Assignment 3

Zhang Yifan 11711335

Contents

1 Introduction

The aim of this lab assignment is for us to further understand the RNN and implement our own LSTM. Another part is to implement Generative Adversarial Networks which works on the MNIST dataset.

2 Part 1: PyTorch LSTM

2.1 Task 1

2.1.1 Build LSTM

In this task, I implemented my own LSTM and use it instead of the torch.nn.LSTM. My LSTM has two methods: init and forward. In init, I initiated the activation functions and linear layers and the parameters such as batch size and sequence length. All the linear layers such as gx, gh, ix, ih, fx, fh, ox, oh, ph are implemented with torch.nn.Linear and corresponding sizes. The forward function needs to first initiate the parameters and then propagate through the network. For g linear layers, tanh activation function is applied, and for other linear layers is sigmoid. At last, the output goes through ph and softmax function is applied.

2.1.2 Training Process

As for the training process, I used a similar process to the previous assignment. However, for better results, I used torch.optim.lr_scheduler to adjust the learning rate. Every 30 epochs, the learning rate would be adjusted. The optimizer used is still torch.optim.RMSprop and the backward method of torch.nn.Module.

2.2 Task 2

To check the result of my LSTM, I experimented with seq_length 3, 5, 10, 30 with all the default parameters. When sequence length is 5, it achieves the best performance because the memory is limited, if length continues to increase, the perforamce will decrease.
I did several experiments, here are the results.
First, here are the results of sequence length 10 and 3. Length 3 is apparently better than length 10.

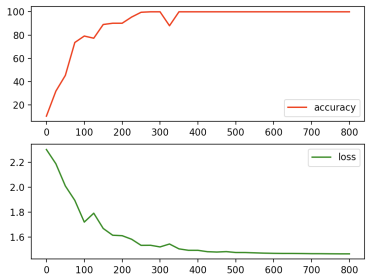

Figure 1: LSTM trained with seq_length 10

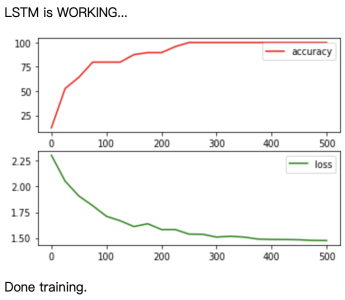

Figure 2: LSTM trained with seq_length 3

Then I compared the results of vanilla rnn and lstm. I trained them with the same parameters and same sequence length 5. And it shows that, with my LSTM, it can achieve 100% accuracy within fewer epochs, about 400, while vanilla needs more than 600 epochs. And LSTM is more stable.

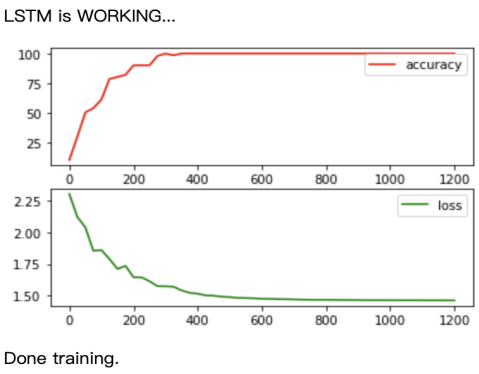

Figure 3: LSTM trained with seq_length 5

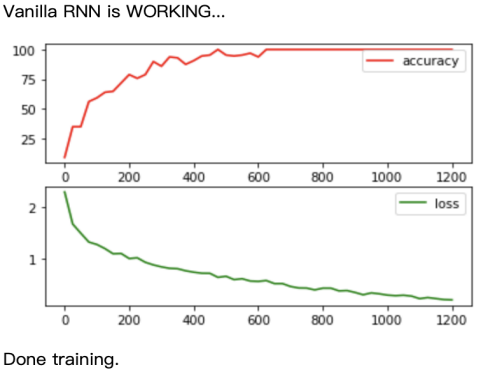

Figure 4: Vanilla trained with seq_length 5

At last, I compared the results of vanilla rnn and lstm trained with the same parameters and same sequence length 30. When the seq_length is set to 30, the perfromance is apparently not that great. However, there are also some interasting results.

Figure 5: LSTM trained with seq_length 30

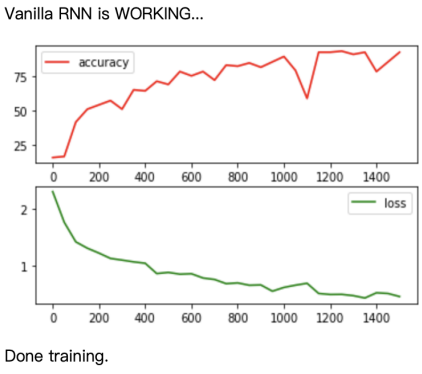

Figure 6: Vanilla trained with seq_length 30

The performance is worse than smaller seq_length because here the max steps is set to 1500, but the result is still not stable at 100% accuracy. Furthermore, the LSTM has reached 100% accuracy over 1400 epochs while the Vanilla RNN hasn't within 1500 epochs. And the accuracy of LSTM is more stable while of the

Vanilla RNN, there are dramatic fluctuations during the training process. This proves that LSTM has better performance than Vanilla RNN.

2.2.1 README

Run the part1.ipynb, there are two sections each plot the figures of my LSTM and Vanilla RNN from the last assignment.

3 Part II: Generative Adversarial Networks

3.1 Task 1

3.1.1 Introduction

The discriminator is to learn to distinguish between generated and real samples. Generation is to learn to sample from the distribution represented by the training set. The generator would take random noise as input and outputs a fake image. By minimax, here it means, the generator will generate a sample, and there would be noise and therefore another fake picture. The discriminator should learn to tell which is the real sample and which is the fake image. So the adversarial training here means the generator tries to fool the discriminator while the discriminator tries to get better at distinguishing fake vs real images. When the discriminator spots a fake picture, the generator would adjust its parameters, until the generator reproduces the true data distribution and the discriminator is unable to find differences. And we use backpropagation to train both the networks.

3.1.2 GAN

I build my GAN in my_gan.py. First, we shall implement the Generator and Discriminator. The generator consists of several layers, so I used torch.nn.Sequential to represent this series of layers. The layers are convolution layers, batch norm layers, and leakyrelu layers, where the negative slope is set to 0.2 and inplace are true. At last, since the output should be non-linear, go through tanh activation function. The forward process of generator, should be to forward the input z through all the convolution blocks (the layers initialized), while the backward process can be the backward function of nn.Module. The discriminator also consists of several layers: convolution layers, batch norm layers, and leakyrelu layers, where the negative slope is set to 0.2 and inplace are true. Since the discriminator is used to classify if the input is true or generated, so the discriminator should be a binary classifier. As for the training process, for both networks, I used Adam optimizer. And torch.utils.data.DataLoader was used to load the mnist dataset. For each epoch and each batch, first, generate the false picture with a random noise by the generator, and compute the loss between the generated image and true target. Then the discriminator will try to discriminate both the true image and the generated image, and respectively compute the loss. The generator loss is the cross-entropy loss between the generated image and true target, and the discriminator loss is the average of cross-entropy loss of both pictures. And at every save_internal, I would save the generated image and the model for future convenience.

3.2 Task 2

3.2.1 Introduction

I sampled 25 images from my trained GAN and included them in Jupyter notebook: task2.ipynb. They are taken at the start of training, halfway through training and after training has terminated. They are repectively at batches: 0, 1000, 2000, 3000, 5000, 7500, 9000, 11500, 13500, 16000, 20000, 32000, 41000, 50000, 60000, 79500, 80000, 99500, 120000, 130000, 140000, 150000, 160000, 170000, 185000, 187000.

3.2.2 README

To see all those 25 figures, run task2.ipynb or open Part2/task2_images folder. There are three categories each represent images generated at the beginning, during the training, and at the end.

3.2.3 Examples

Here are some example images.

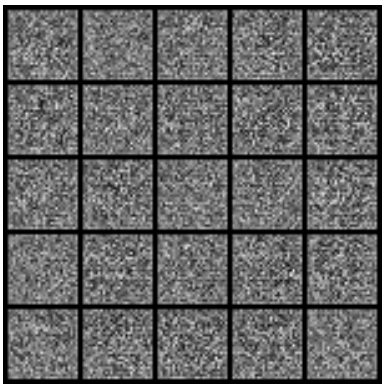

Figure 7: Image at beginning

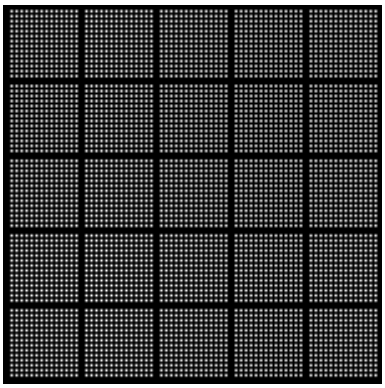

Figure 8: Image at 2000 batches

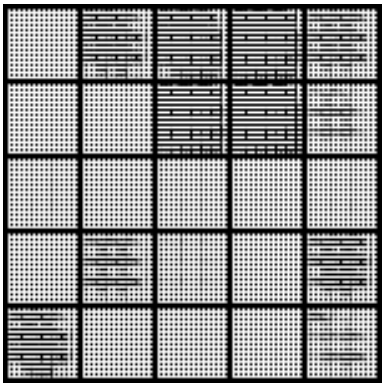

Figure 9: Image at 5000 batches

Figure 10: Image at 7500 batches

Figure 11: Image at 11500 batches

Figure 12: Image at 13500 batches

Figure 13: Image at 50000 batches

Figure 14: Image at 79500 batches

Figure 15: Image at 170000 batches

Figure 16: Image after training terminated

At the very beginning, the generated figures are very similar to random noise and we cannot tell any figure from the images. During the process, we can tell some figures from the generated images but they are not very clear or consistent. At batches 79500, most figures are clear but some figures are unconspicuous. After training terminated, all the figures are clear and consistent. As we can see from the figures, the generated images are more and more clear, and hard to discrminate from the mnist dataset.

3.3 Task 3

3.3.1 Introduction

In order to finish task3, first, we need to generate two noises. And if they are from different classes, we can use them as begin_noise and end_noise. To get the interpolate between these noises in latent space, we can gradually add the percentage of end_noise from 0 to 1 to the generating noise in nine steps. And for each generating noise, we generate an image and show it in the result. So, at last, there will be nine images.
Here are some of my experiments:

With different percentage of begin noise and end noise, the number changes from 2 to 7.

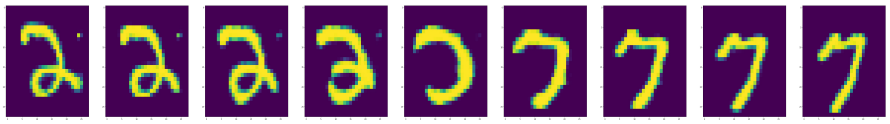

Figure 17: Interpolate 1

With different percentage of begin noise and end noise, the figure changes from 2 to 6, while the intermediate figure looks like 1.

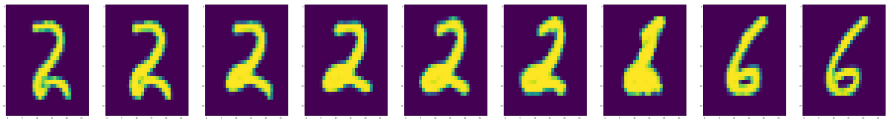

Figure 18: Interpolate 2

With different percentage of begin noise and end noise, the figure changes from 6 to 7, while the intermediate figure looks like 5.

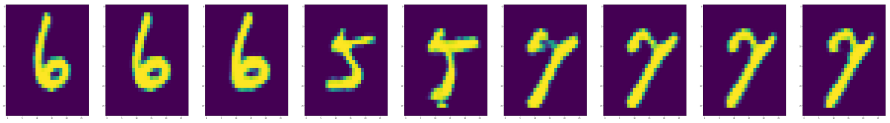

Figure 19: Interpolate 3

3.3.2 README

Run part3.ipynb to get the result of task3.

4 Appendix

The structure of the uploaded zip file 11711335_assignment3.zip : In the root directory, there is a report in .pdf and two folders, one for each part. In folder Part 1, there are LSTM code, Vanilla RNN code drom last assignment, part1.ipynb to show result and a folder named figures which store the result plots. In folder Part 2, there are code my_gan.py, task3.py, two notebooks for each task and two folders called task2_images and task3_images which store the images. And in models folder, there is the final model after training terminated.

5 References

References

[1] https://github.com/eriklindernoren/PyTorch-GAN

[2] https://deepnotes.io/softmax-crossentropy

[3] https://blog.csdn.net/Strive_For_Future/article/details/83213971