponential for the variances, etc. At this point, we still had not added in an additional learned parameter for the end of stroke because we were focusing solely on two dimensional sample data.

At this point, we also had not yet trained on our actual handwriting data for fear of over-complicating the beginning processes of our model. We wanted to ensure that this basic model was doing what we wanted on our sample data. We therefore created some sample data that had some similarities to basic handwriting - that is, loops and straight hard lines. For reference, see Figure 3.7 and Figure 3.8.

The results from the MDN prediction were successful to a large degree. Note, both dropout and random noise were added here so that our model would learn more of the general characteristics of the data rather than memorizing the data exactly. The dropout rate was 0.85, meaning that for every epoch 15 percent of the total weight connections would be randomly chosen to be temporarily severed. Dropout is an effective way to bolster the model and prevent overfitting. Note, that our model is probabilistic so the model's predictions are represented as a heat map. We queried on a meshgrid to test

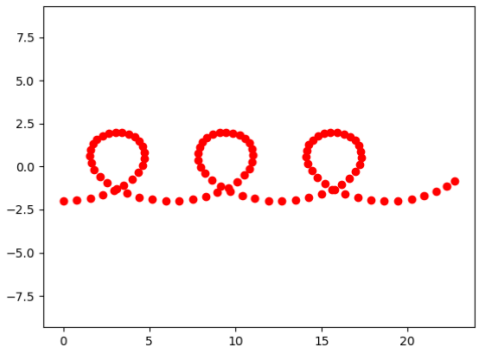

FIGURE 3.7: MDN Sample Data 1

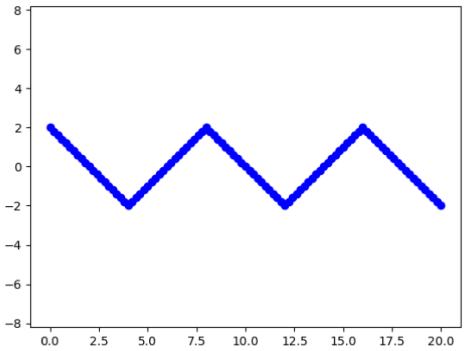

FIGURE 3.8: MDN Sample Data 2

the probability of seeing certain regions over others. For some convincing results, see figures 3.9 and 3.10. In addition, there are other visualizations in Appendix C.

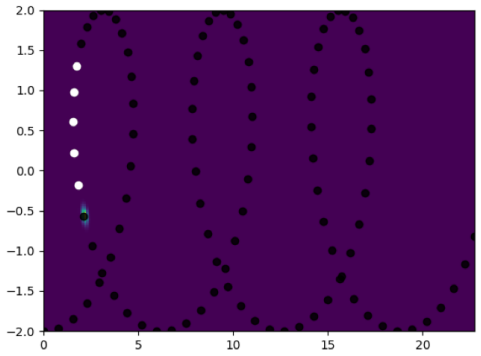

FIGURE 3.9: MDN Prediction Viz 1

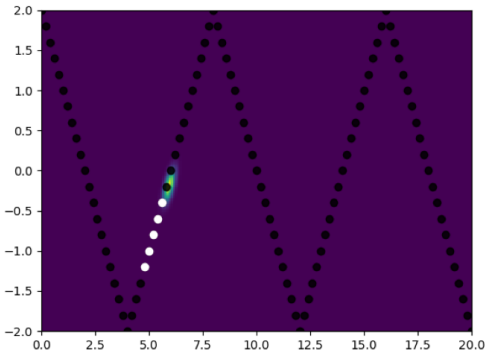

FIGURE 3.10: MDN Prediction Viz 2

With these visualizations created, we were confident that we understood the structure of a Mixture Density Network and we could replace our hidden layers. We thus moved onto the Stacked LSTM cell.

3.3 Stacked LSTM

Before attempting to simply replace the hidden layer in our MDN with the LSTM cascade outlined in section 2.3.3, we focused on building the LSTM cascade. We continued to use the same shape of sample output data as above - the loops and jagged lines - but duplicated them many times creating a longer stream of data, as LSTMs should have no problem learning long term dependencies.

When building our model in `tensorflow` it was necessary to pay special attention to how we achieved the dimensionality increase as well as how we updated the individual

states of the LSTMs and carried these states through timesteps between modules. In order to achieve the dimensionality increase we looped over our LSTMs, first starting with a base case if it was the initial input, and then iteratively concatenating the output of the LSTM with the initial input vector which was passed as input into the LSTM one layer up. Updating the states was done in the same loop. We stored the states of each LSTM in a state list, and as we were determining the input to the next cell, we were concurrently updating the state of the LSTM currently being addressed in the loop.

When we were training our LSTM model we settled on a batch size of twenty and a step size of eight. We continued to implement dropout as well as adding random noise after every epoch in order to avoid overfitting.

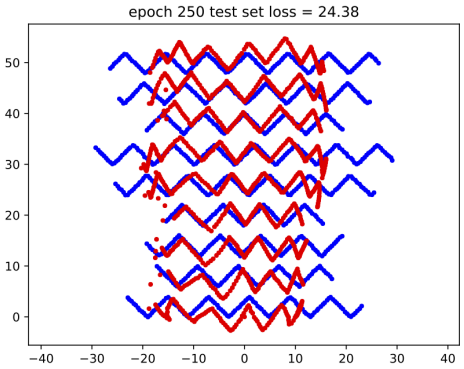

FIGURE 3.11: LSTM Prediction Viz 1

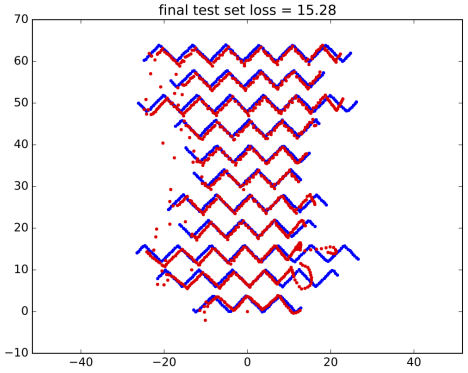

FIGURE 3.12: LSTM Prediction Viz 2

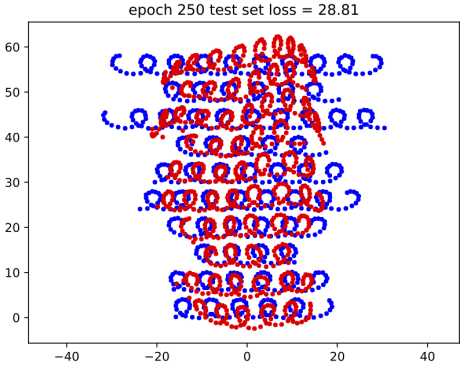

FIGURE 3.13: LSTM Prediction Viz 3

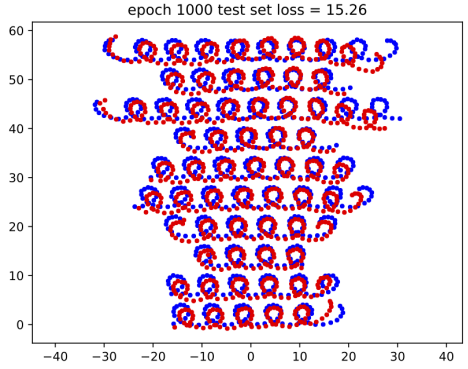

FIGURE 3.14: LSTM Prediction Viz 4

From Figures 3.11-3.14 we clearly see that our LSTM cascade is learning the general shape of the input data. After only 250 epochs the cascade has some idea of the general shape of the data but still makes mistakes and seems to get lost at certain points. However, once it reaches around 1000 epochs it does a much better job. In both examples we see it loosing its way near the end of the sequences. This happens because

the end of all the sequences was treated as validation data, meaning it was data it had not been trained on.

The outcome of these figures gave us the confidence necessary to move on to the next phase - connecting our LSTM cascade and MDN.

3.4 Final Model - Stacked LSTM and MDN

In order to create our final model we took our MDN model, removed the hidden layer of 30 nodes, and replaced it with our LSTM cascade. The largest obstacle we overcame to achieve this step was concerned with variable reuse. When we made our LSTM cascade module certain variables were reused across time steps - most importantly those pertaining to the states - however, this had not yet been a necessary function of our MDN. This meant when initially coupling the two classes, we were generating new MDN parameters after every timestep rather than carrying them through every timestep.

Once our final model was built, we were finally able to train on our handwriting data. We used a batch size of 5, step size of 200 and three Gaussians. Note, that once again our model is probabilistic so the model's predictions are represented as a heat map. We always queried on validation data, meaning our model have never before seen the data.

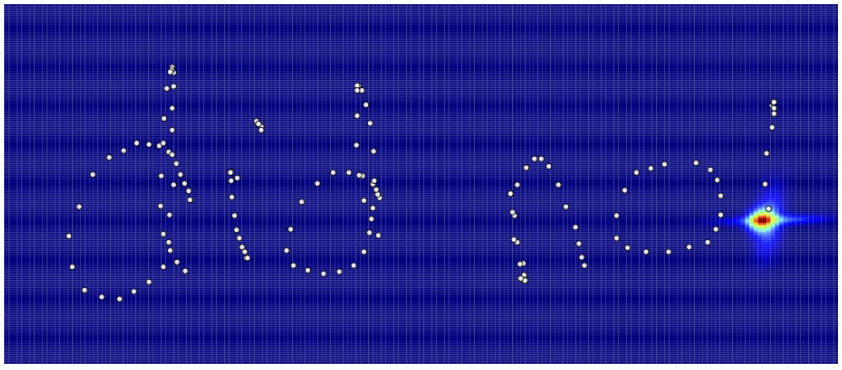

FIGURE 3.15: Final Model Prediction: "Did Not" at Epoch 0

In Figures 3.15 and 3.16 we are writing out the words *did not* and are asking our model to predict the next point in the letter *t*. In Figure 3.15 we see that our model predicts a large probability space but as we train longer by running through more epochs, shown in figure 3.16, the probability space gets much tighter and our model becomes very confident about where the next point is going to land. Note that the predictions not only get tighter but also move down as our model learns the proper spacing.

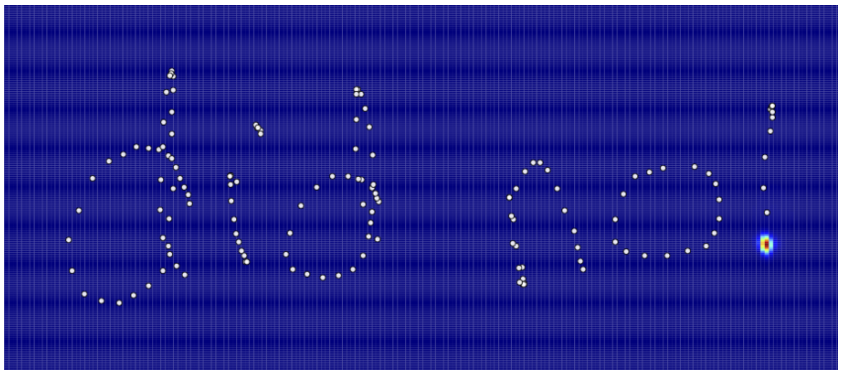

FIGURE 3.16: Final Model Prediction: "Did Not" at Epoch 60

Figures 3.17 and 3.18 once again demonstrate learning but this time in the other direction. We are writing out *work* and give our model the first point of the letter *k*. Initially our model is very confident about where it thinks the next point is going to be, but as it runs over the data more and more times it learns that the probability space of the second point in a letter is much larger than it originally thought, which is a logical progression. We would not expect our model to know perfectly where the second point would be.

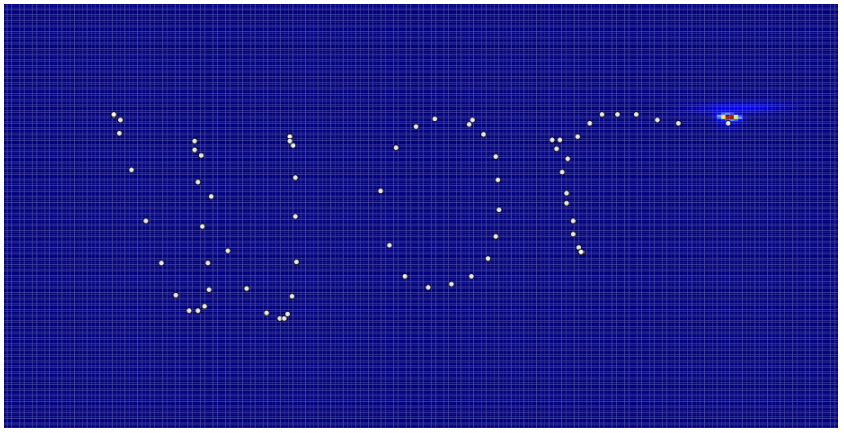

FIGURE 3.17: Final Model Prediction: "Work" at Epoch 0

Once we were convinced that our model was training successfully we allowed it to run overnight and saved a final trained model that would be used for any further querying and visualizations.

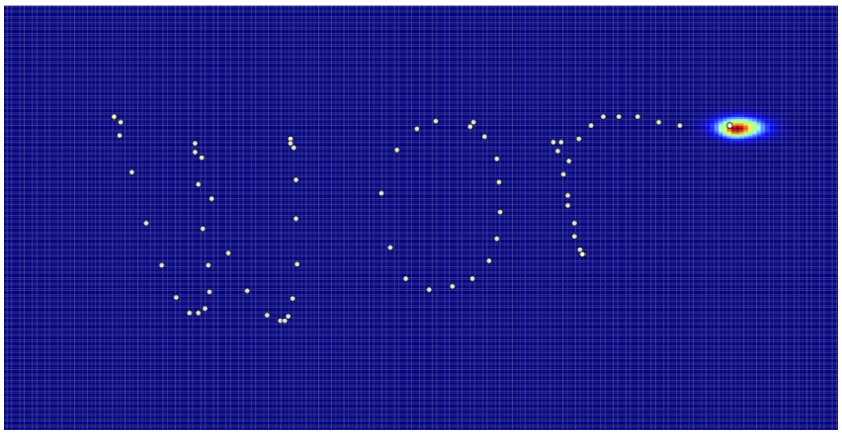

FIGURE 3.18: Final Model Prediction: "Work" at Epoch 30

3.5 Extraneous

3.5.1 Data

Our handwriting data came from the IAM Online Handwriting Database. 657 writers were asked to use a smart tablet to write down lines of the Lancaster-Oslo-Bergen text corpus for British English, resulting in a total of 11035 lines of handwriting in our training set, and 580 lines in our validation set. The data came in XML format where every line was a separate folder, and every line was subsequently broken up into strokes. The raw data was (x, y, eos) points where eos was the binary end-of-stroke marker. All our training was done on offsets as they are independent of starting point, and all lines were treated as separate sequences as there were limited, or no, dependencies between lines.

3.5.2 Saving the Model

At some point due to the sheer size of our model and the length that training took ($\sim$ 12 hours for a reasonable loss), we needed to be able to quickly load in a trained model and query for interesting results. As a result, Tensorflow's `tf.train.Saver()` was used excessively. We would store checkpoints for various runs and after a multitude of epochs, we would then load them in query our model. We then specified in our program that the second command line argument would be the checkpoint of the various model that we wanted to load in.

Another check that we implemented to confirm that we had successfully loaded in the model was to print out the global variables after the restore. This was done by iterating over `tvars = tf.global_variables()`. Note, that Tensorflow will create

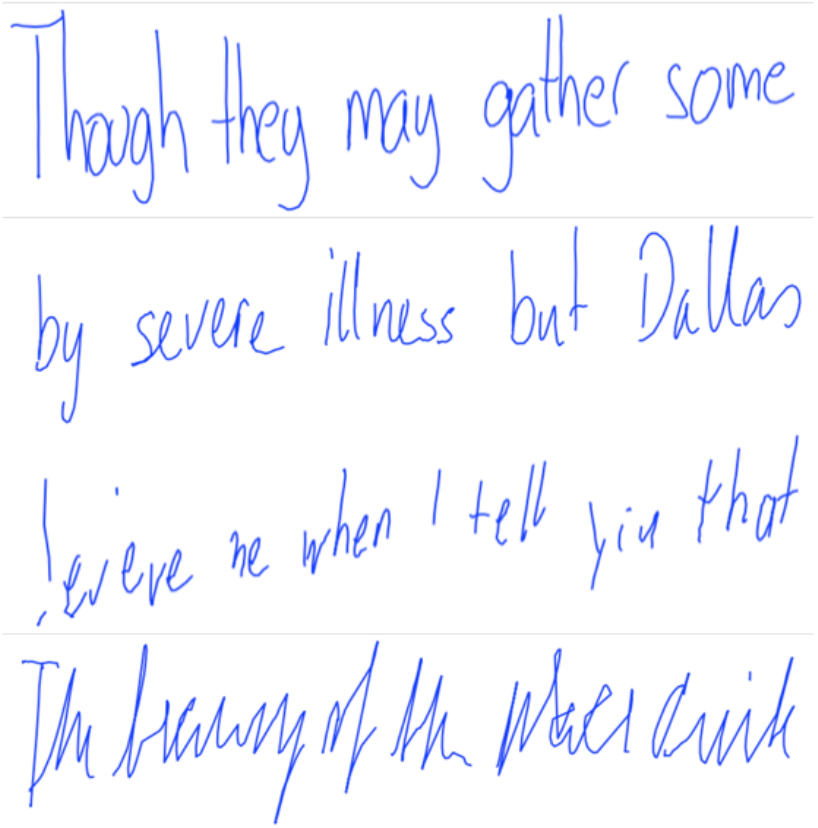

FIGURE 3.19: Example Lines of Data

duplicate variables for the ones that have been trained. We were not sure where the source of this came from, besides the suspicion that Tensorflow creates a slightly modified version of a variable if it is being optimized.

Chapter 4

Future Work and Extensions

4.1 Current Model

While our current model does effectively learn how handwriting is being generated and is strong with its predictive ability, we were unable to get the model to sample effectively. This concept was proposed by Graves with the idea that because the model learned probabilistically what handwriting generally looks like, that by sampling from the learned distribution and feeding in the outputs the model would actually generate handwriting. However, for multiple reasons, including simple lack of memory and training time, we were not able to effectively generate handwriting.

An obvious extension from generating handwriting would by synthesizing handwriting, which is one of the huge attractions from the Graves paper. He is able to modify his architecture to have a window which learns the dependencies between certain letters. This would be an interesting extension for future E90s as the architecture of the system is rather varied and also allows for exploration into convolution and the power that brings.

4.2 Potential Other Models

Another interesting idea that Graves proposes is simulating this model and architecture, but instead of using handwriting data, using audio signals. With this idea, it might be possible for the model to learn to talk convincingly at random. With a slight modification to the model, it could learn the dependencies between words, similar to how Graves's modified model learned how to synthesis handwriting.

Another possible extension that we were curious about is the possibility of extending this model with other languages. This concept hinges on a few different things - namely, the data source available, but also the regularity of how the characters of the language are constructed. For example, it might be hard for a language that does not have a set alphabet because of the difficulty to simulate strokes.

Appendix A

Software Details - Tensorflow

Our programs were implemented in Python 2.7, although we did use various `future` imports so that our code would be compatible with Python 3.0+. The version of Tensorflow used was version 0.11. Note, that from when we `pip` installed Tensorflow to finishing our E90, the most up to date Tensorflow version is now 1.1. Because of the numerous updates, finding reasonable documentation as well as version control was a challenge in and of itself.

More generally however, Tensorflow introduces a new sort of programming paradigm that both authors struggled with. In Tensorflow, there are two steps to a successful program:

1. Building your computational graph
2. Running your computational graph

While in initially this may sound relatively easy, there are numerous complexities that need to be fully understood before feeling comfortable with Tensorflow. The definition of a computational graph should also be elaborated on. A computational graph is a graph that represents the orders of operation and dependencies of your program. Now, there is no simple `+` operation, rather there is an `add` node that is created. Tensorflow essentially creates this graph where the nodes are operations or tensors and the edges show the connections and or dependencies of the objects.

This is effective for multiple reasons, but effective for machine learning because the order of operations is very clear. Backpropagation can happen efficiently and is entirely abstracted away because the dependency of variables is baked into the program itself. Therefore computing partial derivatives on the fly becomes a no brainer that Tensorflow can execute efficiently.

One of the most obvious discomforts with Tensorflow was the inability to do simple `print` debugging. All of the data is stored as a `Tensor` object. As a result, when you print something out, the value has not been computed yet and all you see is a tensor of a various shape and size. While this can be effective for checking dimensionality conversions and reductions, it is hard to have an idea of the actual values without actually running your computational graph and then pulling out specific values.

There was generally a lack of good tutorials that walked through Tensorflow. Google's own Tensorflow tutorials don't explicitly go into this structure and dependency graph. As a result, the debugging process was elongated because of the inability to debug like a normal program. Certain things like the `QueueRunner` class were not easily explained from the documentation. Small details like improper initialization of a `QueueRunner` variable would lead to your computational graph seizing to do anything at all. No error would be thrown, the graph would seemingly loop forever.

However, by the end of the project, both authors felt immensely more comfortable working with Tensorflow and manipulating data. Both hope to utilize Tensorflow on future projects.

Appendix B

Hardware Details - Conway

We were lucky enough to be able to use our advisor, Matt Zucker's desktop. The OS installed was Ubuntu 14.04.5 LTS. We installed Tensorflow v0.11 through a virtual environment so we could preserve our version if we had to download Tensorflow again. The computer was conned "Conway", which we commonly referred to and `ssh`ed into or `scp`ed from.

Conway provided a dramatic speedup because of its sole purpose as well as the GPU being fully enabled. A large benefit of Tensorflow is its ability to automatically check and perform computation on the GPU if it is accessible. While we simply had a CPU enabled version of Tensorflow downloaded on their computers, this rapidly became unable to efficiently burn through the models due to the size of the data provided.

While there were various hyperparameters that could have been modified in order to change the duration of training, the final model with the entire training dataset still would have taken up to near 70 days to make 2500 passes through the data. On our computers, this same process would have taken on the order of 229 days.

Appendix C

Other Visualizations

Here are some other visualizations from the MDN with solely a hidden layer.

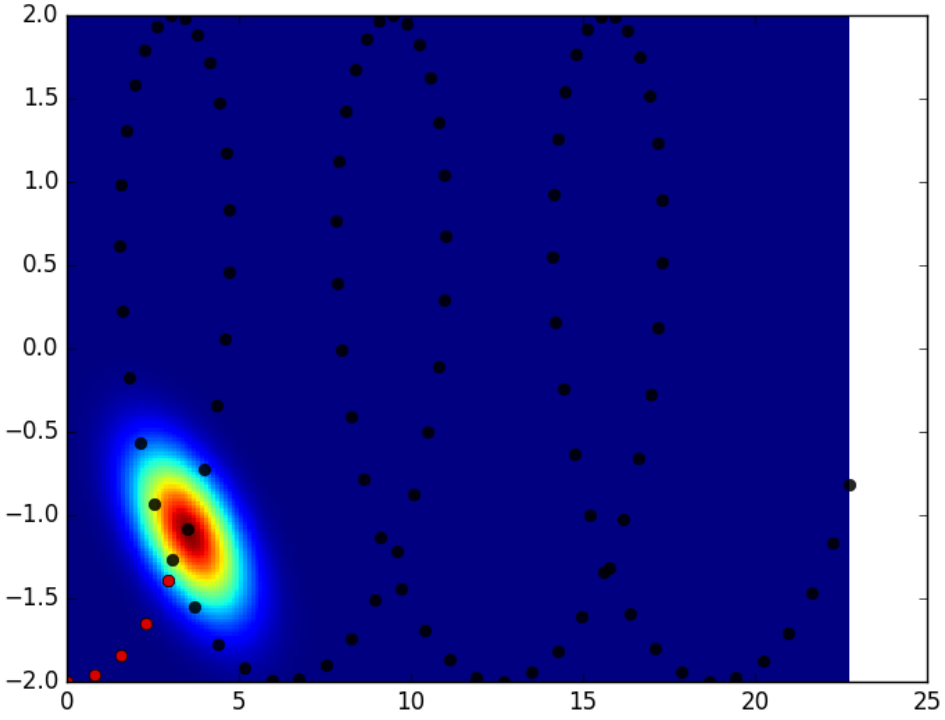

FIGURE C.1: Sample Prediction from MDN

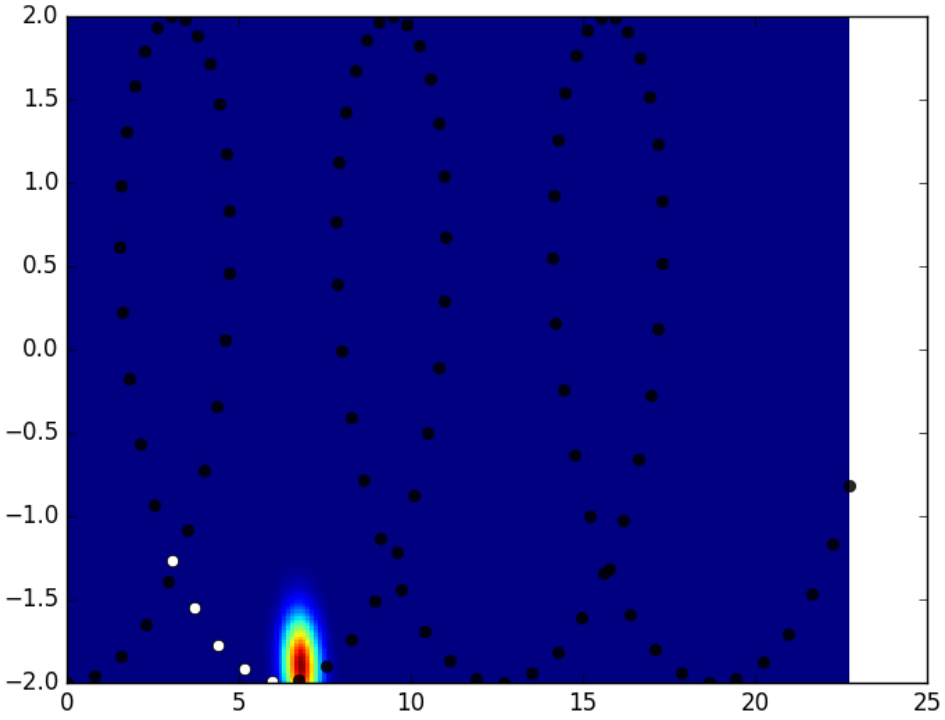

FIGURE C.2: Sample Prediction 2 from MDN

Appendix D

Code

D.1 MDN Class Code

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import sys

# Prediction output dimension
output_dimension = 2

# Make everything float32
d_type = tf.float32


class MDN:

  def __init__(self, input_data_from_lstms, targets, final_dimension_from_lstm, is_training):
    # The input is now longer raw data
    # The input is now the output of our LSTMs
    # We will use this as a module in our model
    self.NHIDDEN = 30
    self.NCOMPONENTS = 3
    self.STEP = final_dimension_from_lstm
    self.ndim = output_dimension
    self.data = input_data_from_lstms
    self.iterations = 8000

    self.x = x = input_data_from_lstms

    ''' Here, we are building the MDN'''
    # Setting up the Pi's (they are one dimensional)
    w_pi = tf.get_variable(name = 'w_pi', initializer = tf.random_normal(
        [final_dimension_from_lstm, self.NCOMPONENTS], dtype = d_type))

    b_pi = tf.get_variable(name = 'b_pi', initializer = tf.random_normal(
```

```
        [self.NCOMPONENTS], dtype = d_type))

    p = tf.add(tf.matmul(x, w_pi),b_pi)
    pis = tf.exp(p)
    pis = pis/tf.reshape(tf.reduce_sum(pis, axis=1), [-1, 1])

    # Setting up the correlations
    w_corr = tf.get_variable(name = 'w_corr', initializer = tf.random_normal(
        [final_dimension_from_lstm, self.NCOMPONENTS], stddev = 0.2, dtype = d_type))

    b_corr = tf.get_variable(name = 'b_corr', initializer = tf.random_normal(
        [self.NCOMPONENTS], stddev = 0.2, dtype = d_type))

    precorr = tf.add(tf.matmul(x,w_corr), b_corr)
    corr = tf.tanh(precorr) # Shape (?, number of components) = (?, 3)

    # Setting up the means
    w_mu = tf.get_variable(name = 'w_mu', initializer = tf.random_normal(
        [final_dimension_from_lstm, self.NCOMPONENTS*self.ndim], dtype = d_type))

    b_mu = tf.get_variable(name = 'b_mu', initializer = tf.random_normal(
        [self.NCOMPONENTS*self.ndim], dtype = d_type))

    # no activation function for the mus
    mu = tf.add(tf.matmul(x,w_mu),b_mu) # Shape (?, num comp * num dimensions) = (?,6)


    # Setting up sigma and variance
    w_sigma = tf.get_variable(name = 'w_sigma', initializer = tf.random_normal(
        [final_dimension_from_lstm, self.NCOMPONENTS*self.ndim], dtype = d_type))

    b_sigma = tf.get_variable(name = 'b_sigma', initializer = tf.random_normal(
        [self.NCOMPONENTS*self.ndim], dtype = d_type))

    sigma = tf.add(tf.matmul(x,w_sigma),b_sigma)
    sigma = tf.exp(sigma) # Shape (?, num comp * numb dimensions) = (?, 6)

    # end of stroke parameter
    eos = tf.get_variable(name = 'eos', initializer = tf.random_normal([1], dtype = d_type))
    self.eos = eos = tf.divide(1.0, tf.add(1.0,tf.exp(eos)))


    ''' Here we are going to build the mixture probabilites'''
    sum_of_pis = tf.reduce_sum(pis)

    # Target values
    xy_targets = tf.slice(targets, [0, 0], [-1, 2])
    eos_targets = tf.slice(targets, [0, 2], [-1, 1])
    self.actual = actual = tf.reshape(xy_targets, [-1, 1, self.ndim])

    # Need to first do some dimensionality manipulation
    # specifically for mu and sigma
    # need to reshape with the 3 first, so that our x1 and x2 points get multiplied correctly
    mu = tf.reshape(mu, [-1, 3, 2])
    sigma = tf.reshape(sigma, [-1, 3, 2])
```

```
  # The output we feed into this function should be that from the hidden layers
  var = tf.mul(sigma,sigma) # Shape (?, 3, 2)

  # Compress along the number of gaussians so that we still have the shape: 1 x 3 )
  s1s2 = tf.reduce_prod(sigma, axis=2) # Shape: (?,3)
  # s1s2 = tf.reshape(s1s2,[-1]) # Shape: (3,) # CHECK SHAPE HERE

  # deviation from mean is [?, n_components]
  # so x is going to be like 150 x 2 and then mus is going to be 3 x 2
  # actual - (?, 2, 1)
  # mu - (?, 6)
  dev = tf.sub(actual,mu) # Shape: (?, 3, 2)
  # Shape explanation: we have a bunch of sequences and we have x and y points
  #          for three different components

  # Have to build Z equations first (eqn 25)
  # Broadcasting works - ? x 2 x 3
  z12_before = tf.div(tf.mul(dev,dev),var)
  # shape above: (?, 2, 3)
  z12 = tf.reduce_sum(z12_before, axis=2)
  # shape above: (?, 3)

  # dev is ? x 3 x 2. the six comes from numberofcomp * number of dimensions
  # after reduce prod it's ? x 3 which is good.
  # six is good because we have three components and each has an x and a y!!
  reduce_dev = tf.reduce_prod(dev, axis=2) # Shape: (?, 3)

  # z3 is going to be (?, 3)
  # this math also works out and makes sense
  z3 = tf.div(reduce_dev*corr*2, s1s2)
  Z = z12 - z3

  # Building Normal Distribution (eqn 24)
  normalizer = (2.0 * np.pi * s1s2) * tf.sqrt(1.0-tf.mul(corr,corr))

  expon_part = tf.exp(tf.div(-Z, 2.0 * (1.0 - tf.mul(corr,corr))))

  N = tf.div(expon_part, normalizer) # Shape: (?, 3)

  # Building conditional probability (eqn 23)
  # overall mixture probabilities has shape [?]
  N_by_pis = tf.mul(N,pis)
  self.N_by_pis = N_by_pis

  eos_operator = tf.mul(eos,eos_targets) + tf.mul(1.0-eos, 1.0-eos_targets)
  self.eos_operator = eos_operator

  N_by_pis_by_eos = tf.mul(N_by_pis, eos_operator)

  self.mixture_prob = mixture_prob = tf.reduce_sum(N_by_pis_by_eos, axis=1)

def compute_loss(self):
  mixture_prob = self.mixture_prob
```

```
    negative_log_odds = -tf.log(tf.maximum(mixture_prob,1e-20))
    loss = tf.reduce_sum(negative_log_odds)

    # log prob flat array batch size by num steps
    # corrected loss
    # reshape to batchsize x num steps
    # elementwise product with erreweight
    return loss, negative_log_odds

  def return_mixture_prob(self):
    return self.mixture_prob
```

D.2 LSTM Code

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from collections import namedtuple
import sys
import os

from MDNClass import MDN
from DataLoader import DataLoader
from gmm_sample import *

# Choose 1 or 2 for different ydata graphs
PLOT = 1

# Make everything float32
d_type = tf.float32

# Batch size for training
train_batch_size = 10

# Number of steps (RNN rollout) for training
train_num_steps = 250

# Dimension of LSTM input/output
hidden_size = 3

# should we do dropout? (1.0 = nope)
train_keep_prob = 0.80

# number of training epochs
num_epochs = 1000

# how often to print/plot
update_every = 10
```

```
# how often to save
save_every = 1

# initial weight scaling
init_scale = 0.1

# Number of things in our cascade
steps_in_cascade = 3

# Input dimension
input_dimension = 3

# Handle sequencing or not
handle_sequences_correctly = True

# do xy offsets or not
do_diff = True

# learning rate
learning_rate = 1e-4

# do we want gifs?! yes?
CREATE_GIFS = False

# do we want to generate handwriting
GENERATE_HANDWRITING = True

# do we want to visualize with tensorboard
CREATE_TENSORBOARD = False

######################################################################
# Helper function for below

def get_xy_data(n):
  u = np.arange(n)*0.4 + np.random.random()*10
  if PLOT == 1:
    x = u
    y = 8.0*(np.abs((0.125*u - np.floor(0.125*u)) - 0.5)-0.25)
  else:
    x = u + 3.0*np.sin(u)
    y = -2.0*np.cos(u)

        x -= x.min()
        y -= y.min()
  return x, y

######################################################################
# Get training data -- the format returned is xi, yi, 0 except for new
# "strokes" which are xi, yi, 1 every time the "pen is lifted".

def get_data(data):
    cur_count = 0
    all_data = []
    all_sequence_info = []
```

```
    subsequence_index = 0

    for i in range(len(data)):
      sequence = data[i]
      length_sequence = len(sequence)

      # sequence info has two columns
      sequence_info = np.zeros((length_sequence,2),dtype = int)

      # first column is all ones expcet for 0 for the very first point in sequence
      sequence_info[0,0] = 0
      sequence_info[1:,0] = 1

      # second column just holds which subsequence we are on -- not used for training
      # used to visualize rows in PDFs

      sequence_info[:,1] = subsequence_index
      subsequence_index += 1

      all_sequence_info.append(sequence_info)

    all_sequence_info = np.vstack(tuple(all_sequence_info))
    all_data = np.vstack(tuple(data))

    return all_data, all_sequence_info

########################################################################
class Input(object):

    def __init__(self, posdata, seqinfo, config):

      batch_size = config.batch_size
      num_steps = config.num_steps
      self.posdata = posdata
      self.seqinfo = seqinfo

      with tf.name_scope('producer', [posdata, batch_size, num_steps]):

          # Convert original raw data to tensor
          raw_data = tf.convert_to_tensor(posdata, name='raw_data', dtype=d_type)

          # Convert sequence continuations to tensor - just want for column
          raw_seq = tf.convert_to_tensor(seqinfo[:,0], name = 'sef_info', dtype=d_type)

          # These will be tensorflow variables
          data_len = tf.size(raw_data)//3
          batch_len = data_len // batch_size
          epoch_size = (batch_len - 1) // num_steps

          # Prevent computation if epoch_size not positive
          assertion = tf.assert_positive(
              epoch_size,
              message="epoch_size␣==␣0,␣decrease␣batch_size␣or␣num_steps")
```

```
          with tf.control_dependencies([assertion]):
            epoch_size = tf.identity(epoch_size, name="epoch_size")

          # Truncate our raw_data and reshape it into batches
          # This is just saying grab as much of it as we can to make a clean reshaping
          data = tf.reshape(raw_data[:batch_size*batch_len, :],
                            [batch_size, batch_len, 3])

          seq = tf.reshape(raw_seq[:batch_size*batch_len],
                           [batch_size, batch_len])

          # i is a loop variable that indexes which batch we are on
          # within an epoch
          i = tf.train.range_input_producer(epoch_size, shuffle=False).dequeue()

          # each slice consists of num_steps*batch_size examples
          x = tf.slice(data, [0, i*num_steps, 0], [batch_size, num_steps, 3])
          y = tf.slice(data, [0, i*num_steps+1, 0], [batch_size, num_steps, 3])

          preserve_state = tf.slice(seq, [0, i*num_steps], [batch_size, num_steps])
          err_weight = tf.slice(seq, [0, i*num_steps+1], [batch_size, num_steps])
      # Assign member variables
      self.x = x
      self.y = y
      self.epoch_size = ((len(posdata) // batch_size)-1) // num_steps

      self.preserve_state = preserve_state
      self.err_weight = tf.reshape(err_weight, [batch_size, num_steps, 1])


########################################################################
# Class of Cascading LSTMs
class LSTMCascade(object):

  def __init__(self, config, model_input, is_train, is_sample=False, external_targets=None):

      # Stash some variables from config
      hidden_size = config.hidden_size
      batch_size = config.batch_size
      num_steps = config.num_steps
      keep_prob = config.keep_prob

      # Scale factor so we can vary dataset size and see "average" loss
      # Do this in case we're just looking at a single point and we're querying
      self.loss_scale = batch_size * num_steps * model_input.epoch_size

      # Stash input
      self.model_input = model_input

      # we don't need to reshape the data!
      if is_sample:
          self.lstm_input = tf.placeholder(tf.float32, shape = [1,1,3])
          model_input.y = tf.zeros(shape=[1,1,3])
      else:
          self.lstm_input = model_input.x
```

```
# this is going to be the final dimension
# this is always even
final_high_dimension = input_dimension * steps_in_cascade * (steps_in_cascade+1) // 2

# note: input dimension is equivalent to the hidden size of the LSTM cell
hidden_size = input_dimension

# this will hold all of our cells
lstm_stack = []

# this will hold all of our states as it goes
self.state_stack = []

# this will hold the initial states
init_state_stack = []

# This will reduce our final outputs to the appropriate lower dimension
# Make weights to go from LSTM output size to 2D output
w_output_to_y = tf.get_variable('weights_output_to_y', [final_high_dimension, 2],
                                dtype=d_type)

# we need to # LSTMS = # steps in cascade
for i in range(steps_in_cascade):

    # Make an LSTM cell
    lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(
        hidden_size * (i+1), forget_bias=0.0,
        state_is_tuple=True)

    # Do dropout if needed
    if is_train and keep_prob < 1.0:
        print('doing␣dropout␣with␣prob␣{}'.format(config.keep_prob))
        lstm_cell = tf.nn.rnn_cell.DropoutWrapper(
            lstm_cell, output_keep_prob=keep_prob)

    initial_state = lstm_cell.zero_state(batch_size, d_type)

    lstm_stack.append(lstm_cell)
    init_state_stack.append(initial_state)
    self.state_stack.append(initial_state)

# cache our initial states
self.initial_state = init_state_stack

# Need an empty total output list of ys
outputs = []

# we need this variable scope to prevent us from creating multiple
# independent weight/bias vectors for LSTM cell
with tf.variable_scope('RNN'):

    # For each time step
    for time_step in range(num_steps):
```

```
            # This is y_i for a single time step
            time_step_output = []

            # Prevent creating indep weights for LSTM
            if time_step > 0:
                tf.get_variable_scope().reuse_variables()

            # model_input.preserve_state is a vector of 0's or 1's corresponding
            # to 0 means reset LSTM, 1 means don't
            preserve_step = tf.reshape(model_input.preserve_state[:, time_step],
                                       [config.batch_size,1])

            for i in range(steps_in_cascade):

                with tf.variable_scope("RNN"+str(i)):
                  # Run the lstm cell using the current timestep of
                  # input and the previous state to get the output and the new state
                  curr_lstm_cell = lstm_stack[i]
                  curr_state = self.state_stack[i]


                  # state.c and state.h are both shape (batch_size, hidden_size)
                  # when I multiply by (batch_size, 1) it broadcasts
                  curr_stateTuple = type(curr_state)
                  possible_state = curr_stateTuple(c = curr_state.c*preserve_step,
                                                   h = curr_state.h*preserve_step)

                  # Need a special base case for the first lstm input
                  if i == 0:
                      cell_input = self.lstm_input[:, time_step, :]
                  else:
                      # All of these variables will be defined because of our base case
                      cell_input = tf.concat(concat_dim = 1, values =
                                   [self.lstm_input[:, time_step, :], cell_output])

                  (cell_output, curr_state) = curr_lstm_cell(cell_input,
                                        possible_state)

                  # Update our state list
                  self.state_stack[i] = curr_state

                  # Update the output for the single cell
                  time_step_output.append(cell_output)

            # For every timestep, we need a valid y output that should be of N*L*(L+1)/2
            concated_time_steps = tf.concat(concat_dim = 1 , values = time_step_output)
            outputs.append(concated_time_steps)

    # we need to bookmark the final state to preserve continuity
    # across batches when we run an epoch (see below)
    self.final_state = self.state_stack

    # concatenate all the outputs together into a big rank-2
    # matrix where each row is of dimension hidden_size
    lstm_output_rank2 = tf.reshape(tf.concat(1, outputs), [-1, final_high_dimension])
```

```
    if external_targets is None:
        # reshape original targets down to rank-2 tensor
        targets_rank2 = tf.reshape(model_input.y, [batch_size*num_steps, 3])
    else:
        targets_rank2 = tf.reshape(external_targets, [-1, 3])

    with tf.variable_scope('MDN'):
        ourMDN = MDN(lstm_output_rank2, targets_rank2, final_high_dimension, is_train)
        self.pis, self.corr, self.mu, self.sigma, self.eos = ourMDN.return_params()

    # The loss is now calculated from our MDN
    MDNloss, log_loss = ourMDN.compute_loss()
    self.log_loss = log_loss

    if external_targets is None:
        log_loss = tf.reshape(log_loss, [batch_size, num_steps,1])

        loss = log_loss * model_input.err_weight
        self.loss_by_err_wt = loss
        # What we now care about is the mixture probabilities from our MDN
    else:
        loss = MDNloss

    with tf.variable_scope('MDN'):
        self.mixture_prob = ourMDN.return_mixture_prob()
        self.ncomponents = ourMDN.NCOMPONENTS

    # loss is calculated in our MDN
    self.loss = tf.reduce_sum(loss)
    self.loss_before_max = self.loss
    self.err_wt_reduce_sum = tf.reduce_sum(model_input.err_weight)
    self.loss /= tf.maximum(tf.reduce_sum(model_input.err_weight),1)
    self.after_max_division = self.loss
    # generate a train_op if we need to
    if is_train:
        self.train_op = tf.train.RMSPropOptimizer(learning_rate).minimize(self.loss)
    else:
        self.train_op = None


def run_epoch(self, session, return_predictions=False, query=False):
    # we always fetch loss because we will return it, we also
    # always fetch the final state because we need to pass it
    # along to the next batch in the loop below.
    # final state is now a list!! Update!! of three state tensors
    fetches = {
        'loss': self.loss,
        'final_state': self.final_state,
        'log␣loss' : self.log_loss,
       'loss␣before␣max': self.loss_before_max,
        'err␣wt␣reduce␣sum': self.err_wt_reduce_sum,
       'after␣tf.max␣div': self.after_max_division,
    }
```

```
    # we need to run the training op if we are doing training
    if self.train_op is not None:
        fetches['train_op'] = self.train_op

    # we need to fetch the network outputs if we are doing predictions
    if return_predictions:
        fetches['p'] = self.mixture_prob

    # run the initial state to feed the LSTM - this should just be
    # zeros
    state = session.run(self.initial_state)

    # we will sum up the total loss
    total_loss = 0.0

    all_outputs = []

    ################################################
    # for each batch:

    for step in range(self.model_input.epoch_size):

        for level in range(len(state)):
            # the input producer will take care of feeding in x/y,
            # but we need to feed in the LSTM state
            c, h = self.initial_state[level]
            feed_dict = { c: state[level].c, h: state[level].h }

            # run the computation graph?
            vals = session.run(fetches, feed_dict)

            # get the final LSTM state for the next iteration
            state = vals['final_state']

        # stash output if necessary
        if return_predictions:
            all_outputs.append(vals['p'])

        # update total loss
        total_loss += vals['loss']

    # do average
    total_loss /= self.loss_scale

    # return one or two things
    if not return_predictions:
        return total_loss
    elif query:
        return total_loss, vals['p']
    else:
        return total_loss, np.vstack(all_outputs)

def query(self, input_data, y, curr_state_list):

    return self.mixture_prob
```

```
def sample(self, session, duration=600):
    def sample_gaussian_2d(mu1, mu2, s1, s2, rho):
        mean = [mu1, mu2]
        cov = [[s1*s1, rho*s1*s2], [rho*s1*s2, s2*s2]]
        x = np.random.multivariate_normal(mean, cov, 1)
        return x[0][0], x[0][1]

    CHEAT = False

    if CHEAT:
        prev_x = np.zeros((4,1,3), dtype = np.float32)
        prev_x[0,0,2] = 1
        prev_x[:,0,0] = 2.5
        prev_x[:,0,1] = 5.5
        writing = np.zeros((duration,3), dtype = np.float32)
        prev_state = session.run(self.initial_state)
        fetches = [self.pis, self.corr, self.mu, self.sigma, self.eos, self.final_state]
        for i in range(duration):
            if i < 4:
                x_in = prev_x[i].reshape(-1,1,3)
            else:
                x_in = sample.reshape(-1,1,3)

            for level in range(len(prev_state)):
                c, h = self.initial_state[level]

                feed_dict = {self.lstm_input : x_in, c: prev_state[level].c,
                             h: prev_state[level].h }
                pis, corr, mu, sigma, eos, next_state = session.run(fetches, feed_dict)
            sample = gmm_sample(mu.reshape(-1,3,2), sigma.reshape(-1,3,2), corr, pis, eos)
            writing[i,:] = sample
            prev_state = next_state

    else:
        # first stroke
        prev_x = np.zeros((1,1,3), dtype=np.float32)
        prev_x[0,0,2] = 1 # we want to see the beginning of a new stroke

        # this will hold all the info
        writing = np.zeros((duration,3), dtype=np.float32)

        # this is a list of three states
        prev_state = session.run(self.initial_state)

        fetches = [self.pis, self.corr, self.mu, self.sigma, self.eos, self.final_state]

        for i in range(duration):
            print('At␣sample␣iteration:␣{}'.format(i))

            for level in range(len(prev_state)):

                c, h = self.initial_state[level]

                feed_dict = {self.lstm_input : prev_x, c: prev_state[level].c,
```

```
                                h: prev_state[level].h }
                    pis, corr, mu, sigma, eos, next_state = session.run(fetches, feed_dict)



                sample = gmm_sample(mu.reshape(-1,self.ncomponents,2),
                                    sigma.reshape(-1,self.ncomponents,2),
                                    corr, pis, eos)

                writing[i, :] = sample
                prev_x = sample.reshape(-1,1,3)
                prev_state = next_state

        return writing


########################################################################
# plot input vs predictions

def integrate(xyoffs, seq):

    # split up into subsequences
    n = xyoffs.shape[0]

    start_indices = np.nonzero(seq[:,0] == 0)[0]

    all_outputs = []

    for i, start_idx in enumerate(start_indices):
        if i + 1 < len(start_indices):
            end_idx = start_indices[i+1]
        else:
            end_idx = n
        xyslice = xyoffs[start_idx:end_idx]
        all_outputs.append(np.cumsum(xyslice, axis=0))

    return np.vstack(tuple(all_outputs))

def make_plot(epoch, loss, data, seq, pred):

    titlestr = '{} test set loss = {:.2f}'.format(epoch, loss)
    print(titlestr)

    y = seq[:,1] * 6

    if do_diff:
        data = integrate(data, seq)
        pred = integrate(pred, seq)

    plt.clf()
    plt.plot(data[:,0], data[:,1]+y, 'b.')
    plt.plot(pred[:,0], pred[:,1]+y, 'r.')
    plt.axis('equal')
    plt.title(titlestr)
    plt.savefig('test_data_pred_lstm_3.pdf')
```

```
def make_handwriting_plot(generated_data, generated_seq):
    titlestr = 'Generated␣Handwriting'
    if do_diff:
        data = integrate(generated_data, generated_seq)
    plt.clf()
    plt.plot(data[:,0], data[:,1], 'r.')
    plt.axis('equal')
    plt.title(titlestr)
    plt.savefig('GeneratedHW.pdf')

def make_heat_plot(epoch, loss, query_data, seq, xrng, yrng, xg, pred, i):
    p = pred.reshape(xg.shape)
    titlestr = '{}␣query␣set␣loss␣=␣{:.2f}'.format(epoch,loss)

    y = seq[:,1] * 6
    query_data = integrate(query_data, seq)
    last_point = query_data[-1]
    plt.clf()
    ax = plt.gca()
    xdata = xrng+last_point[0]
    ydata = -(yrng+last_point[1])
    plt.pcolormesh(xdata, ydata, p, cmap='jet')
    plt.plot(query_data[:,0], -query_data[:,1], 'wo', alpha = 0.90, markersize=3)
    plt.axis('equal')
    plt.axis([xdata.min(), xdata.max(), ydata.min(), ydata.max()])
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    plt.title(titlestr)
    plt.savefig('Gifs/LSTMHeatMap' + str(i) + '.pdf', bbox_inches='tight', pad_inches = 0)

def make_heat_plot_no_integrate(epoch, loss, query_data, xrng, yrng, xg, pred, i):
    p = pred.reshape(xg.shape)
    titlestr = '{}␣query␣set␣loss␣=␣{:.2f}'.format(epoch,loss)

    last_point = query_data[-1]
    plt.clf()
    ax = plt.gca()
    xdata = xrng+last_point[0]
    ydata = -(yrng+last_point[1])
    plt.pcolormesh(xdata, ydata, p, cmap='jet')
    plt.plot(query_data[:,0], -query_data[:,1], 'wo', alpha = 0.90, markersize=5)
    plt.axis('equal')
    plt.axis([xdata.min(), xdata.max(), ydata.min(), ydata.max()])
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    plt.title(titlestr)
    plt.savefig('NewGifs/LSTMHeatMap' + str(i) + '.pdf', bbox_inches='tight')

######################################################################
# main function

def main():

    # configs are just named tuples
```

```
Config = namedtuple('Config', 'batch_size,␣num_steps,␣hidden_size,␣keep_prob')

# generate training and test configurations
train_config = Config(batch_size=train_batch_size,
                      num_steps=train_num_steps,
                      hidden_size=hidden_size,
                      keep_prob=train_keep_prob)

test_config = Config(batch_size=1,
                     num_steps=1,
                     hidden_size=hidden_size,
                     keep_prob=1)

query_config = Config(batch_size= 1,
                      num_steps = 1,
                      hidden_size = hidden_size,
                      keep_prob = 1)

generate_config = Config(batch_size = 1,
                       num_steps = 1,
                       hidden_size = hidden_size,
                       keep_prob = 1)

# range to initialize all weights to
initializer = tf.random_uniform_initializer(-init_scale, init_scale)

# Import our handwriting data
data = DataLoader()

our_train_data = data.data[0:2000]
our_valid_data = data.valid_data[0:2000]
our_query_data = data.valid_data[225:227]

# generate our train data
train_data, train_seq = get_data(our_train_data)

# get our validation data
valid_data, valid_seq = get_data(our_valid_data)

# get the query data
query_data, query_seq = get_data(our_query_data)
query_data, query_seq = query_data[0:145, :], query_seq[0:145,:]

# Let's get our mesh grid for visualization
int_query_data = integrate(query_data, query_seq)
# int_query_y = query_seq[:,1] * 6
# itq = -int_query_data[:,1] + int_query_y

last_point = int_query_data[-1]
xmin, xmax = (int_query_data[:,0]-last_point[0]).min()-10,
              (int_query_data[:,0]-last_point[0]).max()+10

ymin, ymax = ((int_query_data[:,1]-last_point[1]).min()-10),
              ((int_query_data[:,1]-last_point[1]).max()+10)
```

```
print('xmin:␣{}␣xmax:␣{}␣\n␣ymin:␣{}␣ymax:␣{}'.format(xmin,xmax,ymin,ymax))

xrng = np.linspace(xmin, xmax, 200, True)
yrng = np.linspace(ymin, ymax, 200, True)

xg, yg = np.meshgrid(xrng, yrng)

xreshape, yreshape = xg.reshape(-1,1), yg.reshape(-1,1)
third_col = np.ones(xreshape.shape)

mesh_target = np.hstack([xreshape, yreshape, third_col])
mesh_target = mesh_target.reshape(-1, 1, 3).astype('float32')

# generate input producers and models -- again, not 100% sure why
# we do the name_scope here...
with tf.name_scope('train'):
    train_input = Input(train_data, train_seq, train_config)
    with tf.variable_scope('model', reuse=None, initializer=initializer):
        train_model = LSTMCascade(train_config, train_input, is_train=True)

with tf.name_scope('valid'):
    valid_input = Input(valid_data, valid_seq, train_config)
    with tf.variable_scope('model', reuse=True, initializer=initializer):
        valid_model = LSTMCascade(train_config, train_input, is_train=False)

with tf.name_scope('query'):
    query_input = Input(query_data, query_seq, query_config)
    with tf.variable_scope('model', reuse=True, initializer=initializer):
        query_model = LSTMCascade(query_config, query_input, is_train=False,
                                  external_targets=mesh_target)

prev_x = np.zeros((2,1,3), dtype = np.float32)
generate_data, generate_seq = get_data(prev_x)

with tf.name_scope('generate'):
    generate_input = Input(generate_data, generate_seq, generate_config)
    with tf.variable_scope('model', reuse=True, initializer=initializer):
        generate_model = LSTMCascade(generate_config, generate_input, is_sample=True,
                                     is_train=False)

if CREATE_GIFS:
    query_models = []
    for i in range(2,len(query_data)):
        with tf.name_scope('gif_query'+str(i)):
            seg_query_data = query_data[0:i,:]
            seg_query_seq = query_seq[0:i,:]

            int_seg_query_data = integrate(seg_query_data, seg_query_seq)

            last_point = int_seg_query_data[-1]
            xmin, xmax = (int_seg_query_data[:,0]-last_point[0]).min()-10,
                         (int_seg_query_data[:,0]-last_point[0]).max()+10

            ymin, ymax = ((int_seg_query_data[:,1]-last_point[1]).min()-10),
                         ((int_seg_query_data[:,1]-last_point[1]).max()+10)
```

```
            xrng = np.linspace(xmin, xmax, 200, True)
            yrng = np.linspace(ymin, ymax, 200, True)

            xg, yg = np.meshgrid(xrng, yrng)

            xreshape, yreshape = xg.reshape(-1,1), yg.reshape(-1,1)
            third_col = np.ones(xreshape.shape)

            mesh_target = np.hstack([xreshape, yreshape, third_col])
            mesh_target = mesh_target.reshape(-1, 1, 3).astype('float32')

            query_input = Input(seg_query_data, seg_query_seq, query_config)
            with tf.variable_scope('model', reuse=True, initializer=initializer):
                query_models.append(LSTMCascade(query_config, query_input,
                                    is_train=False, external_targets=mesh_target))

# print out all trainable variables:
tvars = tf.trainable_variables()
print('trainable␣variables:')
print('\n'.join(['␣␣-␣' + tvar.name for tvar in tvars]))

# create a session
session = tf.Session()

# # let's save our computation graph IF we don't already have a parameter
saver = tf.train.Saver()

# need to explicitly start the queue runners so the index variable
# doesn't hang
tf.train.start_queue_runners(session)

if len(sys.argv) > 1:

    saver.restore(session, sys.argv[1])

    print('Did␣a␣restore.␣Here␣are␣all␣the␣variables:')

    tvars = tf.global_variables()
    print('\n'.join(['␣␣-␣' + tvar.name for tvar in tvars]))

    if CREATE_GIFS:
        for idx, model in enumerate(query_models):
            int_query_data = integrate(model.model_input.posdata, model.model_input.seqinfo)
            last_point = int_query_data[-1]
            xmin, xmax = (int_query_data[:,0]-last_point[0]).min()-10,
                         (int_query_data[:,0]-last_point[0]).max()+10

            ymin, ymax = ((int_query_data[:,1]-last_point[1]).min()-10),
                         ((int_query_data[:,1]-last_point[1]).max()+10)

            xrng = np.linspace(xmin, xmax, 200, True)
            yrng = np.linspace(ymin, ymax, 200, True)
            l, pred = model.run_epoch(session,return_predictions=True, query=True)
            make_heat_plot_no_integrate('Model␣{}'.format(idx), l, int_query_data, xrng, yrng,
```

```
                                        xg, pred, idx)

        if GENERATE_HANDWRITING:
            strokes = generate_model.sample(session)
            seq = np.ones(shape = (strokes.shape[0], 1))
            seq[0,0] = 0
            make_handwriting_plot(strokes, seq)
            print('Handwriting␣generated.')

        if CREATE_TENSORBOARD:
            writer = tf.summary.FileWriter("tensorboard_output", session.graph)
            writer.close()

    else:

        # initialize all the variables
        session.run(tf.global_variables_initializer())

        # for each epoch
        for epoch in range(num_epochs):

            # run the epoch & get training loss
            l = train_model.run_epoch(session)
            print('training␣loss␣at␣epoch␣{}␣␣␣␣is␣{}'.format(epoch, l))
            if epoch % save_every == 0:

                print('Saving␣model.....␣')

                if not os.path.isdir('models'):
                    os.mkdir('models')

                written_path = saver.save(session, 'models/rnn_demo',
                          global_step=epoch)
                print('saved␣model␣to␣{}'.format(written_path))

            # see if we should do a printed/graphical update
            if epoch % update_every == 0:

                print()

                l = valid_model.run_epoch(session)
                print('validation␣loss␣at␣epoch␣{}␣is␣{:.2f}'.format(epoch, l))

                l, pred = query_model.run_epoch(session, return_predictions=True, query=True)

        written_path = saver.save(session, 'models/rnn_demo', global_step=num_epochs)
        print('saved␣final␣model␣to␣{}'.format(written_path))
        # do final update
        l, pred = query_model.run_epoch(session, return_predictions=True, query=True)
        make_heat_plot('final', l, query_data, xrng, yrng, xg, pred)

if __name__ == '__main__':

    main()
```

D.3 Data Loader Code

Data processing code used for this project predominantly comes from Otoro Design company and GitHub user *hardmaru* who was gracious enough to make it available to the public

```
import os
import numpy as np
import xml.etree.ElementTree as ET
import matplotlib.pyplot as plt
import pickle

class DataLoader():
  def __init__(self, batch_size=50, seq_length=300, scale_factor = 10, limit = 500):
    self.data_dir = "./Data"
    self.batch_size = batch_size
    self.seq_length = seq_length
    self.scale_factor = scale_factor # divide data by this factor
    self.limit = limit # removes large noisy gaps in the data

    data_file = os.path.join(self.data_dir, "strokes_training_data.cpkl")
    raw_data_dir = self.data_dir+"/lineStrokes"

    if not (os.path.exists(data_file)) :
        print("creating␣training␣data␣pkl␣file␣from␣raw␣source")
        self.preprocess(raw_data_dir, data_file)

    self.load_preprocessed(data_file)
    self.reset_batch_pointer()

  def preprocess(self, data_dir, data_file):
    # create data file from raw xml files from iam handwriting source.
    filelist = []
    # Set the directory you want to start from
    rootDir = data_dir
    for dirName, subdirList, fileList in os.walk(rootDir):
      for fname in fileList:
        filelist.append(dirName+"/"+fname)

    # function to read each individual xml file
    def getStrokes(filename):
      tree = ET.parse(filename)
      root = tree.getroot()

      result = []

      x_offset = 1e20
      y_offset = 1e20
      y_height = 0
      for i in range(1, 4):
        x_offset = min(x_offset, float(root[0][i].attrib['x']))
        y_offset = min(y_offset, float(root[0][i].attrib['y']))
        y_height = max(y_height, float(root[0][i].attrib['y']))
```

```
    y_height -= y_offset
    x_offset -= 100
    y_offset -= 100

    for stroke in root[1].findall('Stroke'):
      points = []
      for point in stroke.findall('Point'):
        points.append([float(point.attrib['x'])-x_offset,float(point.attrib['y'])-y_offset])
      result.append(points)

    return result

  # converts a list of arrays into a 2d numpy int16 array
  def convert_stroke_to_array(stroke):

    n_point = 0
    for i in range(len(stroke)):
      n_point += len(stroke[i])
    stroke_data = np.zeros((n_point, 3), dtype=np.int16)

    prev_x = 0
    prev_y = 0
    counter = 0

    for j in range(len(stroke)):
      for k in range(len(stroke[j])):
        stroke_data[counter, 0] = int(stroke[j][k][0]) - prev_x
        stroke_data[counter, 1] = int(stroke[j][k][1]) - prev_y
        prev_x = int(stroke[j][k][0])
        prev_y = int(stroke[j][k][1])
        stroke_data[counter, 2] = 0
        if (k == (len(stroke[j])-1)): # end of stroke
          stroke_data[counter, 2] = 1
        counter += 1
    return stroke_data

  # build stroke database of every xml file inside iam database
  strokes = []
  for i in range(len(filelist)):
    if (filelist[i][-3:] == 'xml'):
      print('processing␣'+filelist[i])
      strokes.append(convert_stroke_to_array(getStrokes(filelist[i])))

  f = open(data_file,"wb")
  pickle.dump(strokes, f, protocol=2)
  f.close()

def load_preprocessed(self, data_file):
  f = open(data_file,"rb")
  self.raw_data = pickle.load(f)
  f.close()

  # goes thru the list, and only keeps the text entries that have more than seq_length points
  self.data = []
  self.valid_data =[]
```

```
    counter = 0

    # every 1 in 20 (5%) will be used for validation data
    cur_data_counter = 0
    for data in self.raw_data:
      if len(data) > (self.seq_length+2):
        # removes large gaps from the data
        data = np.minimum(data, self.limit)
        data = np.maximum(data, -self.limit)
        data = np.array(data,dtype=np.float32)
        data[:,0:2] /= self.scale_factor
        cur_data_counter = cur_data_counter + 1
        if cur_data_counter % 20 == 0:
          self.valid_data.append(data)
        else:
          self.data.append(data)

  def validation_data(self):
    # returns validation data
    x_batch = []
    y_batch = []
    for i in range(self.batch_size):
      data = self.valid_data[i%len(self.valid_data)]
      idx = 0
      x_batch.append(np.copy(data[idx:idx+self.seq_length]))
      y_batch.append(np.copy(data[idx+1:idx+self.seq_length+1]))
    return x_batch, y_batch
```

D.4 GMM Sampling Code

GMM sampling was kindly written by our advisor Matt Zucker

```
import numpy as np
import matplotlib.pyplot as plt

# arrays have shapes
#  mu is n-by-k-by-2
#  sigma is n-by-k-by-2
#  rho is n-by-k
#  pi is n-by-k
def gmm_sample(mu, sigma, rho, pi, eos):

    ##################################################
    # verify shapes

    n, k = rho.shape
    assert mu.shape == (n, k, 2)
    assert sigma.shape == (n, k, 2)
    assert pi.shape == (n, k)
    assert eos.shape == (n,)

    ##################################################
```

```
    # choose a mixture component
    c = np.cumsum(pi, axis=1)
    r = np.random.random((n,1))

    # first index where r less than or equal to c
    mixture_comp = (r <= c).argmax(axis=1)

    ##################################################
    # get that component of mu, sigma, rho for each item

    idx = np.arange(n)

    mu = mu[idx, mixture_comp]
    sigma = sigma[idx, mixture_comp]
    rho = rho[idx, mixture_comp].reshape((-1, 1))

    ##################################################
    # do sampling

    s1 = sigma[:, 0].reshape((-1, 1))
    s2 = sigma[:, 1].reshape((-1, 1))

    a = s1
    b = rho*s2
    c = s2*np.sqrt(1.0 - rho**2)

    rx = np.random.normal(size=(n, 1))
    ry = np.random.normal(size=(n, 1))

    rx_prime = a * rx
    ry_prime = b * rx + c * ry

    r = np.hstack((rx_prime, ry_prime)) + mu

    eos_samples = (np.random.random((n,1)) <= eos).astype(np.float32)

    return np.hstack((r, eos_samples))

# arrays have shapes
#   mu is k-by-2
#   sigma is k-by-2
#   rho is (k,)
#   pi is (k,)
#   eos is a scalar
#   x is n-by-3 (x, y, eos)
def gmm_eval(mu, sigma, rho, pi, eos, x):

    ##################################################
    # verify shapes
    k = rho.shape[0]
    n = x.shape[0]
    assert mu.shape == (k, 2)
    assert sigma.shape == (k, 2)
    assert pi.shape == (k,)
    assert x.shape == (n, 3)
```

```
    assert np.isscalar(eos)

    ################################################
    # do stuff from Graves paper to evaluate probs.

    dev = x[:, None, :2] - mu[None, :, :]
    var = sigma**2
    s1s2 = sigma.prod(axis=1)

    z12_before = dev**2 / var
    z12 = z12_before.sum(axis=2)

    reduce_dev = dev.prod(axis=2)
    z3 = (reduce_dev*rho*2) / (s1s2)
    Z = z12 - z3

    normalizer = (2.0 * np.pi * s1s2) * np.sqrt(1.0-rho**2)

    expon_part = np.exp(-Z / (2.0 * (1.0 - rho**2)))

    N = expon_part / normalizer

    p = (N * pi).sum(axis=1)

    x3 = x[:,2]
    p_eos = eos * x3 + (1.0-eos) * (1.0-x3)

    return p * p_eos

# ######################################################################
```

Bibliography

Bishop, C. M. (1994). Mixture density networks. Technical report.

Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

Hardmaru (2017). hardmaru/write-rnn-tensorflow.

Olah, C. (2015). Understanding lstm networks. *Colah's Blog*.

Olah, C. and Carter, S. (2016). Attention and augmented recurrent neural networks. *Distill*.

Otoro (2015a). Otoro design.

Otoro (2015b). Otoro design.

Wikipedia (2017a). Artificial neural network — wikipedia, the free encyclopedia. [Online; accessed 4-May-2017].

Wikipedia (2017b). Expectation–maximization algorithm — wikipedia, the free encyclopedia. [Online; accessed 3-May-2017].