Assignment 3

Mustafa Emre Başar, 21726999
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b21726999@cs.hacettepe.edu.tr

May 6, 2021

1 Introduction

This assignment consist of two parts. In the first part, we were supposed to implement a CNN and train it to observe the hyperparameter effects on model. In the second part, we were expected to work on a pre-trained model and examine the what kind of differences can layers can make.

Detailed explanation of steps are examined in further sections.

2 Test and Train Data Set Creation

Data set creation was made manually. I tried to pick images that have different kind of features to increase flexibility. I used 80 percent of the images for train, 10 percent for the validation and rest for the test data set.

Sample counts shown below;

	Train	Test	Validation
airport_inside	61	44	61
artstudio	112	41	14
bakery	324	61	40
bar	483	61	60
bathroom	157	20	20
bedroom	529	67	66
bookstore	304	38	38
bowling	170	22	21
buffet	88	12	11
casino	412	52	51
church_inside	144	18	18
classroom	90	12	11
closet	108	13	14
clothingstore	84	11	11
computerroom	91	12	11
Total	3582	454	447

3 Pre Process

Before applying any kind of model, I first transformed the data with transformer that pytorch includes. I used resize to reduce the redundant data and create a convenient data set. Here RandomHorizontalFlip was to increase unique number of images.ToTensor is used since pytorch works with tensor data type. Finally normalizing each image to get rid of peak values.

Corresponding code snippet ;

```
transformer = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5],
                         [0.5, 0.5, 0.5])
])
```

After preprocess and loading the data, I created my CNN.

4 Part 1

4.1 Architecture

For my CNN I used 5 convolutional ,2 pooling and 2 fully connected layers. I used ReLU activation function since it provides a better result than other functions as it was mentioned in the class.

My kernel size choice was 3 since I think it is the optimum for good accuracy. Stride and padding was set to 1 according to the choice of kernel size.

```
class ConvNet(nn.Module):
 def __init__(self, num_classes=15):
    super(ConvNet, self).__init__()
    # in shape = (256,3,128,128)
    self.conv1 = nn.Conv2d(in_channels=3, out_channels=12, 3,1,1)
    self.bn1 = nn.BatchNorm2d(num_features=12)
    self.relu1 = nn.ReLU()
    self.pool = nn.MaxPool2d(kernel_size=2)
    # layer2
    self.conv2 = nn.Conv2d(in_channels=12,out_channels=16,3,1,1)
    # shape(256?, 20, 64, 64)
    self.relu2 = nn.ReLU()# shape(256?, 20, 64, 64)
    # layer3
    self.conv3 = nn.Conv2d(in_channels=16, out_channels=20, 3,1, 1)
    # shape(256?, 32, 64, 64)
    self.bn3 = nn.BatchNorm2d(num_features=20)
    # shape(256?, 32, 64, 64)
    self.relu3 = nn.ReLU()
    # layer4
    self.conv4 = nn.Conv2d(in_channels=20,out_channels=32,3, 1,1)
    self.bn4 = nn.BatchNorm2d(num_features=32)
    self.relu4 = nn.ReLU()
    # layer5
    self.conv5 = nn.Conv2d(in_channels=32,out_channels=64,3, 1,1)
    self.bn5 = nn.BatchNorm2d(num_features=64)
    self.relu5 = nn.ReLU()
    ##
    self.fc =nn.Linear(in_features=64*32*32, out_features=1000)
    self.fc2 = nn.Linear(in_features=1000, out_features=num_classes)
```

I first choice in and out channels as:

$$3 \to 16 - -16 \to 32 - -32 \to 64 - -64 \to 128 - -128 \to 256$$

for layers 1 to 5 respectively but since it resulted in bad accuracy, I tried the reduce the range between in and out layer I end with with better accuracy with ;

$$3 \to 12 - -12 \to 16 - -16 \to 20 - -20 \to 32 - -32 \to 64$$

For forwarding the data we were expected to use residual connections for one model.

```
    def forward(self, input):
        output = self.conv1(input)
        output = self.bn1(output)
        output = self.relu1(output)

        output = self.pool(output)

        output = self.conv2(output)
        output = self.relu2(output)

        output = self.conv3(output)
        output = self.bn3(output)
        output = self.relu3(output)

        ##
        output = self.conv4(output)
        output = self.bn4(output)
        output = self.relu4(output)

        output = self.pool(output)

        output = self.conv5(output)
        output = self.bn5(output)
        output = self.relu5(output)
        ##
        output = output.view(-1, 64*32*32)

        output = self.dropout(output)
        output = self.fc(output)
        output = self.fc2(output)
```

Here I connect every layer with next one to fully observe the difference residual connection can make and I can say that i better in the aspect of time.

Here is the non-residual forwarding;

```
self.non_resi_model = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=12, 3, 1,1),
    nn.BatchNorm2d(num_features=12),
    nn.ReLU(),

    nn.MaxPool2d(kernel_size=2),

    nn.Conv2d(in_channels=12, out_channels=16, 3, 1, 1),
    nn.ReLU(),

    nn.Conv2d(in_channels=16, out_channels=20,3,1,1),
    nn.BatchNorm2d(num_features=20),
    nn.ReLU(),
    ##
    nn.Conv2d(in_channels=20,out_channels=32,3, 1,1),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),

    nn.MaxPool2d(kernel_size=2),

    nn.Conv2d(in_channels=32,out_channels=64,3,1,1),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),
    ##
    nn.Flatten(),

    nn.Linear(in_features=64*32*32, out_features=1000),
    nn.Linear(in_features=1000, out_features=num_classes),

    ).to(device)
```

Architecture for both models are the sane, just forwarding differs.

4.2 Training

Graphs for validation accuracy on different batch sizes and learning rates for both models.

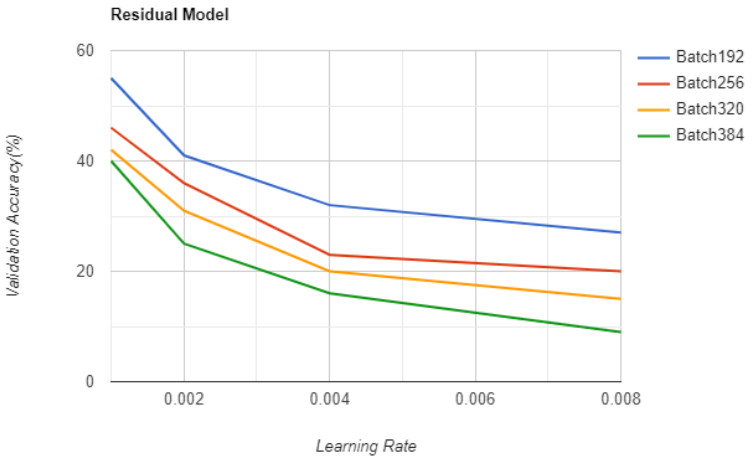

Figure 1: Residual Model

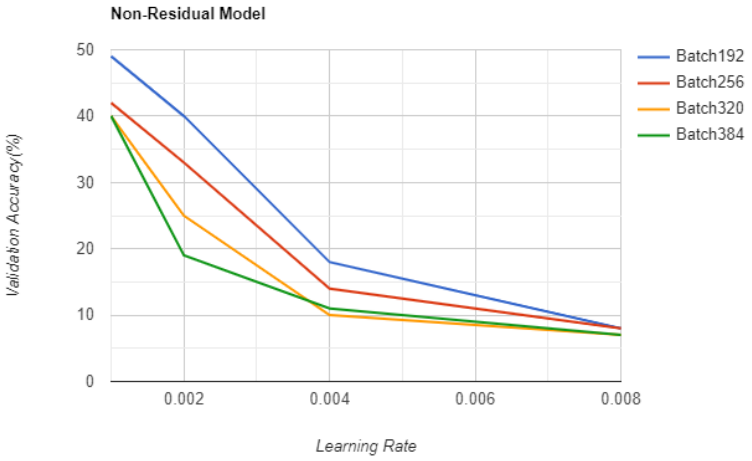

Figure 2: Non-Resiudal Model

As it can be seen, best model for both approach was low learning rate and batch size so I picked them to integrate dropout.

I applied dropout just before the full connected layers, since in and out difference of FC layers are so high, to prevent the over fitting, I chose this place to apply dropout.

I applied two dropout with two different probabilities, 0.25 and 0.50.

RESIDUAL

Without Dropout

Best Validation accuracy : %55 and corresponding Test accuracy : %54

With Dropout

0.25:Best Validation accuracy : %57 and relevant Test accuracy : %55

0.50:Best Validation accuracy : %53 and relevant Test accuracy : %51

NON-RESIDUAL

Without Dropout

Best Validation accuracy : %49 and corresponding Test accuracy : %47

With Dropout

0.25:Best Validation accuracy: %48 and relevant Test accuracy: %49

0.50:Best Validation accuracy: %44 and relevant Test accuracy: %47

It seems that the dropout application does not make that much of a difference although there are small increases in accuracies. The thing is, even there are not much differences it can be seen that the effect of 0.5 value is respectively bigger than 0.25. The reason behind that result might be the place where the dropout is applied, with a better location, we might get better accuracy and also probability value might be the case.

4.3 Evaluation

To use the model that I created, code snippet for execution was like following ;

```
model = ConvNet(num_classes=15).to(device)

optimizer = Adam(model.parameters(),lr=0.001, weight_decay=0.0001)
loss_function = nn.CrossEntropyLoss()

num_epochs = 30

train_count = len(glob.glob(train_path+'/**/*.jpg'))
test_count = len(glob.glob(test_path+'/**/*.jpg'))
validation_count = len(glob.glob(validation_path+'/**/*.jpg'))
```

Here I picked the entropy and optimizer functions according to research of mine. It said that the most appropriate result are gathered with these two functions.

My train, validation and test accuracy calculations;

```
for epoch in range(num_epochs):
    model.train()
    train_accuracy = 0.0
    train_loss = 0.0
    for i, (images,labels) in enumerate(train_loader):
        if torch.cuda.is_available():
            images= Variable(images.cuda())
            labels = Variable(labels.cuda())

        optimizer.zero_grad()
        outputs = model(images)
        loss = loss_function(outputs,labels)
        loss.backward()
        optimizer.step()
        train_loss+=loss.cpu().data*images.size(0)
        _,prediction = torch.max(outputs.data, 1)
        train_accuracy+= int(torch.sum(prediction==labels.data))

    train_accuracy = train_accuracy/train_count
    train_loss = train_loss/train_count
```

```
    model.eval()

    validation_accuracy = 0.0
    for i, (images, labels) in enumerate(validation_loader):
        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())

        outputs = model(images)
        _, prediction = torch.max(outputs.data, 1)
        validation_accuracy+=int(torch.sum(prediction==labels.data))
    validation_accuracy = validation_accuracy / validation_count
    test_accuracy = 0.0
    for i, (images,labels) in enumerate(test_loader):
        if torch.cuda.is_available():
            images= Variable(images.cuda())
            labels = Variable(labels.cuda())

        outputs=model(images)
        _,prediction= torch.max(outputs.data, 1)
        test_accuracy+= int(torch.sum(prediction==labels.data))
    test_accuracy = test_accuracy/test_count
```

And to code snippet to save the model model:

```
if(validation_accuracy> best_accuracy):
        torch.save(model.state_dict(),'best_checkpoint.model')
        best_accuracy=validation_accuracy
```

And the corresponding output for evaluation steps ;

```
Epoch 0 Train Loss: 107 Train Accuracy: 0.14014517029592408 Validation Accuracy: 0.14757709251101322 Test Accuracy: 0.14757709251101322
Epoch 1 Train Loss: 31 Train Accuracy: 0.17839195979899497 Validation Accuracy: 0.14537444933920704 Test Accuracy: 0.14757709251101322
Epoch 2 Train Loss: 9 Train Accuracy: 0.211892797319933 Validation Accuracy: 0.14757709251101322 Test Accuracy: 0.14757709251101322
Epoch 3 Train Loss: 3 Train Accuracy: 0.2434394193188163 Validation Accuracy: 0.14096916299559473 Test Accuracy: 0.1145374449339207
Epoch 4 Train Loss: 2 Train Accuracy: 0.2995533221663874 7 Validation Accuracy: 0.2444933920704846 Test Accuracy: 0.2621145374449339
Epoch 5 Train Loss: 1 Train Accuracy: 0.39838079285315464 Validation Accuracy: 0.3700440528634361 Test Accuracy: 0.3854625550660793
Epoch 6 Train Loss: 1 Train Accuracy: 0.4433277498604132 Validation Accuracy: 0.3634361233480176 Test Accuracy: 0.40308370044052866
Epoch 7 Train Loss: 1 Train Accuracy: 0.46705750977107763 Validation Accuracy: 0.394273127753304 Test Accuracy: 0.43612334801762115
Epoch 8 Train Loss: 1 Train Accuracy: 0.5011166945840313 Validation Accuracy: 0.42951541850220265 Test Accuracy: 0.3964757709251101
Epoch 9 Train Loss: 1 Train Accuracy: 0.5290340591848129 Validation Accuracy: 0.43612334801762115 Test Accuracy: 0.43392070484581496
Epoch 10 Train Loss: 1 Train Accuracy: 0.5418760469011725 Validation Accuracy: 0.4581497797356828 Test Accuracy: 0.460352422907489
Epoch 11 Train Loss: 1 Train Accuracy: 0.5619765494137353 Validation Accuracy: 0.43392070484581496 Test Accuracy: 0.42951541850220265
Epoch 12 Train Loss: 1 Train Accuracy: 0.6060859854829704 Validation Accuracy: 0.44273127753303965 Test Accuracy: 0.4251101321585903
Epoch 13 Train Loss: 1 Train Accuracy: 0.6395868230039085 Validation Accuracy: 0.4647577092511013 Test Accuracy: 0.4647577092511013
Epoch 14 Train Loss: 1 Train Accuracy: 0.6708542713567839 Validation Accuracy: 0.4185022026431718 Test Accuracy: 0.45374449339207046
Epoch 15 Train Loss: 0 Train Accuracy: 0.7124511446119486 Validation Accuracy: 0.3964757709251101 Test Accuracy: 0.42290748898678415
Epoch 16 Train Loss: 0 Train Accuracy: 0.7283640424343942 Validation Accuracy: 0.4581497797356828 Test Accuracy: 0.4889867841409692
Epoch 17 Train Loss: 0 Train Accuracy: 0.7632607481853713 Validation Accuracy: 0.44713656387665196 Test Accuracy: 0.4889867841409692
Epoch 18 Train Loss: 0 Train Accuracy: 0.7984366275823562 Validation Accuracy: 0.36123348017621143 Test Accuracy: 0.3876651982378855
Epoch 19 Train Loss: 0 Train Accuracy: 0.8344500279173646 Validation Accuracy: 0.4052863436123348 Test Accuracy: 0.43832599118942733
Epoch 20 Train Loss: 0 Train Accuracy: 0.8313791178112786 Validation Accuracy: 0.4185022026431718 Test Accuracy: 0.3964757709251101
Epoch 21 Train Loss: 0 Train Accuracy: 0.8241206030150754 Validation Accuracy: 0.34801762114537443 Test Accuracy: 0.394273127753304
Epoch 22 Train Loss: 0 Train Accuracy: 0.7945281965382468 Validation Accuracy: 0.3876651982378855 Test Accuracy: 0.44052863436123346
Epoch 23 Train Loss: 0 Train Accuracy: 0.8157453936348409 Validation Accuracy: 0.2356828193832599 Test Accuracy: 0.29955947136563876
Epoch 24 Train Loss: 0 Train Accuracy: 0.8364042434394193 Validation Accuracy: 0.3854625550660793 Test Accuracy: 0.4251101321585903
Epoch 25 Train Loss: 0 Train Accuracy: 0.9014517029592406 Validation Accuracy: 0.4251101321585903 Test Accuracy: 0.46255506607929514
Epoch 26 Train Loss: 0 Train Accuracy: 0.9293690675600224 Validation Accuracy: 0.3722466960352423 Test Accuracy: 0.43392070484581496
Epoch 27 Train Loss: 0 Train Accuracy: 0.9631490787269682 Validation Accuracy: 0.43392070484581496 Test Accuracy: 0.4889867841409692
Epoch 28 Train Loss: 0 Train Accuracy: 0.9676158570630933 Validation Accuracy: 0.3920704845814978 Test Accuracy: 0.43171806167400884
Epoch 29 Train Loss: 0 Train Accuracy: 0.9734785036292574 Validation Accuracy: 0.42951541850220265 Test Accuracy: 0.4713656387665198
```

Figure 3: Output

4.4 Conclusion for Part 1

First thing I learned is that training a model is very hard and painful thing since it takes hours.Other than that, about model creation and training, I pretty much end up with what I expected. Smaller batch size provide a better accuracy since it examine it in more detail.Other thing that the learning rate that it effects the model same way the batch size does , smaller is better.

5 Part 2

First of all, fine tuning is a technique that use the weights of a already trained model to train a new model. It uses the weights of the pre-trained model to give initial weights to new model. The new model starts with those weights and tries to improve it if it possible. It provides a better training in the aspect of time.

Freezing is an improvement based on fine tuning. Since we have the weights of a already trained model, with freezing, we completely take out the weight computations of desired layers and get an even better performance.

I guess the reason we train only FC layer is, since number of classes can differ from data set to data set, by keeping the FC layer active, we can use the model with our data set by modifying the number of out channels. Simply it provides a generic way of evaluating data sets.

5.1 Training

First of all I get the necessary model and froze all the layers with the following code snippet;

```
model = torchvision.models.resnet18(pretrained=True)

for param in model.parameters():
    param.requires_grad = False
```

Then only activate FC layer;

```
num_ftrs = model.fc.in_features
model.fc = model.fc = nn.Linear(num_ftrs, num_classes)
```

Left the in channels as it is and out channels is set to number of classes.

And the resulting output of just FC layer is as follows;

```
Cost at epoch 0 is 1.70420
Cost at epoch 1 is 1.01407
Cost at epoch 2 is 0.82002
Cost at epoch 3 is 0.73010
Cost at epoch 4 is 0.67436
Test Accuracy
Got 3026 / 3582 with accuracy 84.48
Validation Accuracy
Got 331 / 447 with accuracy 74.05
Test Accuracy
Got 342 / 454 with accuracy 75.33
```

Figure 4: Output

After that, I activated the last convolutional layer ;

```
model.layer4.requires_grad_(True)
```

Here the layer 4 corresponds to the last layer.
And the resulting output of just FC + Convolutional layer is as follows;

```
Cost at epoch 0 is 0.87073
Cost at epoch 1 is 0.36653
Cost at epoch 2 is 0.23318
Cost at epoch 3 is 0.18399
Cost at epoch 4 is 0.11447
Train Accuracy
Got 3539 / 3582 with accuracy 98.80
Validation Accuracy
Got 371 / 447 with accuracy 83.00
Test Accuracy
Got 387 / 454 with accuracy 85.24
```

Figure 5: Output

From the outputs, it can be seen that extra convolutional layer made quite a difference and it was what I expected since it increase the calculation of weights of that layer and modify it according to the data set, it ends up with better accuracy.

5.2 Comparison

At the beginning of this assignment I was expecting to get better accuracy with pre-trained CNN model since it has more layers with better architecture. The results I get proved the same idea.

Best of Part 1 : %57
Best of Part 2 : %83

The reason behind that difference can be a couple things. First, number of in and out channels of convolutional layers can make difference since it effects the filtering. Better filtering gives better accuracy. I experienced this even in the Part 1 with different numbers of in and out channels, I mentioned about it in the Part 1 section.

Also pooling and normalization layers, dropouts can make a difference.

Simply, since it has a better architecture and more accurate weights, it provides a better accuracy.

6 References

[1] https://github.com/gaurav67890/Pytorch_Tutorials/blob/master/cnn-scratch-training.ipynb
[2] https://discuss.pytorch.org/t/add-residual-connection/20148/6
[3] https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7
[4] https://wandb.ai/authors/ayusht/reports/Dropout-in-PyTorch-An-Example--VmlldzoxNTgwOTE
[5] https://medium.com/swlh/deep-learning-for-image-classification-creating-cnn-from-scratch-using-pytorch-d9eeb7039c12
[6] https://www.pluralsight.com/guides/introduction-to-resnet
[7] https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/Basics/pytorch_pretrain_finetune.py
[8] https://discuss.pytorch.org/t/how-the-pytorch-freeze-network-in-some-layers-only-the-rest-of-the-training/7088
[9] https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html