Sequence Prediction using RNN

TEAM 57:
Priyansh Gupta (2019101080)
Harshit Sharma (2019101083)
Shreyash Rai (2019101096)

Documentation

Intro

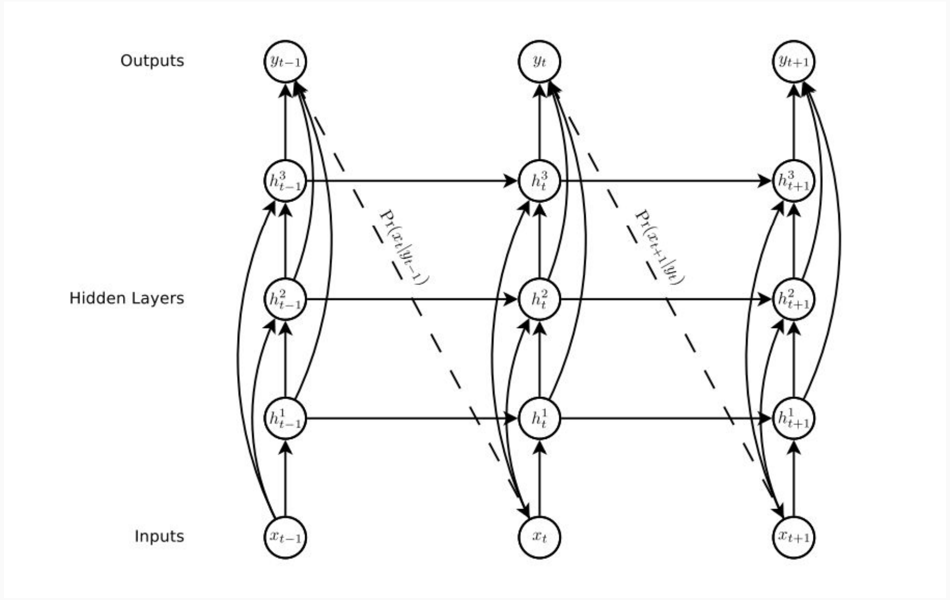
RNNs can be trained for sequence generation by processing real data sequences one step at a time and predicting what comes next. Assuming the predictions are probabilistic, novel sequences can be generated from a trained network by iteratively sampling from the network’s output distribution, then feeding in the sample as input at the next step.

Structure of basic RNN

For our problems, we will send a N length sequence as an input to the rnn and then will find P(Xt | Yt-1).

After this we can use our loss function as:

- log(P(Xt | Yt-1)).



First Problem - Text Prediction

So the target is that for a given input incomplete sentence X, we want to predict and complete the sentence. For the mid-evals we reduced the problem to predicting the next word in an incomplete sentence (which is an easier version of original problem).

What we implemented

So we have trained a network on large text corpus(English language). We have used 5-grams as input to our neural network. The neural network has very basic structure as:

1st layer - Embedding layer(word2vec)

2nd layer - LSTM layer

3rd layer - Linear followed by softmax

Perplexity scores

We found the perplexity scores of each sentence in training and testing corpus.

Perplexity is a well known metric which is used for language modelling.

The resultant perplexities are stored in text files which you can check on github repo.

Average train perplexity = 2.593260516860685

Average test perplexity = 75.56257655594742

Challenges Faced

Vocab is quite large, so had to reduce it by replacing less frequent words with <unknown> tag. Now in case of predicting <unknown>, we have to output a random word as output.

This kind of model requires very heavy training (for 1-2 days on a gpu).Unfortunately we have certain constraints on the machine that we can use and also on time considering course timeline.

Once we increased the size of n_grams, we started facing problem of gradient exploding, which we finally solved by clipping parameters. But again large sequences took lot of computation and took a lot of time for even one epoch on cpu.

Problem 2 - Online Handwriting prediction

We can apply RNN to online handwriting data (online in this context means that the writing is recorded as a sequence of pen-tip locations, as opposed to offline handwriting, where only the page images are available) and using this we can predict the next pen strokes.

If we combine a cluster of pen strokes(say 25), then we may be able to draw a whole character(by predicting).

Database

IAM-ondb database is used by writers, so we are also going to use it.

IAM-OnDB consists of handwritten lines collected from 221 different writers using a ‘smart whiteboard’.

How does my input looks like

So the input vector will have 3 values -> {x, y, flag}.

x-> x-offset of current pen stroke with respect to previous stroke.

y-> y-offset (similar to x)

Flag - > it is either 0 or 1. 1 means that the pen will be lifted after this stroke and otherwise 0.

We will feed sequence of 100 or 400 vectors in our model as input(LSTM).

Normalizing input sequence

It is very important to normalize input sequence by using transformation:-

X^1 = (X- mean(X)) / SD(X).

How the next vector will be predicted

So x and y of next vector will be predicted using a mixture of 20 bivariate gaussian.

So for each gaussian we need 6 parameters, so for 20 we will need 120 in total.

Flag of next vector will be predicted by one of the output neuron of model.

So total output neurons should be equal to = 121

Different activations for each output neuron

$$e_t = \frac{1}{1 + \exp\left(\hat{e}_t\right)} \qquad \Longrightarrow e_t \in (0,1)$$

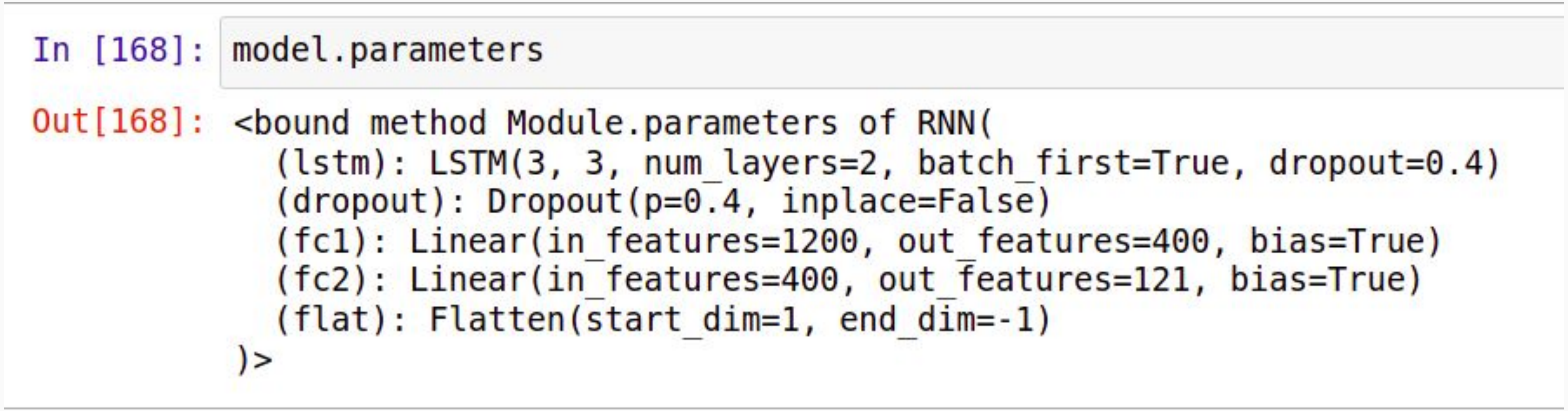
$$\pi_t^j = \frac{\exp\left(\hat{\pi}_t^j\right)}{\sum_{j'=1}^{M} \exp\left(\hat{\pi}_t^{j'}\right)} \qquad \Longrightarrow \pi_t^j \in (0,1), \quad \sum_j \pi_t^j = 1$$

$$\mu_t^j = \hat{\mu}_t^j \qquad \Longrightarrow \mu_t^j \in \mathbb{R}$$

$$\sigma_t^j = \exp\left(\hat{\sigma}_t^j\right) \qquad \Longrightarrow \sigma_t^j > 0$$

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

Model (First Try)



```
In [168]: model.parameters

Out[168]: <bound method Module.parameters of RNN(
            (lstm): LSTM(3, 3, num_layers=2, batch_first=True, dropout=0.4)
            (dropout): Dropout(p=0.4, inplace=False)
            (fc1): Linear(in_features=1200, out_features=400, bias=True)
            (fc2): Linear(in_features=400, out_features=121, bias=True)
            (flat): Flatten(start_dim=1, end_dim=-1)
          )>
```

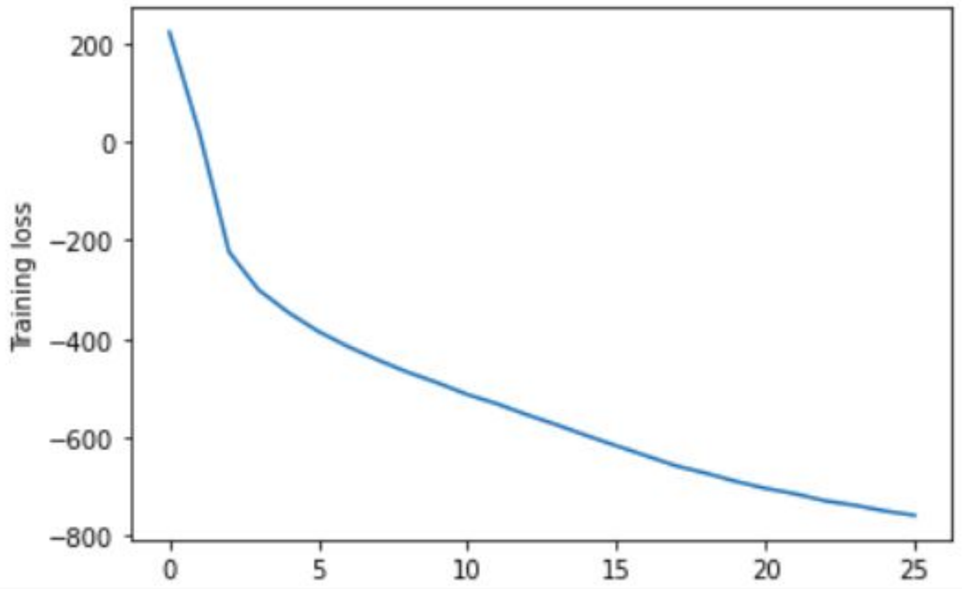
First attempt of training

Why negative loss which is

Continuously decreasing?

What's the problem here.

```
Text(0, 0.5, 'Training loss')
```



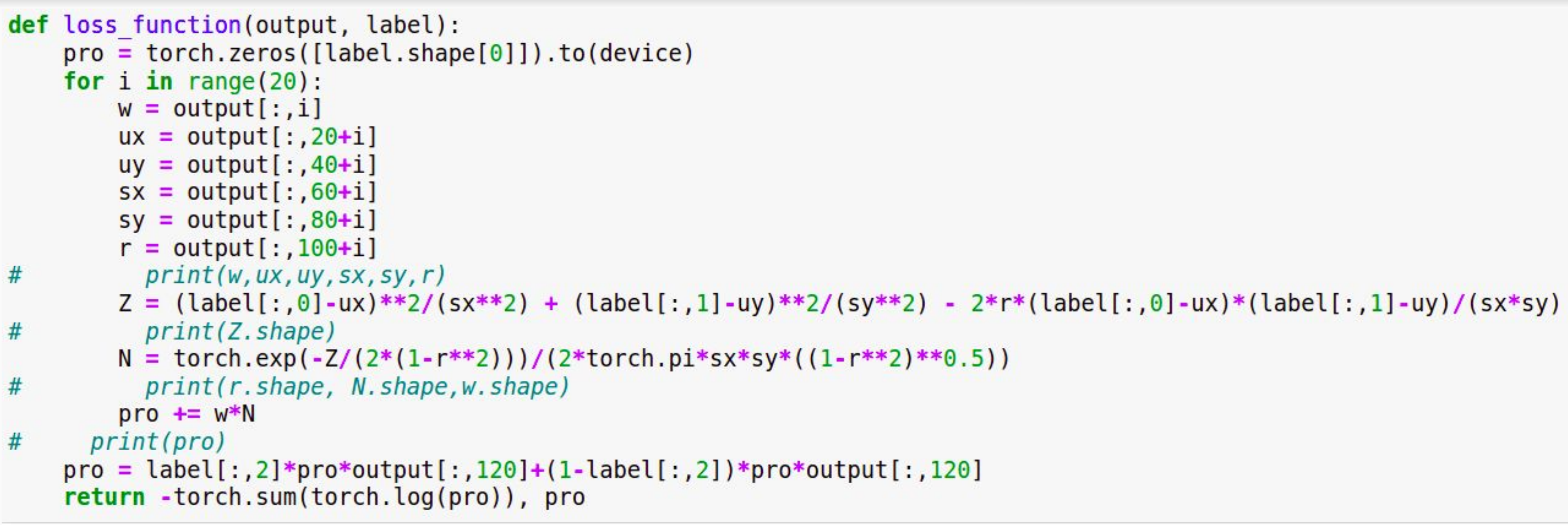
Second Attempt (after clipping)

Even after clipping many probability outputs were getting value greater than 1.

In this case the model parameters and output were not even nan(inf).

The only reason, we could figure out is precision. So we are performing lot of complex maths functions in loss function while calculating probability which may had lead to this problem.

Loss Function

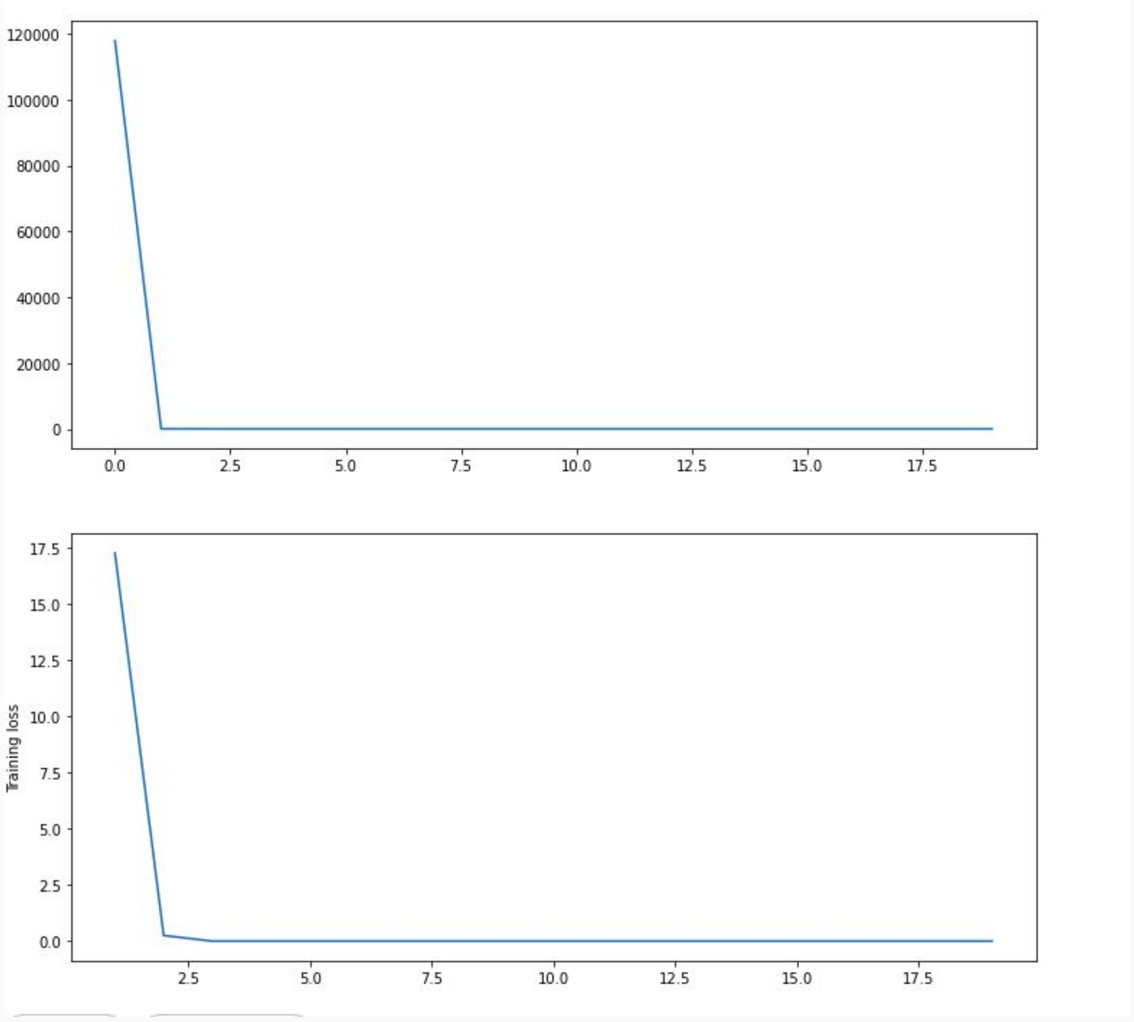


```
def loss_function(output, label):
    pro = torch.zeros([label.shape[0]]).to(device)
    for i in range(20):
        w = output[:,i]
        ux = output[:,20+i]
        uy = output[:,40+i]
        sx = output[:,60+i]
        sy = output[:,80+i]
        r = output[:,100+i]
#         print(w,ux,uy,sx,sy,r)
        Z = (label[:,0]-ux)**2/(sx**2) + (label[:,1]-uy)**2/(sy**2) - 2*r*(label[:,0]-ux)*(label[:,1]-uy)/(sx*sy)
#         print(Z.shape)
        N = torch.exp(-Z/(2*(1-r**2)))/(2*torch.pi*sx*sy*((1-r**2)**0.5))
#         print(r.shape, N.shape,w.shape)
        pro += w*N
#     print(pro)
    pro = label[:,2]*pro*output[:,120]+(1-label[:,2])*pro*output[:,120]
    return -torch.sum(torch.log(pro)), pro
```

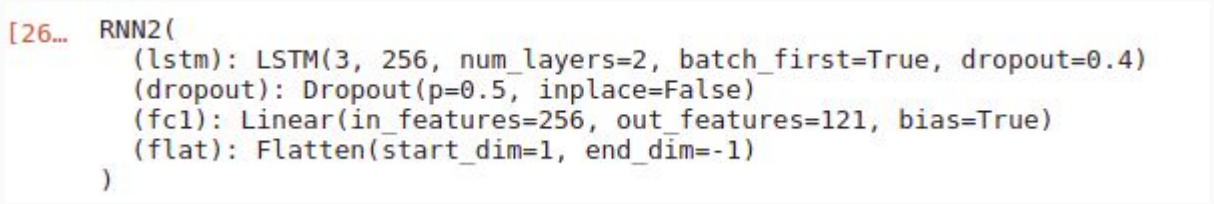
Loss curves

There is definitely something wrong with the choice of model only.

So we decided to change the model structure.



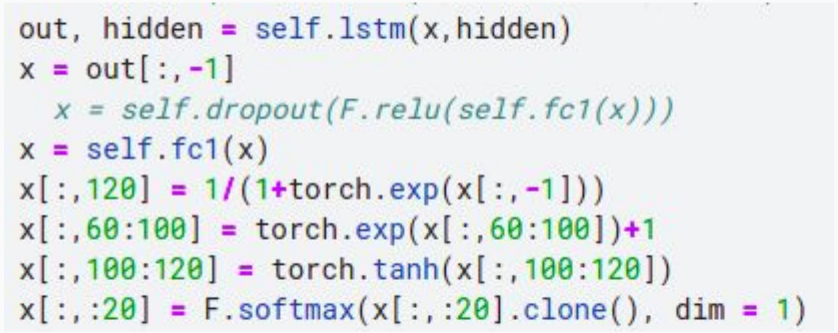
Model-2



```
[26… RNN2(
       (lstm): LSTM(3, 256, num_layers=2, batch_first=True, dropout=0.4)
       (dropout): Dropout(p=0.5, inplace=False)
       (fc1): Linear(in_features=256, out_features=121, bias=True)
       (flat): Flatten(start_dim=1, end_dim=-1)
     )
```

Forward->

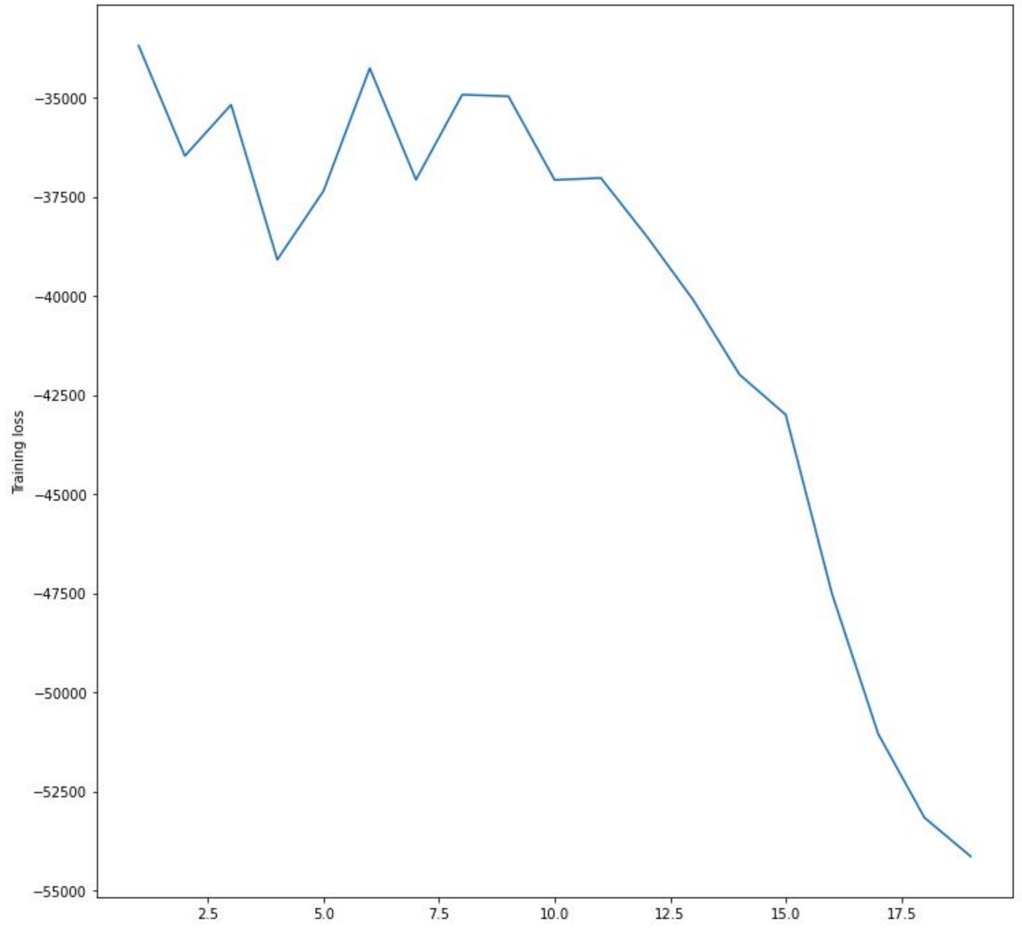
function->



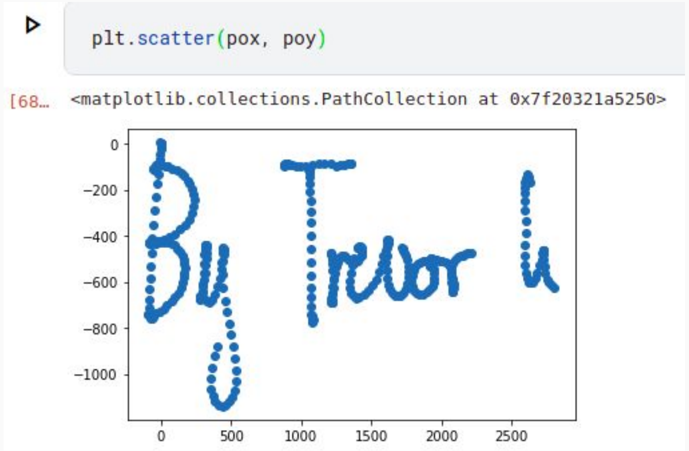
```
out, hidden = self.lstm(x,hidden)
x = out[:,-1]
  x = self.dropout(F.relu(self.fc1(x)))
x = self.fc1(x)
x[:,120] = 1/(1+torch.exp(x[:,-1]))
x[:,60:100] = torch.exp(x[:,60:100])+1
x[:,100:120] = torch.tanh(x[:,100:120])
x[:,:20] = F.softmax(x[:,:20].clone(), dim = 1)
```

Training Loss curve

->

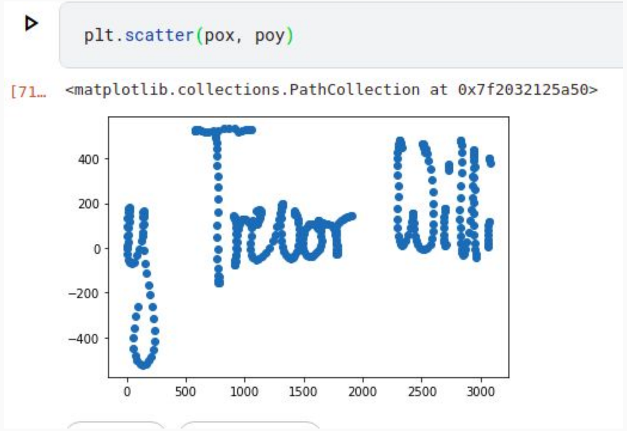


Outputs that we wished for->



```
plt.scatter(pox, poy)
```

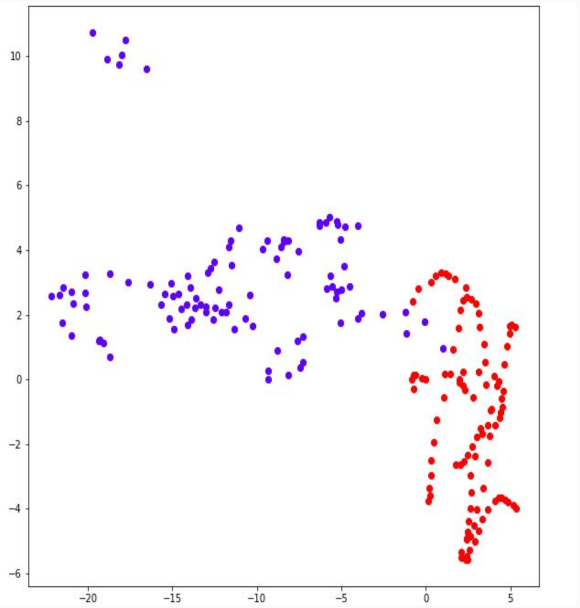
```
[68… <matplotlib.collections.PathCollection at 0x7f20321a5250>
```

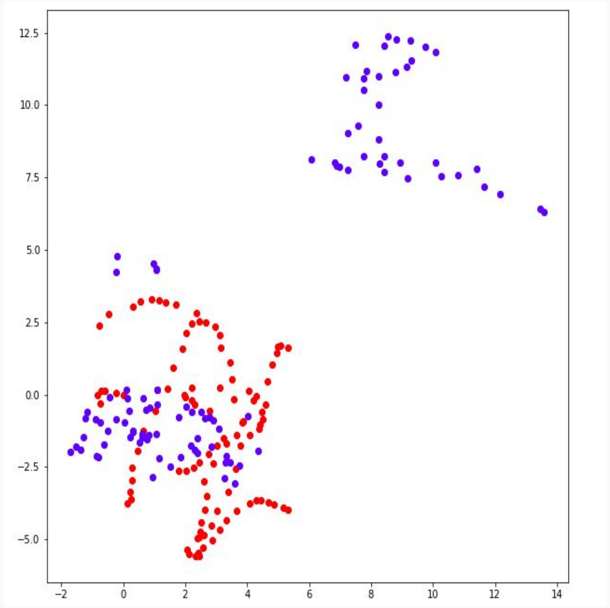


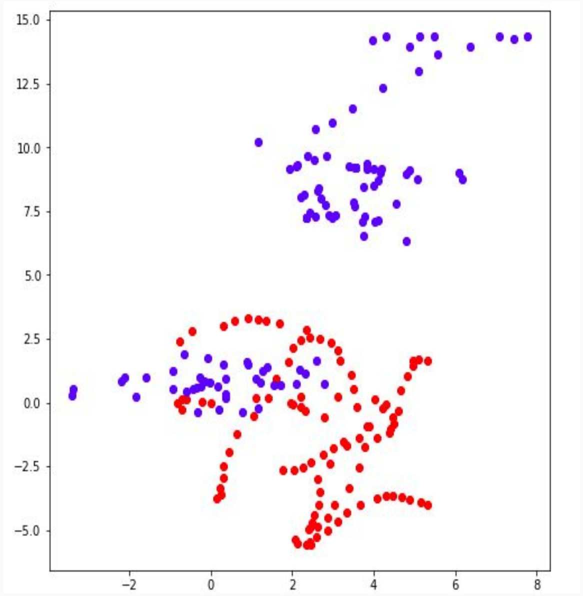
```
plt.scatter(pox, poy)
```

```
[71… <matplotlib.collections.PathCollection at 0x7f2032125a50>
```

Outputs that we got ! Problem Again:



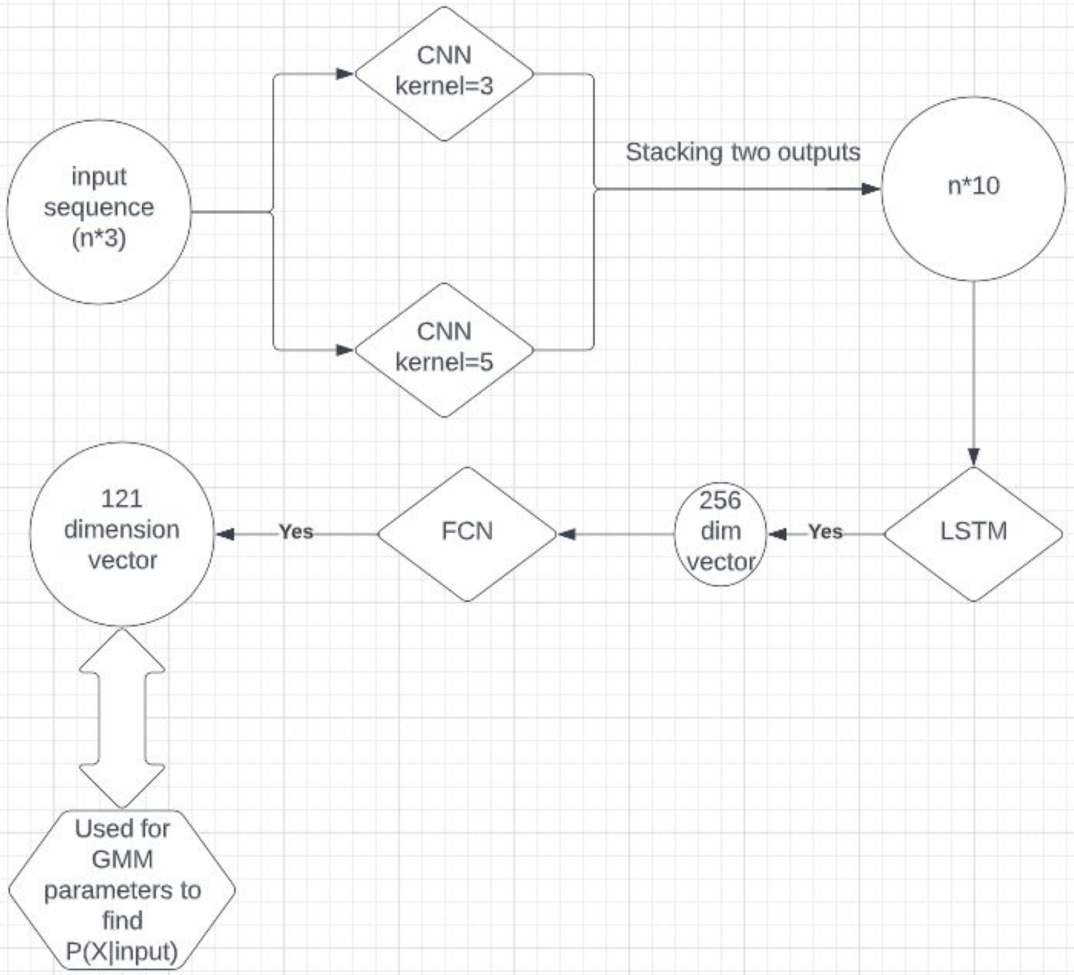




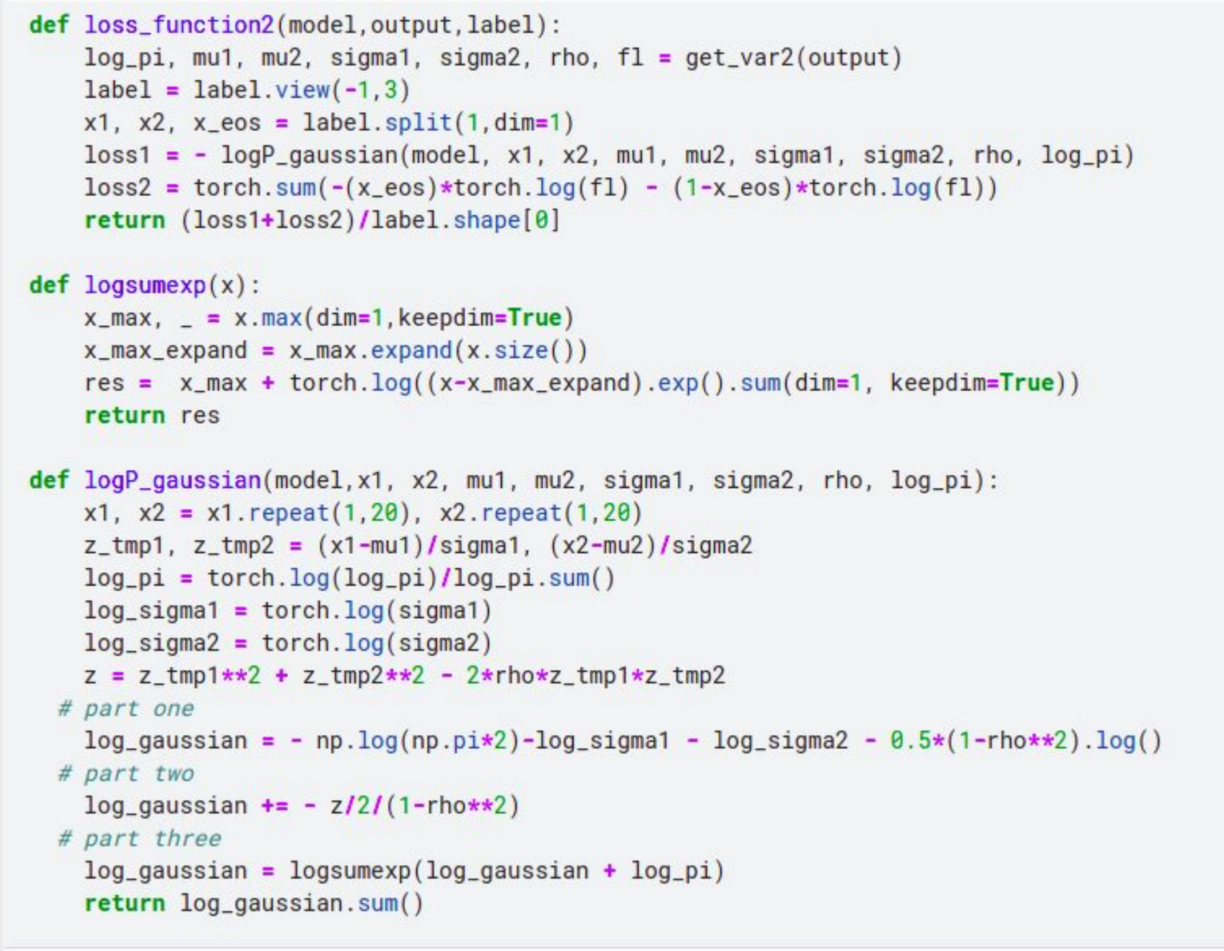
Model-3 (USE of CNN)

Here we tried something, which was not mentioned in the paper. We thought of first using CNN then sending the output with larger number of filter (from 3 to 10) to LSTM layer.

Model architecture

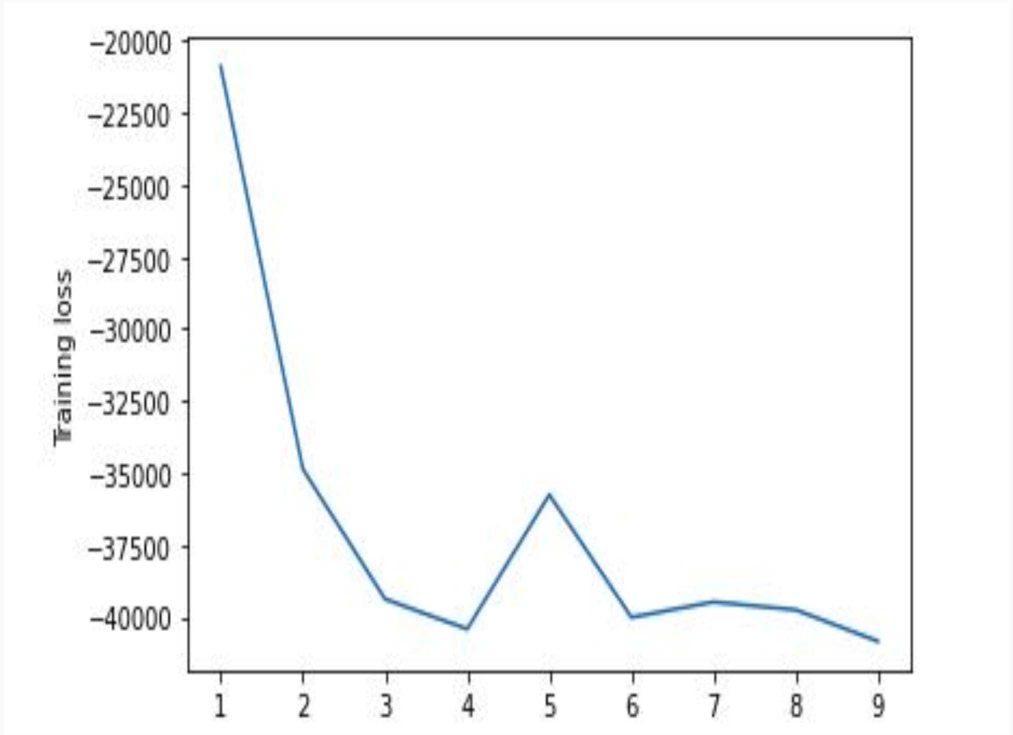


Modified Loss function implementation

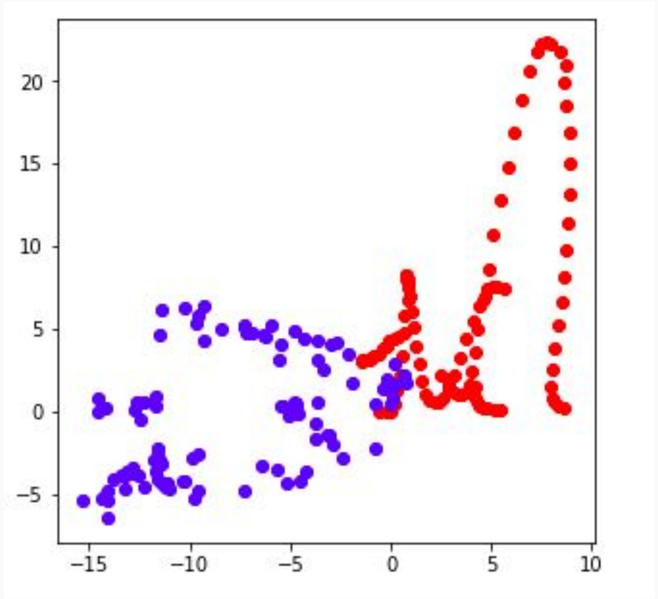


```python
def loss_function2(model,output,label):
    log_pi, mu1, mu2, sigma1, sigma2, rho, fl = get_var2(output)
    label = label.view(-1,3)
    x1, x2, x_eos = label.split(1,dim=1)
    loss1 = - logP_gaussian(model, x1, x2, mu1, mu2, sigma1, sigma2, rho, log_pi)
    loss2 = torch.sum(-(x_eos)*torch.log(fl) - (1-x_eos)*torch.log(fl))
    return (loss1+loss2)/label.shape[0]

def logsumexp(x):
    x_max, _ = x.max(dim=1,keepdim=True)
    x_max_expand = x_max.expand(x.size())
    res =  x_max + torch.log((x-x_max_expand).exp().sum(dim=1, keepdim=True))
    return res

def logP_gaussian(model,x1, x2, mu1, mu2, sigma1, sigma2, rho, log_pi):
    x1, x2 = x1.repeat(1,20), x2.repeat(1,20)
    z_tmp1, z_tmp2 = (x1-mu1)/sigma1, (x2-mu2)/sigma2
    log_pi = torch.log(log_pi)/log_pi.sum()
    log_sigma1 = torch.log(sigma1)
    log_sigma2 = torch.log(sigma2)
    z = z_tmp1**2 + z_tmp2**2 - 2*rho*z_tmp1*z_tmp2
  # part one
    log_gaussian = - np.log(np.pi*2)-log_sigma1 - log_sigma2 - 0.5*(1-rho**2).log()
  # part two
    log_gaussian += - z/2/(1-rho**2)
  # part three
    log_gaussian = logsumexp(log_gaussian + log_pi)
    return log_gaussian.sum()
```

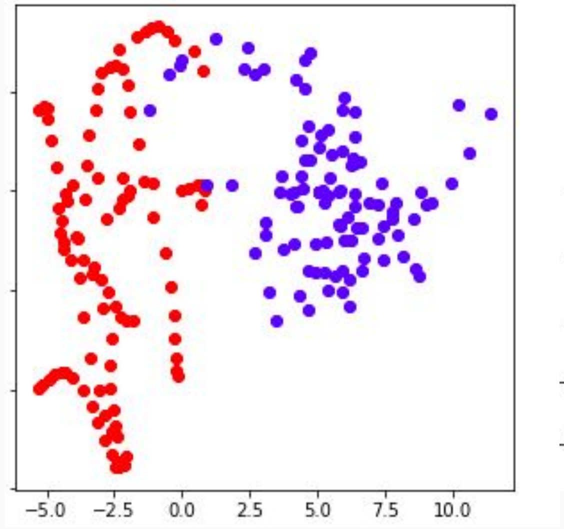
Training Loss Curve->



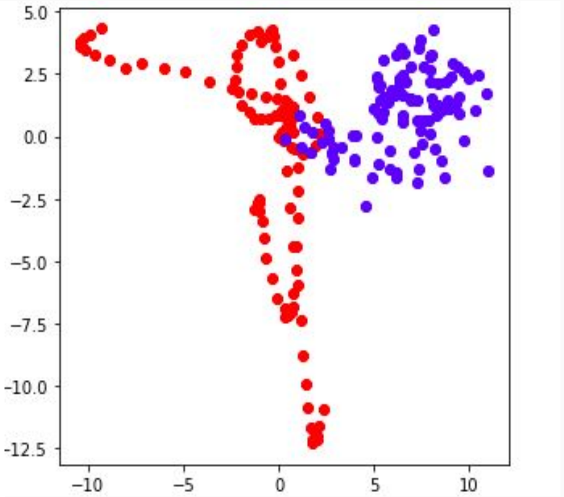
OUTPUTS:-



20
15
10
5
0
−5
−15
−10
−5
0
5
10



−5.0
−2.5
0.0
2.5
5.0
7.5
10.0



5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
-12.5
−10
−5
0
5
10

Thank You!

Github link: https://github.com/xLeviackermanX/SMAI_S22_57