Assignment 2

Zhang Yifan 11711335

Contents

1 Introduction **3**

2 Part I: PyTorch MLP **3**
2.1 Task 1 . . . 3
2.2 Task 2 . . . 3
2.2.1 Make_moons . . . 3
2.2.2 Breast_cancer . . . 4
2.2.3 README . . . 4
2.3 Task 3 . . . 4
2.3.1 Introduction . . . 4
2.3.2 README . . . 5

3 Part II: PyTorch CNN **5**
3.1 Task 1 . . . 5
3.1.1 CNN architecture . . . 5
3.1.2 Training Procedure . . . 5
3.2 Task 2 . . . 5
3.2.1 Analysis . . . 5
3.2.2 README . . . 6

4 Part III: PyTorch RNN **6**
4.1 Task 1 . . . 6
4.1.1 RNN Structure . . . 6
4.1.2 Training Process . . . 7
4.2 Task 2 . . . 7
4.2.1 Analysis . . . 7
4.2.2 README . . . 9

5 Appendix **9**

6 References **9**

1 Introduction

The aim of this lab assignment is for us to further understand the perceptron and get to know CNN and RNN. In this assignment, first, we were asked to implement a perceptron with PyTorch and compare it with our NumPy version MLP. Then we were asked to implement a CNN using PyTorch to classify the CIFAR10 dataset. As for the last part, we were asked to implement a PyTorch RNN to predict the last digit of the input palindrome. This is a relatively easy topic for humans because we can easily find the pattern here, however, it's more difficult for the machine to learn this pattern so we want to see how good RNN can achieve. For efficiency and a better demonstration of my models, I changed some default parameters in the source code such as the max step and evaluation frequency. I finished all three parts of assignment 2 and here is my report.

2 Part I: PyTorch MLP

2.1 Task 1

This part asks us to implement the MLP architecture and the training procedure by completing the files pytorch_mlp.py and pytorch_train_mlp.py. Since we have already understood the structure of the multi-layer perceptron in the last assignment, all we had to do was to look at the PyTorch APIs and convert the NumPy version to PyTorch. I encountered some problems when I was trying to finish this part. For example, when I was trying to set the hidden layers for the MLP, I found that using a simple list does not work. So I looked at a lot of materials and I found that I should use the torch.nn.ModuleList to represent this list of hidden layers. And when loading the dataset, whether the make_moons or the CIFAR10, I found the best way to represent them is to use the train_loader, test_loader, and valid_loader to load these data.

2.2 Task 2

2.2.1 Make_moons

The second task of this part is to train the PyTorch MLP and NumPy MLP on the same data and strategy and try to compare the results. For this task, I choose stochastic gradient descent and first used make_moons to generate the train and test data. For convenience, I put the train methods of two models in the same file, so I can guarantee that they are trained and tested on the same data, and for better visualization of results.

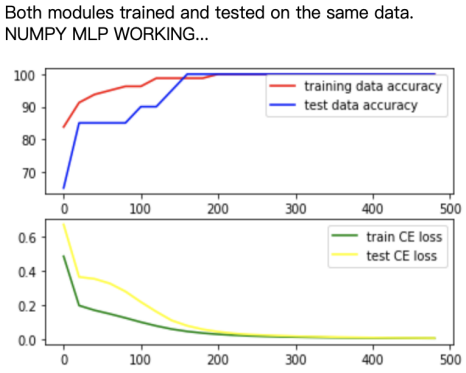

Figure 1: Trained on Numpy MLP

Figure 2: Trained on PyTorch MLP

As we can see from the figure, the data used for the two models are the same, and they have similar curves of accuracy and loss. However, probably with some luck, my NumPy version of MLP achieved 100% accuracy on the training data slightly earlier than the PyTorch version.

2.2.2 Breast_cancer

I decided to do more experiments on more datasets to see the capacities of both models. When I was exploring the datasets of scikit_learn, I found a very interesting dataset: breast cancer classification. It is a dataset of different cancer tissues and we need to classify whether they are benign or malignant. I used datasets.load_breast_cancer() to load this data.

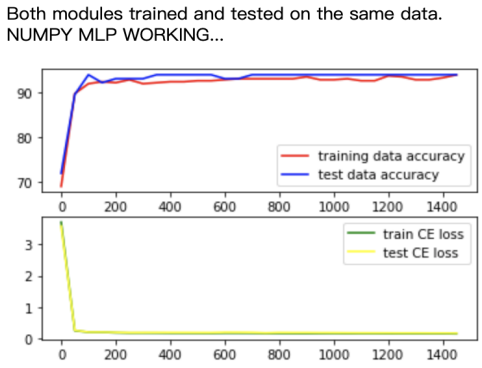

Figure 3: Trained on Numpy MLP

Figure 4: Trained on PyTorch MLP

We can see the curves are not that smooth as the previous one. This is because this dataset is relatively small and only has 569 samples. However, they still achieve similar results and both can reach over 90% accuracy within 300 epochs.

2.2.3 README

Run the part1.ipynb, then wait for some time and we can see the result. The max epoch here is 500 because I found that, with SGD and default parameters, both models can achieve 100% accuracy within about 200 epochs.

2.3 Task 3

2.3.1 Introduction

For this task, we are asked specifically to use PyTorch MLP to train the CIFAR10 dataset. And we are asked to improvise and try to get a better result. So for this part, I wrote another file called cifar.py and refined the MLP structure and training process. For the MLP, I made several modifications. For hidden layers, I specify it has two hidden layers and each has 100 nodes. And I added dropout to the structure for better results.

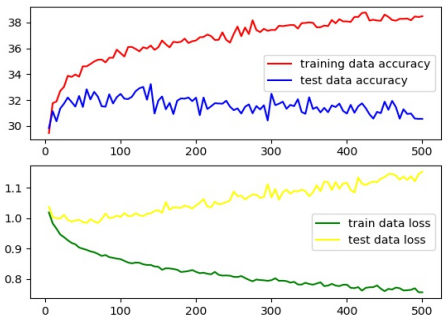

Figure 5: CIFAR10 with MLP

The results are not very satifying with MLP, but they are acceptable. We can achieve higher accuracy with CNN later.

2.3.2 README

Run the part1.ipynb and see the training results.

3 Part II: PyTorch CNN

3.1 Task 1

In this task, we are asked to implement the CNN architecture and the training process. I followed the instructions on the last pages of Lecture7 to implement this CNN architecture.

3.1.1 CNN architecture

First I initialized the convolution layers and batch norm and pooling layers. I decided that after each convolution layer, we add a batch norm layer, and we only need one pool layer since they are all of the same sizes. I used torch.nn.Conv2d() to initialize the convolution layers and torch.nn.BatchNorm2d for batch norm layers. The batch norm layers should be of the same size as the output size of the according to convolution layers. And remember to set padding to 1 and color channels to 3 for each layer. The pooling layer should be torch.nn.MaxPool2d(3, 2, 1). And finally, the fc1 layer should be torch.nn.Linear. Here the size of the fc layer is tricky and after some calculation, I decided that it should be torch.nn.Linear(512 * 1 * 1, 10) where 512 is the output of the last convolution layer. And as for the forward process, we need to make sure that the order is: the input first goes through one convolution layer, then the batch norm layer, then the activation function while here used is torch.nn.functional.relu, and at last the pooling layer when it's required. And before fc layer, we need to convert x by x.view(-1, 512 * 1 * 1) and then fc layer and relu layer. Since torch.nn.Module already has a backward function, we don't need to implement it by ourself.

3.1.2 Training Procedure

First, we need to initialize the transformer and load train_data and test_data, and check whether CUDA is available on the server. And then we use torch.utils.data.DataLoader to prepare training data and test data. For entropy loss computation and optimizer, we respectively use torch.nn.CrossEntropyLoss() and torch.optim.Adam() as demanded in the default parameters. For each epoch, we first forward the input, compute the loss, used the loss to backward train the model, and compute and visualize the accuracy on evaluation epochs. We can get the predictions from the model and check the accuracy. Since there are 10 classes of the output, I have calculated the accuracy for each class and the total accuracy for all ten classes. On evaluation epochs, I also compute the accuracy and loss of the test data to see if the model is learning.

3.2 Task 2

3.2.1 Analysis

This part asked us to analyze the performance of the model by plotting accuracy and loss curves in a Jupyter notebook. We have to apply mini-batch gradient descent and Adam optimizer. Other than this, I used SGD and different batch sizes to experiment. So here are the result and my analysis. First I used Adam and the default batch size 32 to train, I found that although the training accuracy can reach about 100% in around 80 epochs, however, the test accuracy can only reach about 85%. The loss and accuracy of train data suggest the training method is working, however, this model and dataset tend to overfit such that the test accuracy is relatively lower than train accuracy.

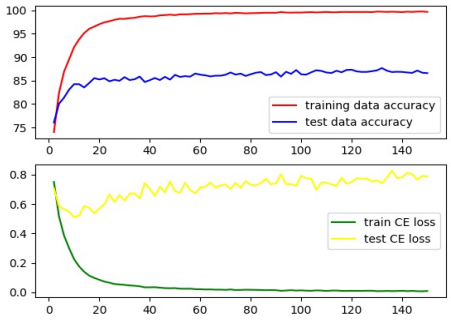

Figure 6: Optimizer: Adam Batch size: 32

I did more experiments with another optimizer: SGD. With SGD optimizer, it can reach 100% accuracy for training data within 40 epochs. However, the test accuracy is only about 70%. SGD does not work as well as Adam in this part.

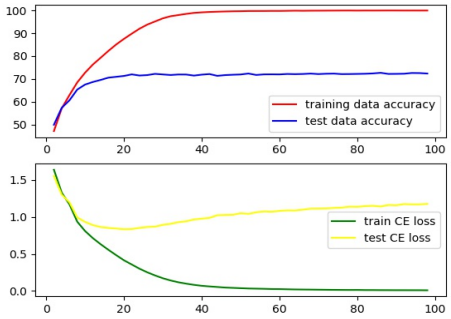

Figure 7: Optimizer: SGD Batch size: 32

3.2.2 README

Run part2.ipynb.

4 Part III: PyTorch RNN

4.1 Task 1

This task asks us to implement the Vanilla RNN with PyTorch and the training process. The aim is to predict the last digit of the palindrome of the designated sequence length.

4.1.1 RNN Structure

We need to build the RNN according to the assignment requirements. U used torch.nn.LSTM, while the parameters are: torch.nn.LSTM(seq_length, hidden_dim, 1). The output layer is a Linear layer. We also need to initialize self.h0 and self.c0 when initializing, and their sizes are both batch_size * self.hidden_dim. When writing the forward function, it's relatively simple but there are some aspects to pay attention to. While we input x into the LSTM module, the parameters should be (x, (self.h0, self.c0)). And then the result goes through the output layer and we get the predicted answer.

4.1.2 Training Process

The main idea of the training of Vanilla RNN is to input one batch of data into the model on each epoch and use the entropy loss computed to backward train the model. First, we load data from the PalindromeDataset and divide them into batches according to the batch size. The loss computation is by torch.nn.CrossEntropyLoss(), and the optimizer I used is torch.optim.RMSprop. The parameters I input are: torch.optim.RMSprop(model.parameters(), config.learning_rate, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False).
Then it's the training epochs. First, we need to deal with exploding gradients. The code in the given file is deprecated, so I changed it to torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=config.max_norm). Then we input the batch_inputs, and compute the entropy loss between output and bacth_targets. And on evaluation epochs, we compute the accuracy for training data and test data. Because the default batch is relatively small, so I increased the value of evaluation frequency for a smoother curve.

4.2 Task 2

This part asks us to plot the accuracy and loss curves, and experiment with different palindrome length. Here are my research results.

4.2.1 Analysis

I used max epoch 1500 and evaluation frequency 50 when plotting the curves. First I experimented with three different sequence lengths: 5, 10, 30. I found that with larger sequence length, the stability and accuracy reduces. Here are the figures. This part asks us to plot the accuracy and loss curves, and experiment with different palindrome length. Here are my research results. These are the curves for Vanilla RNN trained with palindrome length 10, optimizer RMSProp, and default parameters.

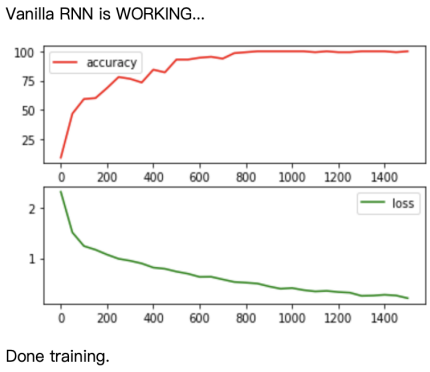

Figure 8: Length: 10 Optimizer: RMSProp

Then I chanllenged it with palindromes with length of 30. These are the curves for Vanilla RNN trained with palindrome length 30, optimizer RMSProp, and default parameters.

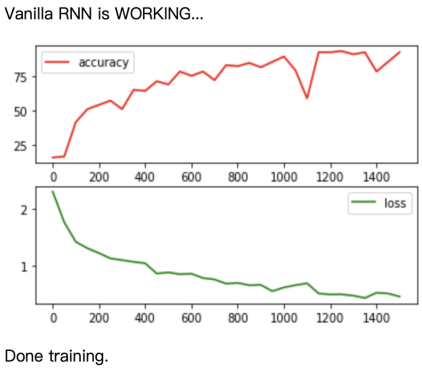

Figure 9: Length: 30 Optimizer: RMSProp

These are the curves for Vanilla RNN trained with palindrome length 5, optimizer RMSProp, and default parameters.

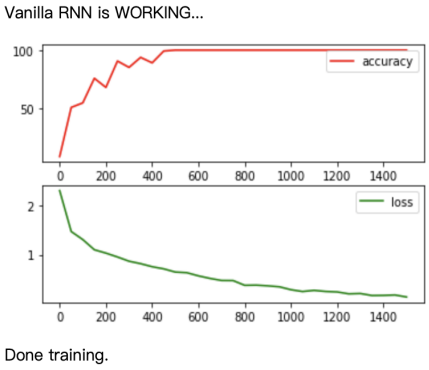

Figure 10: Length: 5 Optimizer: RMSProp

From the above figures, we can easily see that when the length is 5, our model can achieve the best result where the curves are smooth and it reached 100% accuracy only within 500 epochs. When the length increases, to 10, the curves were also smooth, but it reaches almost 100% accuracy later at about 800 epochs. And when the length increases, to 30, the curves were not smooth, and it at best reaches about 80% accuracy within 1500 epochs. These results correspond to our expectations because we know that RNNs have limited memory. And this part was to use RNN to predict the last digit of a palindrome, which to our knowledge, should be the same as the first digit. As the sequence length increases, it becomes more and more unlikely that RNN can discover this pattern so that the accuracy would drop as the sequence length increases.

At last, I figured that I should experiment with another optimizer, so I checked SDG because I think it might achieve 100% accuracy more quickly.

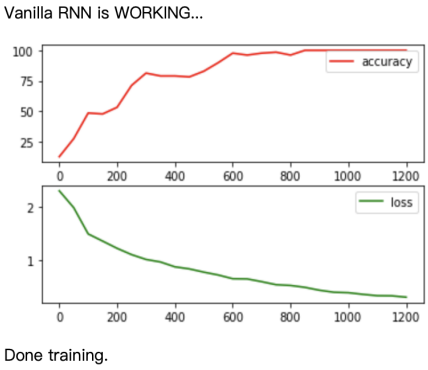

Figure 11: Length: 10 Optimizer: SGD

However, on the contrary, SGD does not work well as RMSProp in this part. So I did some research and I found that, on the foundation of AdaGrad, RMSProp made some improvements on the calculation method of second-order momentum to deal with the sharp decline in the learning rate, and therefore should be the best optimizer for this problem.

4.2.2 README

To run the part3.ipynb. We need to designate the sequence length in this ipynb file to experiment with different lengths, or they are executed with default parameters and RMSProp optimizer. To change optimizer, we need to modify the source code. For example, to use SDG, change optimizer to torch.optim.SGD(model.parameters(), config.learning_rate).

5 Appendix

The structure of the uploaded zip file 11711335_assignment2.zip : In the root directory, there is a report in .pdf and three folders, one for each task. In each folder, there are python files, .ipynb file, and screenshots used in the report.

6 References

References

[1] https://github.com/pytorch/pytorch/pull/9655

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

[3] https://www.zybuluo.com/spiritnotes/note/295894

[4] https://blog.csdn.net/q295684174/article/details/79130666