Report

November 20, 2021

1 HW6 Report

1.1 Jeong Hyun Moon

1.2 2021-35791

1.3 Environment

At first, I have planned on working on the server. However, when I try to run GPU, there were too many jobs queued and Slurm Partition was not able to get any resources. To double check, I have tried 0 (equivalent for CPU) but the CPU was able to get the space. After waiting for approximately an hour, I have ended up working on this assignment in the Google Colab GPU Server. I have broken up this problem into three separate python notebook and obtained the following results. In order to check whether the code is running as GPU, I have first printed the device, checked the model.parameters(), and outputs to see whether device and values are cpu or cuda:0. For all checkpoint-results, I have found out all of them used cuda:0.

1.4 Problem 1

1.4.1 Script As It Is

In order to run the code in appropriate manner, I have started with Script as it is with including the running time. I have included time1 = time.time() in the very beginning of the code and time2 = time.time() at the end of the code. Subtracting the two times, I have obtained the total running time of entire script. Furthermore, I have included train_time = time.time() right after definition of running loss and train_time2 = time.time() right after else clause for when the for clause finish, it would proceed through the else statement marking end of training. Subtracting the two times, I have obtained the following results:

- Epoch: 1/5.
  - Training loss: 0.488
  - Test loss: 0.423
  - Test Accuracy: 0.840
  - Time: 19.77926993370056
- Epoch: 2/5.
  - Training loss: 0.373
  - Test loss: 0.396
  - Test Accuracy: 0.854
  - Time: 19.79761290550232
- Epoch: 3/5.
  - Training loss: 0.339

  - Test loss: 0.373
  - Test Accuracy: 0.866
  - Time: 20.1186466217041
- Epoch: 4/5.
  - Training loss: 0.316
  - Test loss: 0.361
  - Test Accuracy: 0.872
  - Time: 20.131465196609497
- Epoch: 5/5.
  - Training loss: 0.297
  - Test loss: 0.371
  - Test Accuracy: 0.867
  - Time: 20.082324266433716
- Total time: 118.79080986976624

For each Epoch, it took roughly 20 seconds and having an accuracy of mid 80%. For next few trials, I have copied the current script and made adjustment based on what the assignment desired for.

1.4.2 Change In Batch Sizes

The next step for the assignment was to analyze the changes in batch size. In order to do this, one would need to change the following lines of code:

- trainloader = torch.utils.data.DataLoader(trainset, batch_size = 32, shuffle = True, num_workers=4)
- testloader = torch.utils.data.DataLoader(testset, batch_size = 32, shuffle = True, num_workers=4)

In order for this to work, I have changed the batch_size = 32 to batch_size = 16 and batch_size = 64. With these changes, I have obtained the following result for batch size of 16 to be as follows:

- Epoch: 1/5.
  - Training loss: 0.476
  - Test loss: 0.448
  - Test Accuracy: 0.837
  - Time: 27.752448081970215
- Epoch: 2/5.
  - Training loss: 0.375
  - Test loss: 0.404
  - Test Accuracy: 0.855
  - Time: 28.701561212539673
- Epoch: 3/5.
  - Training loss: 0.342
  - Test loss: 0.390
  - Test Accuracy: 0.860
  - Time: 29.306257724761963
- Epoch: 4/5.
  - Training loss: 0.317

  - Test loss: 0.367
  - Test Accuracy: 0.868
  - Time: 28.63768243789673
- Epoch: 5/5.
  - Training loss: 0.303
  - Test loss: 0.361
  - Test Accuracy: 0.874
  - Time: 28.957837343215942
- Total time: 163.80136346817017

For the case of batch size of 64, I have obtained:

- Epoch: 1/5.
  - Training loss: 0.500
  - Test loss: 0.440
  - Test Accuracy: 0.843
  - Time: 16.528880834579468
- Epoch: 2/5.
  - Training loss: 0.382
  - Test loss: 0.427
  - Test Accuracy: 0.849
  - Time: 16.736101388931274
- Epoch: 3/5.
  - Training loss: 0.347
  - Test loss: 0.371
  - Test Accuracy: 0.863
  - Time: 16.673275470733643
- Epoch: 4/5.
  - Training loss: 0.319
  - Test loss: 0.355
  - Test Accuracy: 0.873
  - Time: 16.656588792800903
- Epoch: 5/5.
  - Training loss: 0.300
  - Test loss: 0.363
  - Test Accuracy: 0.867
  - Time: 16.606858253479004
- Total time: 97.05978083610535

As one can see from the above result, as the batch size increases, the time to train and run the entire script decreases. For batch size of 16, it took about 29 seconds for training time, and for batch size of 64, the training took about 17 seconds. However, it appears as if changning the batch size does not seems to affect the test accuracy. Considering training loss and test loss, there appears to be no impact.

1.4.3 Change in lr

For the Learning Rate, the change occurred in following line of code

- optimizer = optim.Adam(model.parameters(), lr = 0.001)

From above code, I have changed lr = 0.001 to be lr = 0.01 and lr = 0.0001. With the following changes, I have obtained the following result for lr = 0.01 to be as follows:

- Epoch: 1/5.
  - Training loss: 0.583
  - Test loss: 0.535
  - Test Accuracy: 0.817
  - Time: 20.39034152030945
- Epoch: 2/5.
  - Training loss: 0.503
  - Test loss: 0.512
  - Test Accuracy: 0.826
  - Time: 20.72736096382141
- Epoch: 3/5.
  - Training loss: 0.486
  - Test loss: 0.509
  - Test Accuracy: 0.835
  - Time: 20.818732261657715
- Epoch: 4/5.
  - Training loss: 0.473
  - Test loss: 0.548
  - Test Accuracy: 0.823
  - Time: 20.53863549232483
- Epoch: 5/5.
  - Training loss: 0.456
  - Test loss: 0.510
  - Test Accuracy: 0.834
  - Time: 20.832820177078247
- Total time: 119.51724076271057

For the case of lr = 0.0001, I have obtained:

- Epoch: 1/5.
  - Training loss: 0.651
  - Test loss: 0.514
  - Test Accuracy: 0.815
  - Time: 20.479434728622437
- Epoch: 2/5.
  - Training loss: 0.455
  - Test loss: 0.460
  - Test Accuracy: 0.835
  - Time: 20.618394374847412
- Epoch: 3/5.
  - Training loss: 0.416
  - Test loss: 0.441
  - Test Accuracy: 0.841
  - Time: 20.535950422286987
- Epoch: 4/5.
  - Training loss: 0.392

  - Test loss: 0.419
  - Test Accuracy: 0.848
  - Time: 20.329606533050537
- Epoch: 5/5.
  - Training loss: 0.374
  - Test loss: 0.409
  - Test Accuracy: 0.853
  - Time: 20.27568006515503
- Total time: 118.01114583015442

For the LR, it has quite unusual pattern. It appears that lr = 0.001 performs the best among 3, then 0.0001, and lastly 0.01. It appears for both changing in lr has impacted training loss and test loss where both losses increased in 0.01 and 0.0001 compared to lr = 0.001. In terms of time, the time to run was about the same for all three cases.

1.4.4 Change from Adam -> SGD

For the last case of optimizer, the change occurred in following line:

- optimizer = optim.Adam(model.parameters(), lr = 0.001)

From above, optim.Adam changed to optim.SGD. With the change, I have obtained the result to be

- Epoch: 1/5.
  - Training loss: 1.355
  - Test loss: 0.901
  - Test Accuracy: 0.726
  - Time: 19.296220779418945
- Epoch: 2/5.
  - Training loss: 0.774
  - Test loss: 0.716
  - Test Accuracy: 0.756
  - Time: 19.082611083984375
- Epoch: 3/5.
  - Training loss: 0.661
  - Test loss: 0.647
  - Test Accuracy: 0.772
  - Time: 19.03814721107483
- Epoch: 4/5.
  - Training loss: 0.607
  - Test loss: 0.608
  - Test Accuracy: 0.780
  - Time: 19.375463485717773
- Epoch: 5/5.
  - Training loss: 0.572
  - Test loss: 0.580
  - Test Accuracy: 0.795
  - Time: 19.402095079421997
- Total time: 112.22894525527954

Changing from Adam to SGD has decreased the run time a little bit. However, both training and test loss increased with change to SGD, and test accuracy has decreased.

1.5 Problem 2

1.5.1 Analysis1

For the result of the run of the general script, I have obtained following result:

- Epoch: 1/5
  - Training loss: 0.487
  - Test loss: 0.430
  - Test Accuracy: 0.841
- Epoch: 2/5
  - Training loss: 0.375
  - Test loss: 0.388
  - Test Accuracy: 0.860
- Epoch: 3/5
  - Training loss: 0.340
  - Test loss: 0.384
  - Test Accuracy: 0.864
- Epoch: 4/5
  - Training loss: 0.313
  - Test loss: 0.370
  - Test Accuracy: 0.869
- Epoch: 5/5
  - Training loss: 0.296
  - Test loss: 0.353
  - Test Accuracy: 0.877

It is possible that I have misunderstood this question. Because nn.sequential does not have way to save intermediate outputs, I have first checked model[0].weight.gradient and model[0].bias.gradient to see how it looks. However, since the question asks for Linear to ReLU, I understood as if these are not used because these were only used for Linear layer. Then, I have checked rather there are any gradient equivalent for the ReLU or model[1], but there does not appear to be any. Then, I have played around with model.parameter()'s param and param.grad, but I found out that these are for the named parameters (weight and bias for the two models). As a result, I have found out that none of the above would work. As a result, I have decided to look from different perspective: saving the intermediate results for each step of the code. This was done using the below code right before ps

[ ]:
```
    output2 = model[0:1].forward(images)
    output3 = model[0:2].forward(images)
    output4 = model[0:3].forward(images)
```

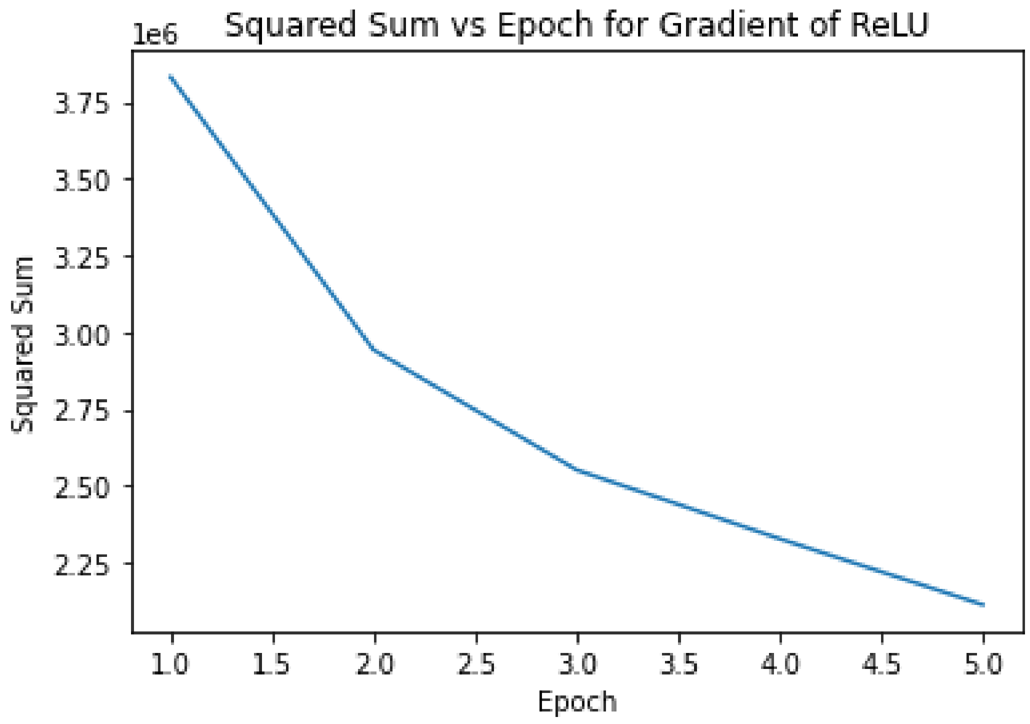
I have checked whether output and output4 is same, and it appeared as they give same result. To further check, I have seen if output2 and output3 have equivalentnon-zero value, and it turns out to be the same. Next step was to get gradient using the output2's (First linear output) result. Since there were many trained image, I have added this code: relu0.append(sum(sum(output2>0))). It will first sum up number of non-zero in inner array, and sum all to output array. After adding all of the array, this would save to relu0 for each epoch. At the end of the outer for loop, I have added the following line: outputs.append(sum(relu0)). The reason why I have done above step is related to gradient of ReLU. For ReLU, we have gradient of value 1 if the input(x) is bigger than 0, else 0. Since we have many input points, test2>0 would get all values that are positive meaning gradient will be 1. Summing up all, we would have total values. Because square of 1 is 1, I have not added square in this code to reduce the run time. I have saved it to prob2.csv, and using this csv, I will show the final result.

[1]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

df = pd.read_csv('prob2.csv')
df.drop(['Unnamed: 0'],axis = 1)
epoch = df['Epoch']
Squared_Sum = df['Squared_Sum']

plt.plot(epoch, Squared_Sum)
plt.xlabel('Epoch')
plt.ylabel('Squared Sum')
plt.title('Squared Sum vs Epoch for Gradient of ReLU')
```

[1]:
```
Text(0.5, 1.0, 'Squared Sum vs Epoch for Gradient of ReLU')
```



As can be seen from above graph and the value of Squared_Sum, we can see that as the epoch increases, there are decrease in squared sum. The way I interpret the graph is as the epoch increases, the model will learn more about the data. Looking from this perspective, this would allow the Linear to produce better result in this case it would be having value less than 0 (after Linear transformation occurs). We can also see from the graph that as the epoch increases the steepness of slope is also decreasing. This I think goes same from previous explanation since model learns and adjust every epoch making it less steep.

1.6 Analysis 2 (Different Approach - Maybe the class method)

As I was reading through the Pytorch tools, it appeared as if the hook is one of the ways to save the gradient of the layers. As I was reading throught the hook, it appears as if the registering back hood and saving the gradient input for the ReLu would achieve the gradient squared sum as defined in the class (I may have misunderstood the hook). In order to do so, first, I have defined function for appending the gradient input to list a. After the model is defined, I have registered backward hook to ReLu. By doing so, everytime the model is performed, it would save the gradient value for ReLU to array. Inside the epoch for loop, I have emptied the list, so that the output for each epoch would be the same. Afterwards, I have initialized a variable called value to save the value for squared sum of Gradients. Then, I have iterated the array a in order to get squared sum of gradients per epoch. To be more detailed, it will first iterate through all the runs for the for loop performed in trainloader. Then, it would loop through the batches to obtain the squared sum of gradients for all sum. After for loop is complete, it would be saved in the outputs. I have demonstrated as in code below. Furthrmore, I have plotted the graph of the result afterwards. In the graph, it performed similar to before where as the epoch increases, there are decrease in squared sum of gradients, and the steepness gets less steep as the epoch increases.

```
[ ]: a = []

     def hookFunc(module, gradInput, gradOutput):
       a.append(gradInput)

     ... # model definintion
     model[1].register_backward_hook(hookFunc)

     ...

     # initialize the array for the graph
     outputs = []

     ... epoch for loop
       a = []

     ... else loop
         ## Squared Sum of Gradient
         value = 0
         # First, go through all of the runs for for loop in trainloader
         for i in range(len(a)):
           # Then for each batch, add the square sum of gradients
```

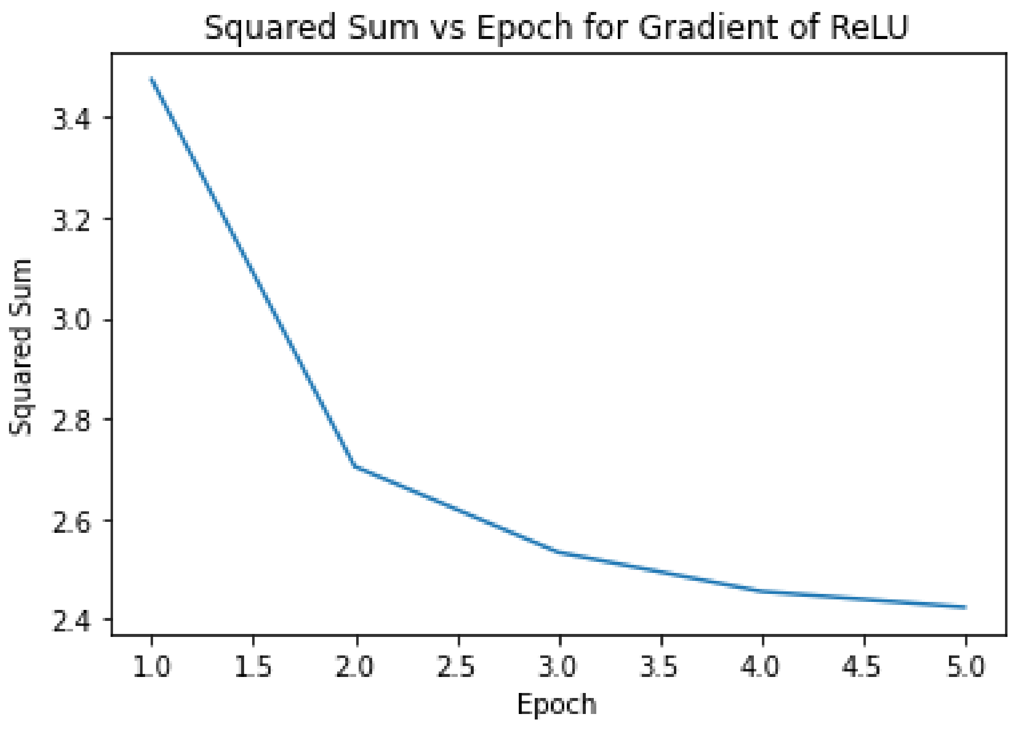
```
      for j in range(len(a[i][0])):
        value = value + torch.sum(torch.square(a[i][0][j]))
    # Save the gradient value
    outputs.append(value)
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

df = pd.read_csv('prob2-2.csv')
df.drop(['Unnamed: 0'],axis = 1)
epoch = df['Epoch']
Squared_Sum = df['Squared_Sum']

plt.plot(epoch, Squared_Sum)
plt.xlabel('Epoch')
plt.ylabel('Squared Sum')
plt.title('Squared Sum vs Epoch for Gradient of ReLU')
```

[2]:
```
Text(0.5, 1.0, 'Squared Sum vs Epoch for Gradient of ReLU')
```



1.7 Problem 3

1.7.1 Analysis

In lecture, we have learned that the place where the gradient resets lies in optimizer.zero_grad(). The following function will make the gradient to be zero. If the function is not called, the gradient will be accumulated throughtout the functions. Before making any changes, I have obtained the following result:

- Epoch: 1/5
  - Training loss: 0.486
  - Test loss: 0.432
  - Test Accuracy: 0.841
- Epoch: 2/5
  - Training loss: 0.373
  - Test loss: 0.400
  - Test Accuracy: 0.853
- Epoch: 3/5
  - Training loss: 0.336
  - Test loss: 0.392
  - Test Accuracy: 0.859
- Epoch: 4/5
  - Training loss: 0.314
  - Test loss: 0.427
  - Test Accuracy: 0.851
- Epoch: 5/5
  - Training loss: 0.299
  - Test loss: 0.365
  - Test Accuracy: 0.871

Running the code without optimizer.zero_grad, I have obtained the following result:

- Epoch: 1/5
  - Training loss: 2.315
  - Test loss: 2.668
  - Test Accuracy: 0.229
- Epoch: 2/5
  - Training loss: 2.715
  - Test loss: 2.805
  - Test Accuracy: 0.195
- Epoch: 3/5
  - Training loss: 2.899
  - Test loss: 3.302
  - Test Accuracy: 0.233
- Epoch: 4/5
  - Training loss: 4.155
  - Test loss: 3.258
  - Test Accuracy: 0.178
- Epoch: 5/5
  - Training loss: 2.956
  - Test loss: 3.321

- Test Accuracy: 0.118

Comparing the two, one with the optimizer.zero_grad (default) actually has performed much better. While one with optimizer.zero_grad has shown some type of pattern where increasing the epoch increase the test accuracy (for most epochs). On the other hand, one without the zero_grad appears to have decrease in test accuracy as the number of epoch increases.