ECE 661: Homework #2

Construct, Train, and Optimize CNN Models

NetID: wh162

1. True/False Questions
    a. Problem 1.1: False, besides subtracting the mean, we also have to divide the square root of the sum of the variance plus a small value.

$$\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

    b. Problem 1.2: False, in PyTorch, after we calculation the loss, we can simply call loss.backward() to perform back propagation.

    c. Problem 1.3: False, not all kind of image are beneficial for data augmentation technique, for example, medical images aren't suitable for data augmentation

    d. Problem 1.4: False, the purpose of using dropout layer is to prevent the model from being over-fitting, it acts like L2 Normalization. As a result, it is not guarantee that the CNN would converge less fast without dropout.

    e. Problem 1.5: True, we know that dropout acts like a regularization in the model, so adding a dropout layer after a regularization would bring out a better performance of a model.

    f. Problem 1.6: True, L1 gives out the weights into values between 0 and 1, whereas L2 calculate by adding values correspond to the weights themselves. Can also check Lec6 P7.

    g. Problem 1.7: False, Leaky ReLU does fixed the dead neuron problems, but due to the inconsistence of the slope, the stability of the model while training is not guaranteed to improve.

    h. Problem 1.8: True, according to the formula below, we know that when N and M is large, we can expect a speedup of 9x.

$$\frac{3 \times 3 \times M \times N \times D_F \times D_F}{3 \times 3 \times M \times D_F \times D_F + M \times N \times D_F \times D_F} \approx 9$$

i. Problem 1.9: False, SqueezeNet replaces some of the 3x3 filters with 1x1. SqueezeNet stacks more convolutional layers at the early stage of CNN architecture.

j. Problem 1.10: True, Refer to Lec7 P31

2. Lab(1): Training simpleNN for CIFAR-10 classification
   a. Problem 2.1: From the code in the below, we can see that the output shape of the model is 10, which is the same as stated at fc3 layer; also the parameter met the expectation from the design of the model.

```
In [30]: # the cifar 10 image has the size of 32x32
size = 32
dummy = torch.randn(5, 3, size, size)

# GPU check
device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device =='cuda':
    print("Run on GPU...")
else:
    print("Run on CPU...")

net = SimpleNN()
net = net.to(device)
out = net(dummy)
print(out.shape)

for name, module in net.named_modules():
    if isinstance(module, CONV) or isinstance(module, FC):
        input = module.input.detach().numpy()
        output = module.output.detach().numpy()
        weight = module.weight.detach().numpy()
        if isinstance(module, CONV):
            num_Param = weight.size
        else:
            num_Param = module.bias.detach().numpy().size + output.shape[1] * input.shape[1]
        print(f'{name:10} {str(num_Param):10}')

Run on CPU...
torch.Size([5, 10])
conv1      600
conv2      1152
fc1        69240
fc2        10164
fc3        850
```

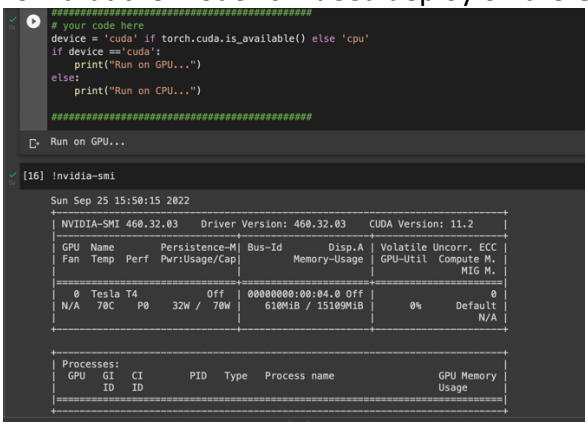
   b. Problem 2.2: For data preprocessing, I chose to use RandomHorizontalFlip, RandomVerticalFlip, and Normalization to augment my dataset. For both vertical and horizontal flipping, I adjusted a probability of 50%. As for Normalization, we normalize the image but subtracting the input channel and mean, then divide by standard deviation, by normalizing the input image, the model is able to prevent over-fitting. I observed that when I add a CenterCrop in the Compose layer, the model will return an error, however I don't know why using that augmentation would return and error.

   c. Problem 2.3:

```
##########################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT,
    mode='train',
    download=True,
    transform=True    # your code
)
val_set = CIFAR10(
    root=DATA_ROOT,
    mode='val',
    download=True,
    transform=False    # your code
)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,  # your code
    shuffle=True,     # your code
    num_workers=4
)
val_loader = DataLoader(
    val_set,
    batch_size=VAL_BATCH_SIZE,  # your code
    shuffle=False,     # your code
    num_workers=4
)
##########################################
```

```
Using downloaded and verified file: ./data/cifar10_trainval_F22.zip
Extracting ./data/cifar10_trainval_F22.zip to ./data
Files already downloaded and verified
```

d. Problem 2.4: By checking the device status from the code and the command line nvidia-smi, I can know that the model is indeed deploy on the GPU.

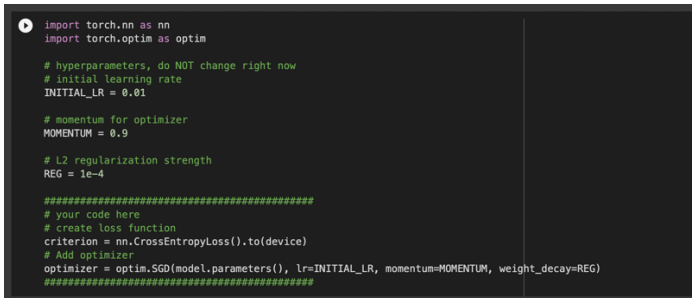


```
##########################################
# your code here
device = 'cuda' if torch.cuda.is_available() else 'cpu'
if device =='cuda':
    print("Run on GPU...")
else:
    print("Run on CPU...")

##########################################
```

```
Run on GPU...
```

```
[16] !nvidia-smi

Sun Sep 25 15:50:15 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   70C    P0    32W /  70W |    610MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

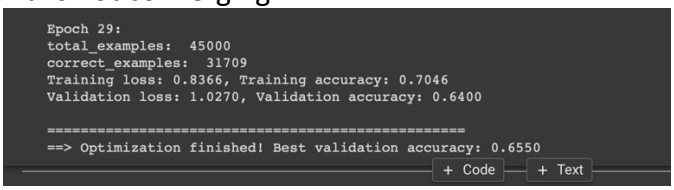
e. Problem 2.5:



```
import torch.nn as nn
import torch.optim as optim

# hyperparameters, do NOT change right now
# initial learning rate
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4

##########################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss().to(device)
# Add optimizer
optimizer = optim.SGD(model.parameters(), lr=INITIAL_LR, momentum=MOMENTUM, weight_decay=REG)
##########################################
```

f. Problem 2.6: Shown in the code

g. Problem 2.7: From the result, we can see that the initial loss of the training part is 1.9319. We know that if the accuracy is low, the loss is big. And we know that with more classes to classify, the model takes more time to converge, as a result, we can say that the lower number of classes in the dataset, the higher the accuracy is and the lower the loss is for the initial state. Also the training accuracy and validation accuracy aren't the same, we can see that training accuracy raises much faster than validation accuracy and this should be real. Due to the fixed learning rate, the accuracy of the model is bouncing around the accuracy 66% and is not converging.



```
Epoch 29:
total_examples:  45000
correct_examples:  31709
Training loss: 0.8366, Training accuracy: 0.7046
Validation loss: 1.0270, Validation accuracy: 0.6400

================================================
==> Optimization finished! Best validation accuracy: 0.6550
```

+ Code + Text

h. Problem 2.8: By adding the decayrate of 10 and decay of 0.1, we can see that the learning rate changes every 10 epochs, and we can see that from the result is we have a much higher training and validation accuracy.

```
Epoch 29:
total:  45000
correct:  35057
Training loss: 0.6328, Training accuracy: 0.7790
Validation loss: 0.9205, Validation accuracy: 0.6922

================================================
==> Optimization finished! Best validation accuracy: 0.6994
```

3. Lab(2) Improving the training pipeline
    a. Problem 3.1: At first, by adding RandomCrop into data augmentation, the training speed for each epoch decreases. The accuracy went up a little bit, the reason of this could be that I'd already added RandomVerticalFlip and RandomHorizontalFlip to my data, as a result adding another augmentation RandomCrop didn't result in huge gap in my accuracy.

```
Epoch 29:
total:  45000
correct:  30454
Training loss: 0.9225, Training accuracy: 0.6768
Validation loss: 0.9675, Validation accuracy: 0.6594

================================================
==> Optimization finished! Best validation accuracy: 0.6594
```

    b. Problem 3.2:
        i. After adding Batch Normalization layer after each convolution layer, the result is that the accuracy went up a bit, from 65% to 66%, which satisfies Lec6 P23 that BN gives about 2% increase in performance.

```
Epoch 29:
total:  45000
correct:  30654
Training loss: 0.9065, Training accuracy: 0.6812
Validation loss: 0.9511, Validation accuracy: 0.6588

================================================
==> Optimization finished! Best validation accuracy: 0.6608
```

        ii. By changing the initial learning rate from 0.01 to 0.1, we can see that the training accuracy went higher and thus showing that batch normalization allows a larger learning rate.

```
Epoch 29:
total:  45000
correct:  31359
Training loss: 0.8555, Training accuracy: 0.6969
Validation loss: 0.9322, Validation accuracy: 0.6720

================================================
==> Optimization finished! Best validation accuracy: 0.6830
```

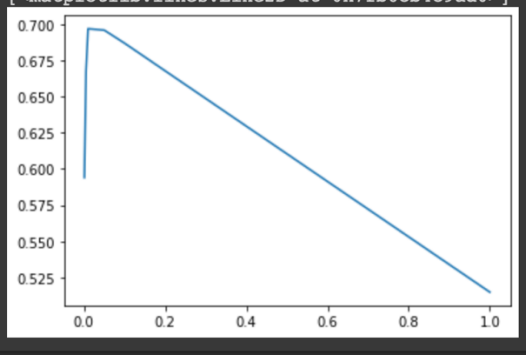
        iii. Yes, by changing our activation function from ReLU to Swish, the accuracy will increase and we got 70% accuracy after training the model with Swish activation function.

```python
def forward(self, x):
    def swish(pre):
      return pre * F.sigmoid(pre)

    # out = F.relu(self.bn1(self.conv1(x)))
    out = swish(self.bn1(self.conv1(x)))
    out = F.max_pool2d(out, 2)
    # out = F.relu(self.bn2(self.conv2(out)))
    out = swish(self.bn2(self.conv2(out)))
    out = F.max_pool2d(out, 2)
    out = out.view(out.size(0), -1)
    # out = F.relu(self.fc1(out))
    out = swish(self.fc1(out))
    # out = F.relu(self.fc2(out))
    out = swish(self.fc2(out))
    # out = F.relu(self.fc3(out))
    out = swish(self.fc3(out))
    return out
```

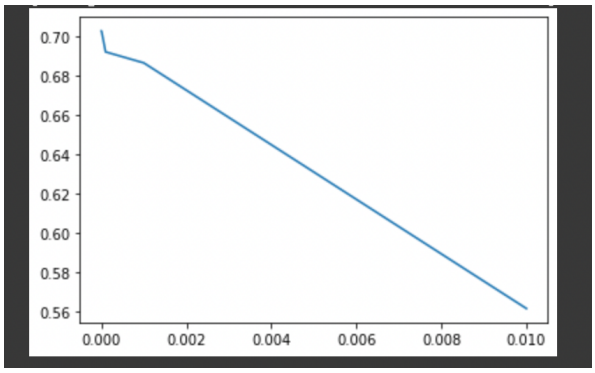
```
Epoch 29:
total:  45000
correct:  32963
Training loss: 0.7575, Training accuracy: 0.7325
Validation loss: 0.8560, Validation accuracy: 0.7032

================================================
==> Optimization finished! Best validation accuracy: 0.7060
```

c. Problem 3.3:
   i. As we can see in the plot below, neither the bigger or the smaller learning rate will result in the best accuracy, the highest accuracy happens at learning rate 0.05 and 0.01. As a result, we can say that there would be a maximum training accuracy between the highest and the lowest learning rate.

```
Done with training all the learning rate
[<matplotlib.lines.Line2D at 0x7fb6cb4c9ad0>]
```



   ii. From the plot we can see that the accuracy increases when the regularization strength decreases. The highest accuracy is the lowest when the regularization strength is the highest, vice versa.



4. Lab(3) Advance CNN Architecture

   From my self-implemented ResNet20 model, I realize that in the paper, the author had a softmax layer at the end of the model. However, in my experience getting rid of the softmax layer gives me a higher accuracy, which brings the validation accuracy up to 75%. Also, I realize that if I add a BN layer after every convolution layer, my model will have a very high training accuracy but a very low validation accuracy, this might result in over-fitting the model, as a result I chose only to add BN layer after the first convolution layer in my Blue, Green, and Red blocks.

   During training, I notice that whenever I decay my learning rate, such as from 0.1 to 0.05, the accuracy will have a jump increase. That's why I start conducting experiments on decay rate and learning rate, trying to find the best combination that will result in the best validation accuracy. An initial guess of mine is that currently I'm using a linear learning rate model, which the learning decay half for every number of epochs. I'm wondering would the performance increase if I use a non-linear learning rate model, such that the learning rate decays in an exponential way of even in a oscillate way that increases and decreases according to the currently accuracy.

   Another approach is I try to adjust different optimizer, such as Adam, RAdam, but the result shows that both optimizer converges at around 83% of accuracy, whereas using SGD the result would be higher. I thought SGD would easily get caught into local minimum and thus result in a lower accuracy, but the result shows that adam actually has a lower accuracy.