ECE-GY 6143 Final Project - Fall 2020

(Applications of Deep Learning: Generating Images)

Neeraja Narayanswamy (nn2108) & **Sahil Makwane** (sm9127)

---

1. INTRODUCTION

1.1 What is MNIST?

The MNIST dataset is a 60,000 training and 10,000 testing sample database available as a subset of the larger set from NIST[2]. The database has size normalized and centered digits. The database seeks to assist programmers learn techniques of pattern-recognition on data from the world without spending too much time on image processing. The reason we chose MNIST is because it provides us the opportunity to input data as just a set of pixel values thus making it feasible for us to focus on implementing the details[6].

Figure 1.1: sample from MNIST Dataset

1.2 What are Recurrent Neural Networks?

Recurrent neural networks or RNNs are a certain type of class of neural networks which permit the previous-outputs to be utilized as the inputs and simultaneously have hidden states. Basically, RNN inputs are the current-inputs as well as the inputs they perceived previously. This characteristic allows recurrent neural networks to have a temporal dynamic behavior. This network architecture is directly based on the feed-forward neural networks. The most common use case of Recurrent neural networks is the processing of variable length sequences of inputs, thus, allowing them the ease of accessibility for tasks such as handwriting-recognition.

The recurrent neurons are responsible for storing the inputs at the previous step and merging that information along with the inputs in the present-step. The decisions at time "t-1" directly influence the decisions taken at time "t". Thus, It can be said the neurons, this merging of information allows correlation between current and previous data-steps. All of this is extremely congruous to the manner in which people make decisions in real life: combining the data from the present environment and merging it with the data from the past (recent or not) to make a particular decision. Clearly demonstrate the synonymous nature of programmed neural networks and the actual decision-making process in our brains.

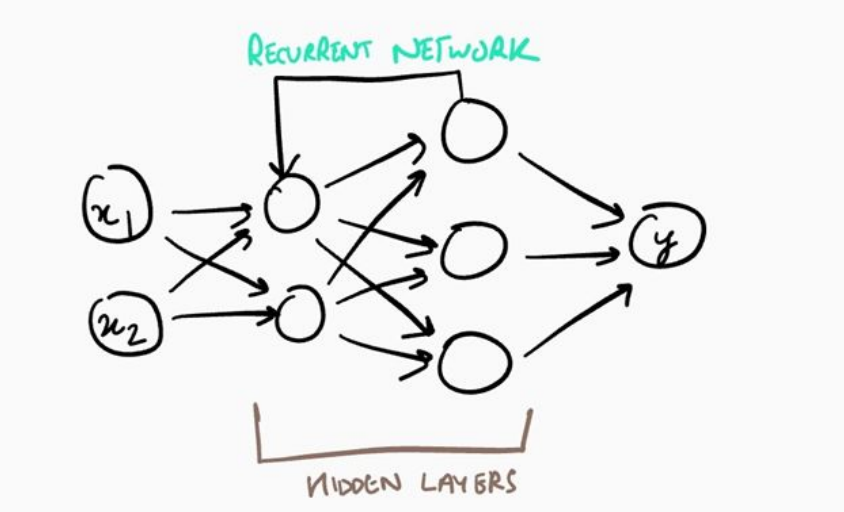

Figure 1.2: Recurrent Neural Network Architecture (x1 and x2 are inputs while y is the output)

Advantages	Drawbacks
• Possibility of processing input of any length • Model size not increasing with size of input • Computation takes into account historical information • Weights are shared across time	• Computation being slow • Difficulty of accessing information from a long time ago • Cannot consider any future input for the current state

Figure 1.3: Source: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

1.3 What is bi-directional recurrent neural network?

Not always does a recurrent neural network work efficiently by reading the past, at times, the future needs to be used to correct the past. In tasks such as handwriting recognition, ambiguity can be present based on just one given portion of the input. Thus, to better understand the context we need to comprehend the upcoming inputs to detect the present inputs.

BRNN or Bidirectional Recurrent Neural Networks are used for connecting 2 hidden-layers of directions which are opposite to each other to the very same output for upping the input-information amount available for processing to the network. BRNN gets information from the previous and following states at

the same time, does not require fixing of the input data and the information of the upcoming (future) input is available to the present state.

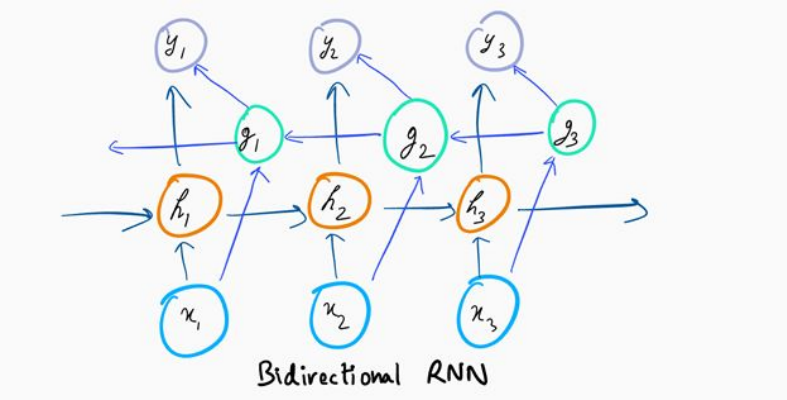

Figure 1.4: Bidirectional Neural Network

Bidirectional Neural Network breaks neurons of a usual Recurrent Neural Network into 2-directions (a positive-time direction, forward state and a negative-time direction, backward state). The inputs of the opposite-direction states are disconnected from the outputs of the 2 states. 2 time directions permit the information of input from the past-future of the present fram to be used (unlike in regular Recurrent Neural Networks)[7].

1.4. What is LSTM (Long Short Term Memory) Networks?

LSTMs (Long Short Term Memory networks) are capable of learning long-term dependencies and add changes as to how the outputs and the hidden states are computed using the inputs. They are not a separate variant of the architecture of recurrent neural networks. It is in their behavior to learn information for long periods of time[8]. Unlike the usual recurrent neural networks, LSTM have a different structure for the repeating module but a similar chain-like-structure. The LSTM adds/removes data from the cell state by employing structures called 'gates'.

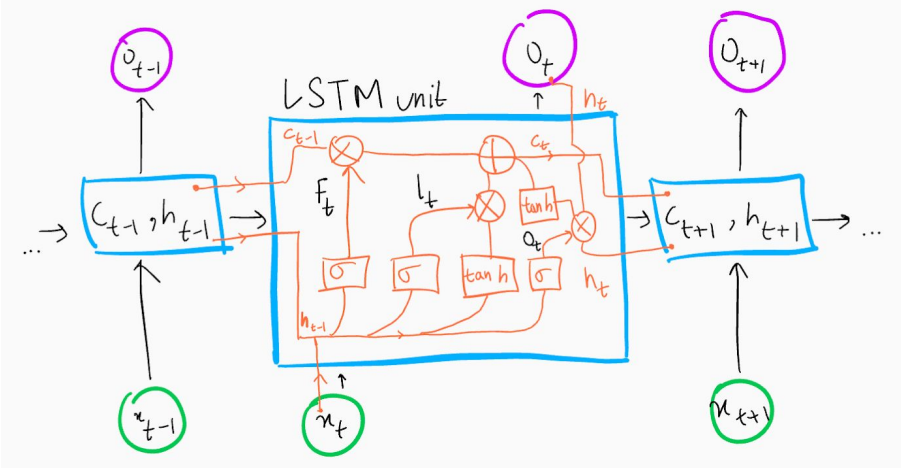

Figure 1.5 Long Short Term Memory Network

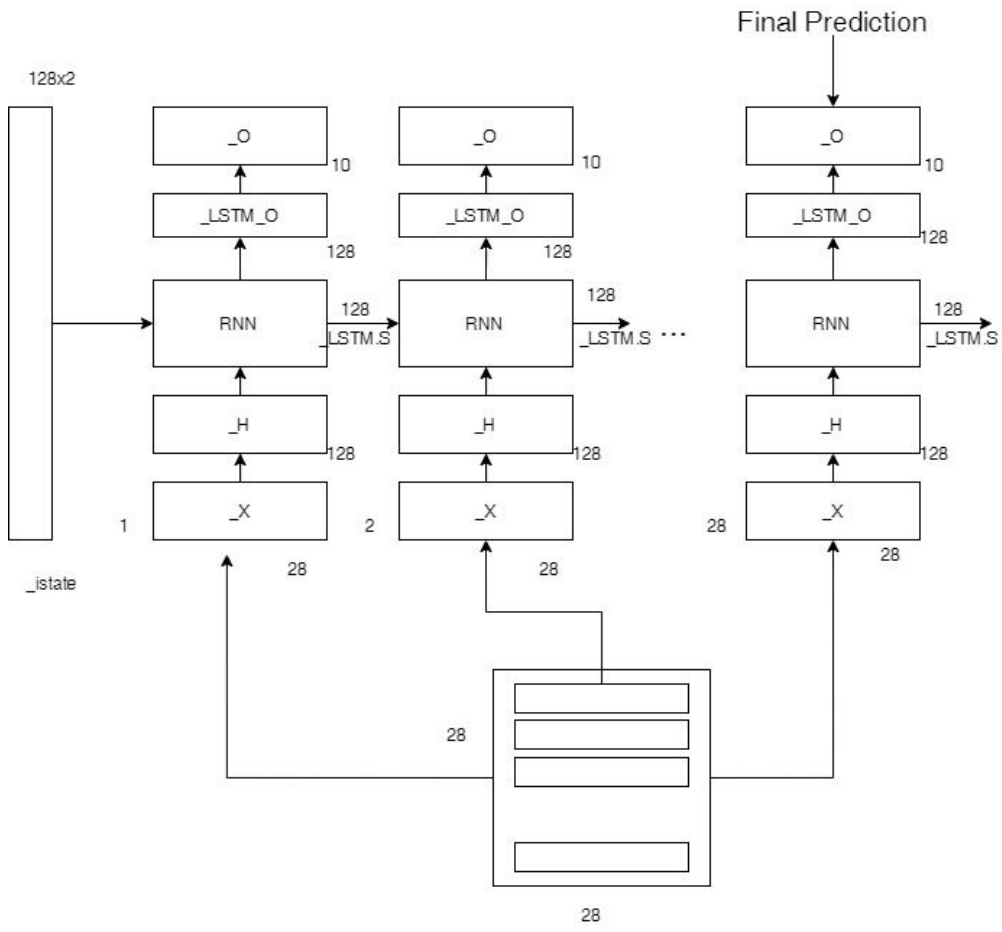

Figure 1.6 Row by row sequential MNIST

Applying RNN on 28 by 28 images from the MNIST dataset in the following 2 ways:

1. Row by row: The cells in the recurrent neural network are looking at a vector of size-28 with total time-steps being 28.
2. Pixel by Pixel: The cells are looking at the i-th pixel in i-th step with total steps being 28*28.

Case 2 is tougher as the model needs to store a long term memory. The speed of the LSTM layer can be increased by using CudnnGRU instead of running long sequences of case 2 approach which can bring the performance down[9].

1.6. What is Sketch RNN?

Sketch RNN[1] is a GAN based model that is capable of drawing a sketch given an input of hand drawn images. The basic structure of the model is the combination of an encoder and decoder. The role of the encoder is to determine the coordinates and the dimensions of the strokes. The dimensions being the starting point,the ending point and whether this is the final stroke of the sketch. The role of the decoder is

to sample an output sketch on a given vector. The way they achieve their goal is by their structure. The encoder is a bi-directional RNN that takes in the sketch as an input and outputs a vector. The two inputs of the bi directional rnn is the sketch sequence and the sequence in reverse. The decoder is an auto regressive rnn that samples the output sketches based on the vector given by the encoder.

The decoder in itself can be a stand alone model which can be used for unconditional image generation based on the step before it. The model is trained to optimize two loss functions :
The reconstruction loss and the kullback leibler divergence loss. The reconstruction loss is measured on the parameters of the vector while the kullback loss is measured on the divergence of the vector. The goal is to ensure there is zero mean and unit variance.

$$Loss = L_R + w_{KL} L_{KL} \quad [1]$$

As a special case, we can also train our model to generate sketches unconditionally, where we only train the decoder RNN module, without any input or latent vectors. By removing the encoder, the decoder RNN as a standalone model is an autoregressive model without latent variables. In this use case, the initial hidden states and cell states of the decoder RNN are initialized to zero. The inputs xi of the decoder RNN at each time step is only Si−1 or S 0 i−1 , as we do not need to concatenate a latent vector z. In Figure 3, we sample various sketch images generated unconditionally by varying the temperature parameter from $\tau = 0.2$ at the top in blue, to $\tau = 0.9$ at the bottom in red.

1.7. Why Sketch RNN for Image classification?

We wanted to train our network on the MNIST dataset. The MNIST is a set of images of hand drawn digits that are 28x28 pixels in size. The reason we chose to train MNIST specifically was because when we initially trained an image classification network for our Intro to ML assignment we noticed that there were some inconsistencies with the actual results of digits written by us and the digits the MNIST identified correctly(Homework 5- Regularization). This was because the algorithm we chose used a methodology in which a heatmap would be generated for digits 0 to 9 and the digit whose heat map would be the closest match of the test image would be the digit the network identifies as the image.

This caused error in our real time objects because not everyone writes the same. For example the way one of us wrote the number 2 for the assignment has the line of the two slightly lower than the images of two the network was trained on, therefore this image of 2 was classified as 7 instead.

The sketch rnn solves this problem as it tries to understand how these digits are made by analyzing each vector of the image. The sketch RNN would take the dataset of images of 2 and try to understand how the number 2 is generated rather than building a heatmap over those images. This is similar to how humans learn to interpret and draw images,therefore we believe an image classifier which has a sketch rnn would be more powerful than a regular image classifier.

The sketch RNN can also be used to generate images for the classifier to train on which is what our approach to the problem was. This intention of generating digits using the sketch rnn was so that we can use the generated images to expand on the existing MNIST dataset without using human effort. This would in turn make image classification algorithms more accurate as they would have a more robust

dataset to work with. This approach can also be applied to other domains of image classification such as letter recognition between different kinds of handwritings or an artwork classifier for different types of paintings.

2. METHODOLOGY & RESULTS

We first downloaded the magenta library on our system and two colab notebooks. The first colab notebook was to download and format the mnist dataset and the second was to replicate the official magenta code. A third notebook that does not use magenta was also created.

In the first notebook we downloaded the open source MNIST dataset and loaded it into numpy arrays of various shapes until we found one that was compatible. The dataset was already split into train validation and test sets which was required by the code we needed to generate the training set.[4]

```
print('Check the dimenssions')
print('X_train.shape:', X_train.shape)
print('Y_train.shape:', Y_train.shape)
print('X_validation.shape:', X_validation.shape)
print('X_validation.shape:', X_validation.shape)
print('X_test.shape:', X_test.shape)
print('Y_test.shape:', Y_test.shape)
print('train_data.shape:', train_data.shape)
print('train_data1.shape:', train_data1.shape)
print('validation_data.shape:', validation_data.shape)
print('validation_data1.shape:', validation_data1.shape)
print('test_data.shape:', test_data.shape)
print('test_data1.shape:', test_data1.shape)
```

```
Check the dimenssions
X_train.shape: (55000, 28, 28)
Y_train.shape: (55000, 10)
X_validation.shape: (5000, 28, 28)
X_validation.shape: (5000, 28, 28)
X_test.shape: (10000, 28, 28)
Y_test.shape: (10000, 10)
train_data.shape: (43670000,)
train_data1.shape: (55000, 784)
validation_data.shape: (3970000,)
validation_data1.shape: (5000, 784)
test_data.shape: (7940000,)
test_data1.shape: (10000, 784)
```

Figure 2.1: Analyzing the different ways the MNIST dataset can be put into by numpy arrays

We then used a command available in the magenta library to save those arrays in the npz file which would then be used by the program that creates the training dataset. Our dataset had 55000 training images,5000 validation images and 5000 test images.[4]

```
reshapetostroke_train = mnist.train.images.reshape([28,28,-1])
print(reshapetostroke_train.shape)
reshapetostroke_valid = mnist.validation.images.reshape([28,28,-1])
print(reshapetostroke_valid.shape)
reshapetostroke_test = mnist.validation.images.reshape([28,28,-1])
print(reshapetostroke_test.shape)
```

```
(28, 28, 55000)
(28, 28, 5000)
(28, 28, 5000)
```

Saving the file in the correct format

```
filename = os.path.join('/content', 'mnist_lstmv6.npz')
np.savez_compressed(filename, train=reshapetostroke_train, valid=reshapetostroke_valid, test=reshapetostroke_test)
```

Figure 2.2: Saving the data in the required format

For the training we downloaded the sketch rnn files from the magenta github repository into our system and changed the code to suit our requirements. For example we reduced the number of iterations from 10,000,000 to 50,000 the 10,000,000 iterations come from the size of the model but due to the time and machine constraints we decided it was best to do only 50,000 iterations.

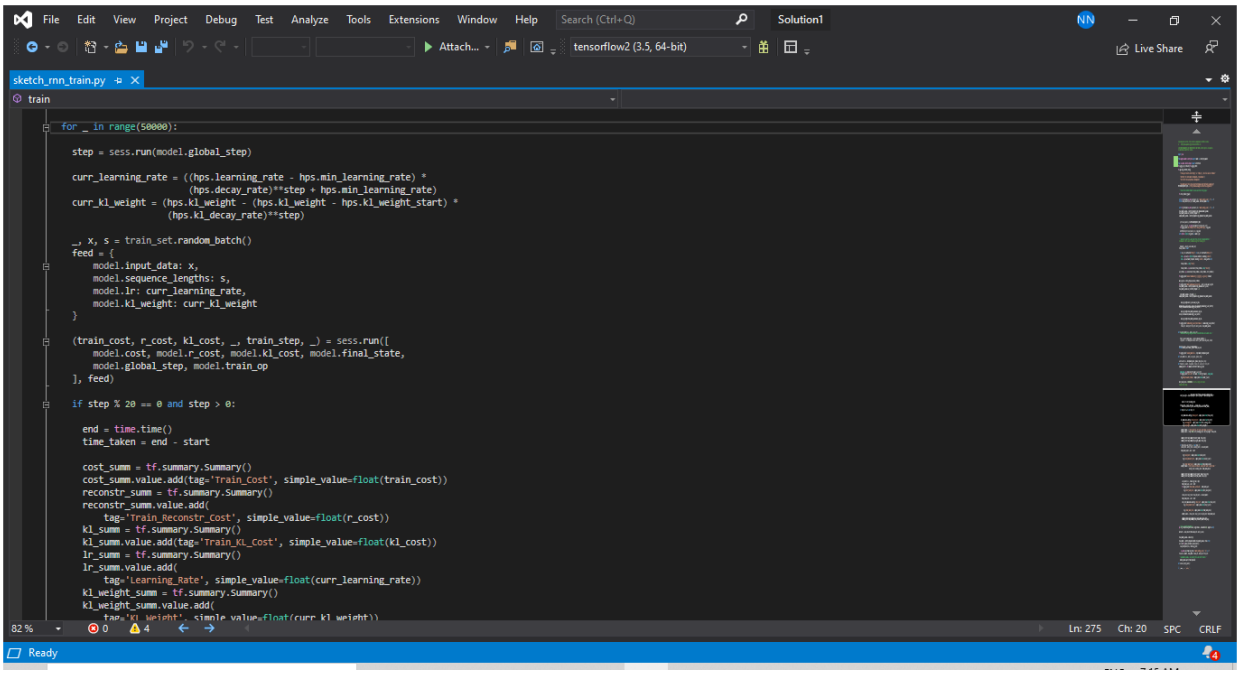

Figure 2.3:Sketch RNN training Algorithm

The training algorithm returns the cost at every 20 steps and it hopes to optimize the model by minimizing the cost which is based on the loss function stated above.At every 500 steps the model checks the and saves the validation cost if that cost was better than the one saved on the previous 500 step checkpoint. [3]

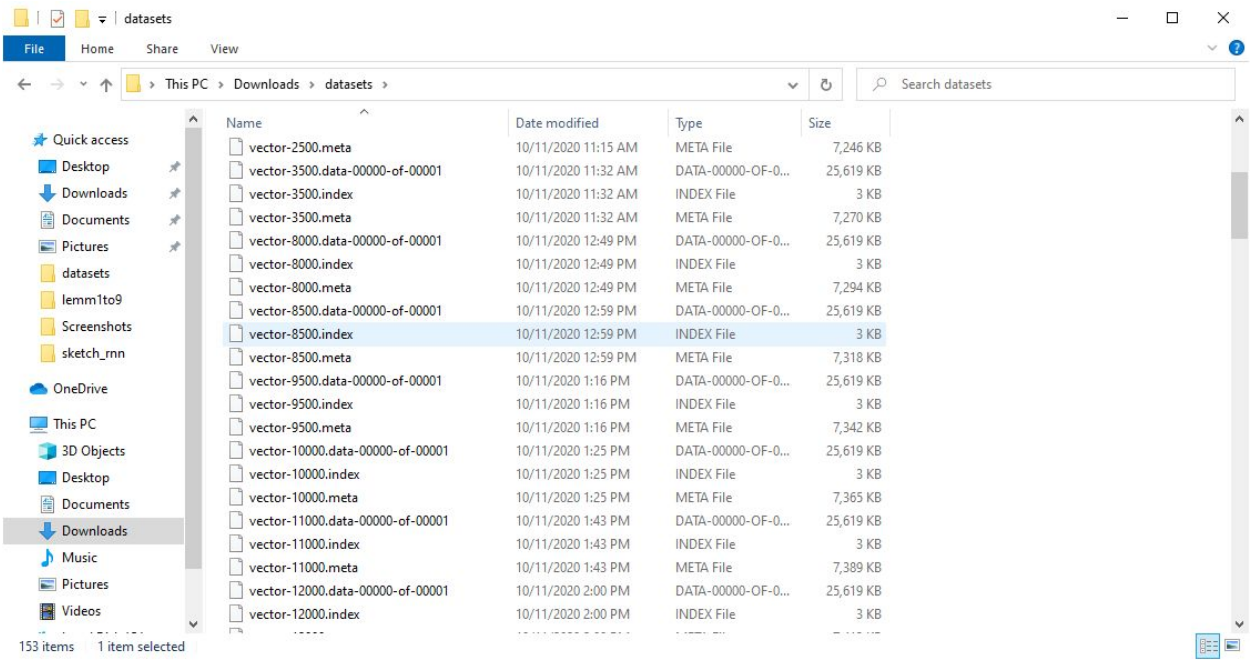

Figure 2.4: Checkpoints with best validation costs saved

After training we finally get a json folder describing all the parameters used for training such as the base encoder and decoder model and size which in our case was the lstm for both encoders and the encoder had a network size of 256 neurons and the decoder had a size of 512 neurons,the learning rate and the total number of iterations it is trained on.[3]


```
{
  "data_set": [ "mnist_lstmv2.npz" ],
  "dec_model": "lstm",
  "dec_rnn_size": 512,
  "enc_model": "lstm",
  "enc_rnn_size": 256,
  "min_learning_rate": 0.00001,
  "output_dropout_prob": 0.9,
  "kl_weight_start": 0.01,
  "num_mixture": 20,
  "is_training": 1,
  "learning_rate": 0.001,
  "decay_rate": 0.9999,
  "use_recurrent_dropout": 1,
  "recurrent_dropout_prob": 0.75,
  "batch_size": 100,
  "input_dropout_prob": 0.9,
  "augment_stroke_prob": 0.15,
  "use_input_dropout": 0,
  "z_size": 128,
  "save_every": 500,
  "use_output_dropout": 0,
  "kl_weight": 0.5,
  "kl_tolerance": 0.2,
  "random_scale_factor": 0.15,
  "conditional": 1,
  "kl_decay_rate": 0.99995,
  "grad_clip": 1.0,
  "num_steps": 50000,
  "max_seq_len": 250
}
```


Figure 2.5 : Configurations of the trained model

We then save this configuration file along with the checkpoint and the vectors of the last best validation cost into a folder which we upload on google drive. We then mount the google drive on our colab notebook and run the model on the reconstructed magenta notebook.[3]

2.2 RESULTS

We were successfully able to replicate the results of the sheep,catbus,owl and pig models that were pre-trained on the dataset. We were also successfully able to make the code generate an image of 1's.

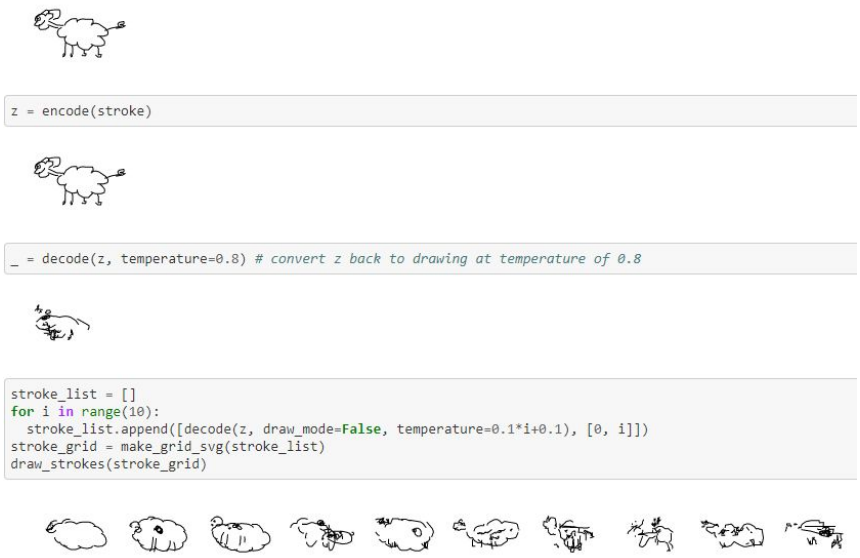

Figure 2.6: Sheep Images generated via a pretrained model

The image of 1's generated proves that a sketch rnn can learn digit recognition the same way humans can

Figure 2.7 : Sketch RNN generating 1's using our model

As we can see with the style the 1's are generated are similar to the way different people write 1, if we look closely we can see that some ones are written as '|' while others are written as '1'. This proves that we can use SketchRNN to generate a wide subset of images that nearly match how humans write and use this data to feed image classification algorithms for a more accurate result.

3. CONCLUSION

We can therefore conclude that the SketchRNN Classifier can successfully generate more image samples for digit classification which in turn can help improve algorithms that recognize handwritten digits which would be very useful to the algorithms of natural language processing. We now understand how different people draw different images for the same object[5] and how sketch RNN can help us identify the object by its unique method of image classification and generation.

4. APPLICATIONS AND FUTURE WORK

Sketch RNN has countless number of artistic and creative applications. The decoder-only model trained on various classes can be beneficial to aiding the work of artists by recommending several ways in which sketches can be finished. One of the significant use cases can be to generate aesthetic looking sketches out of bad drawings[1]. The MNIST handwriting model can assist kids in learning how to write the numbers. We can incorporate various kinds of user feedback in the form of rating data into the training process.

REFERENCES

[1] A Neural Representation of Sketch Drawings - https://arxiv.org/pdf/1704.03477.pdf

[2] Mnist database - http://yann.lecun.com/exdb/mnist/

[3] Magenta Model - https://github.com/magenta/magenta/tree/master/magenta/models/sketch_rnn

[4] Loading and preparing the MNIST Dataset - https://medium.com/the-artificial-impostor/notes-understanding-tensorflow-part-2-f7e5ece849f5

[5] Understanding how different people draw different images for the same object - https://medium.com/analytics-vidhya/analyzing-sketches-around-the-world-with-sketch-rnn-c6cbe9b5ac80

[6] Understanding LSTM in Tensorflow - https://jasdeep06.github.io/posts/Understanding-LSTM-in-Tensorflow-MNIST

[7] Bidirectional Recurrent Neural Networks - https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks

[8] Understanding LSTMs - http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[9] Understanding Tensorflow - https://medium.com/the-artificial-impostor/notes-understanding-tensorflow-part-2-f7e5ece849f5