CSCI 677: HW4

Nisha Tiwari | email: nishatiw@usc.edu | USC Id: 7888495181

Results

All the models mentioned below were trained over the same dataset (90,000 + 30,000 (using augmentation)). Starting learning rate was fixed to 0.001 and decreased to 0.0005 or 0.0001 for higher epoch values. Batch size was fixed to 64.

Model	**Modifications wrt Base Model**	**Accuracy (Test)**	**Accuracy (Validation)**	**#epochs required**
Best	-Smaller filter size -Batch Norm -Increased number of feature maps -Increased #parameters for fully connected layers	64.38	64.66	5
Stepping stone to Best	-Smaller filter size	52.25	54.51	10
Base	NA	50.73	51.22	16

Based on the experiments over multiple parameters over the base model, I can make the following conclusions:

1. As we decrease the filter size, the accuracy tends to increase.
2. Adding batch normalization and increasing the number of feature maps boosts the classifier performance by ~10%
3. Most of the examples are mis-classified as a category which has some common features with the original category. For example, dog being classified as cat or vice versa.
4. Decreasing the learning rate for higher number of epochs boosts the accuracy to some extent.
5. Weight initialization using Xavier uniform distribution did not help in boosting the performance.
6. Model with higher number of weights tends to trains faster than the base model.

Variants of the base model

I tried various models with different filter sizes, number of feature maps, and application of Batch Norm. Here, I first present the model with best performance, followed by other variants which I tried and at the end the original LeNet-5 model’s performance.

Best Model

The best model achieved an accuracy of `64.38%`.

Model Definition

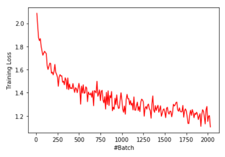
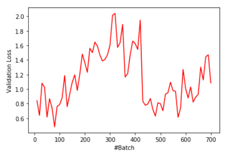
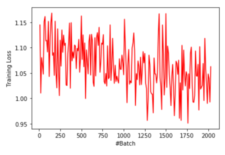
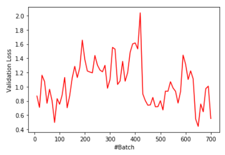
The changes compared to the base model are highlighted in bold.

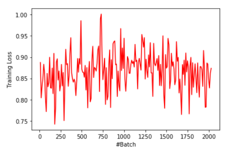
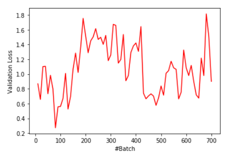
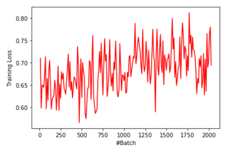
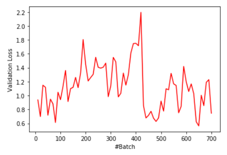
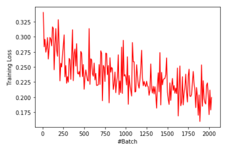
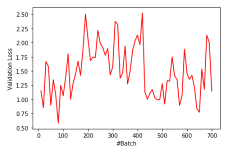
```
(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc1): Linear(in_features=2304, out_features=600, bias=True)
  (fc2): Linear(in_features=600, out_features=200, bias=True)
  (fc3): Linear(in_features=200, out_features=10, bias=True)
```

Results

Table 1 shows the progression of accuracy and loss with the number of epochs

Table 1 (Best Model's accuracy with the number of epochs)

Learning rate	**Epoch**	**Training Loss Function (Batch size 64)**	**Validation Loss Function (Batch size 128)**	**Accuracy on Validation Set(%)**
0.001	1			59.15
0.001	2			62.62

0.001	3			62.79
0.001	4			63.07
0.0001	5			64.66

Confusion Matrix

I present here the confusion matrix for the best model. Bold number shows the correct classification for each class.

1. Numbers in red shows the high number of mis-classfications for a class as some other class.
2. As we can observe, the automobile is classified as a truck around 1646 times. Considering the similarity between automobile and truck category, these errors make sense. Similarly, cat as a dog and dog as a cat has been incorrectly classified a lot of times.

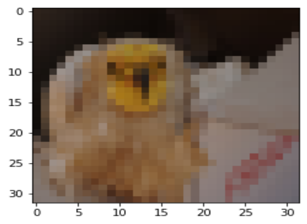
```
       [ aero, auto, bird,  cat, deer, dog,  frog, horse,ship, truck]
aero   [ 6724,  238,  392,  114,  154, 112,   52,  142,  758,  260  ]
auto   [ 260, 6525,    77,  130,   70,  125,   53,  125,  445, 1646  ]
bird   [ 442,    94, 5479,  707,  651,  648,  585,  267,  356,   98  ]
cat    [ 105,    88,  603, 4299,  803, 1651,  595,  374,  166,  107  ]
deer   [ 143,    60,  593,  699, 5036,  915,  211,  748,  149,   99  ]
dog    [ 111,   122,  602, 1653,  935, 4274,  310,  742,  157,  131  ]
frog   [  67,    45,  679,  744,  335,  333, 7005,   55,  108,   42  ]
horse  [ 154,   109,  226,  282,  715,  595,   60, 6252,  142,  169  ]
ship   [ 741,   379,  280,  239,  210,  221,   93,  141, 6332,  441  ]
truck  [ 253, 1340,    69,  133,   91,  126,   36,  154,  387, 6007  ]
```

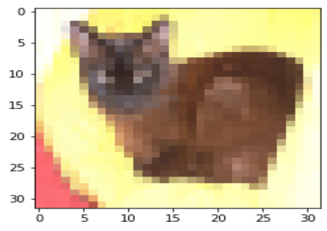
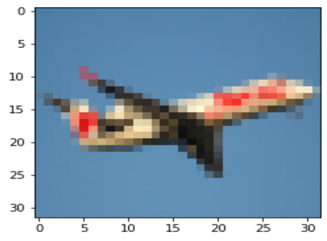
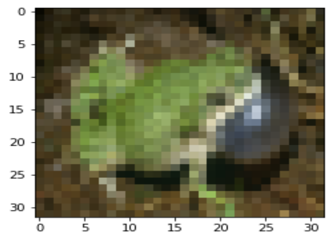
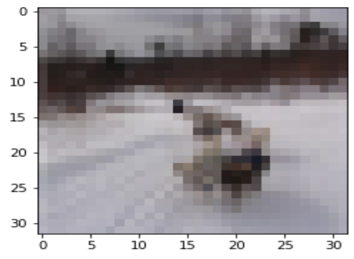
Per Class Accuracy

Class Name	Accuracy (%)
airplane	75.16
automobile	69.00
bird	58.74
cat	48.90
deer	58.20
dog	47.29
frog	74.42
horse	71.83
ship	69.75
truck	69.88

Mis-classified examples

Now, I present the following table with a few mis-classified examples by the best model.

Image	**Correct Label**	**Classified Label**
	Bird	Cat

	Cat	Deer
	Aeroplane	Bird
	Frog	Cat
	Dog	Bird

Stepping stones to the best model

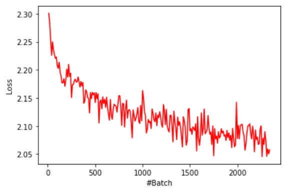
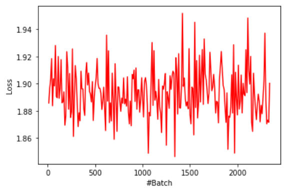
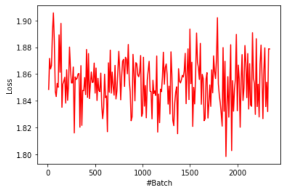
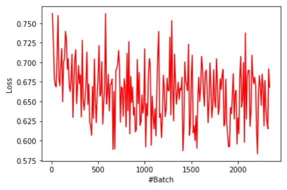
Here I present the model which gave me the first significant improvement over the base model. This model achieved an accuracy of `52.25%` which is close to the base model accuracy but in a fewer number of epochs.

```
(conv1): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
(pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(conv2): Conv2d(6, 16, kernel_size=(4, 4), stride=(1, 1))
```

```
(pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(fc1): Linear(in_features=576, out_features=120, bias=True)
(fc2): Linear(in_features=120, out_features=84, bias=True)
(fc3): Linear(in_features=84, out_features=10, bias=True)
```

Table 2 shows the progression of accuracy and loss with the number of epochs

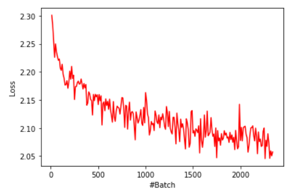
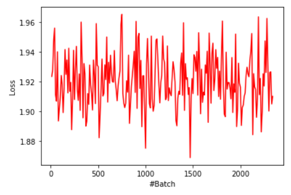
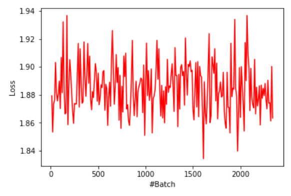
Table 2 (Stepping stone Model's accuracy with the number of epochs)

Learning rate	**Epoch**	**Loss function (training)**	**Accuracy (on validation set)**
0.001	1		41.06
0.001	6		51.15
0.001	9		52.16
0.0001	10		54.51

Base Model

The accuracy for the base model is `50.73%`

Table 3 (Base Model's accuracy with the number of epochs)

Learning rate	**Epoch**	**Loss function (training)**	**Accuracy (on validation set)**
0.001	1		39.36
0.0001	8		50.50
0.00001	16		51.22

Code Description

Data Extraction

To extract the tar.gz folder in Google Colab drive

```
import tarfile
tar = tarfile.open("/content/drive/My Drive/Colab Notebooks/CINIC-10.tar.gz")
tar.extractall()
tar.close()
```

Data Augmentation

I have used Augmentor library to augment the data. Augmentor provides methods to crop, scale, modify brightness, and flip an image among others. For the training, I chose to add 30,000 to 40,000 new images to the data.

```
import Augmentor
p = Augmentor.Pipeline("/content/train")
p.crop_centre(0.5,0.9)
```

```
p.flip_left_right(0.5)
p.scale(0.5,1.1)
p.zoom(0.5,1.1,1.3)
p.random_brightness(0.2,0.3,0.7)
p.resize(1,32,32)
p.sample(30000)
```

Mean and standard deviation Calculation for Normalization

After the new data is added to the training set, I have calculated the mean and standard deviation using the following code

```
mean = 0.
std = 0.
samples = 0.
for batch, labels in train_loader:
    data=torch.tensor(data)
    batch_samples = data.size(0)
    data = data.view(batch_samples, data.size(1), -1)
    mean += data.mean(2).sum(0)
    std += data.std(2).sum(0)
    samples += batch_samples
mean /= samples
std /= samples
```

DataLoader Code

I created data loaders for each train,valid and test dataset. As suggested in the assignment description, I have used batch size as 64 for training. I divided test and valid in batches of size 128 to utilize the parallelization of Pytorch data loaders. Each of the dataset is transformed to tensors and normalized based on the calculated mean and standard deviation from the training set.

```
TRAIN_DATA_PATH="/content/train"
data_transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.4598,
0.4520, 0.4116),(0.1902, 0.1869, 0.1896))])
train = torchvision.datasets.ImageFolder(root=TRAIN_DATA_PATH, transform=data_transform)
VLAID_DATA_PATH="/content/valid"
valid = torchvision.datasets.ImageFolder(root=VLAID_DATA_PATH, transform=data_transform)
TEST_DATA_PATH="/content/test"
test = torchvision.datasets.ImageFolder(root=TEST_DATA_PATH, transform=data_transform)
train_loader = torch.utils.data.DataLoader(train, batch_size=64, shuffle=True, num_workers=6)
valid_loader = torch.utils.data.DataLoader(valid, batch_size=128, num_workers=6)
test_loader = torch.utils.data.DataLoader(test,  batch_size=128,num_workers=6)
```

Model Creation

Following code is the implementation of the original Lenet-5 model.

```
import torch.nn as nn
import torch.nn.functional as Fn

class Model(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = Fn.relu(self.pool1(self.conv1(x)))
        x = Fn.relu(self.pool2(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = Fn.relu(self.fc1(x))
        x = Fn.relu(self.fc2(x))
        x= Fn.softmax(self.fc3(x))
        return x
```

Weight Initialization

I wrote the following code to initialize the weights of the model using xavier uniform distribution.

```
def init_weights(m):
    if type(m) == nn.Linear:
      torch.nn.init.xavier_uniform_(m.weight)
      torch.nn.init.constant(m.bias, 0)
    if isinstance(m, nn.Conv2d):
      torch.nn.init.xavier_uniform_(m.weight)
      torch.nn.init.constant(m.bias, 0)
model.apply(init_weights)
```

Training and Validation

Training

The following code creates an instance of the Model, defines the loss and optimizer function, trains the model for certain number of epochs, tracks the loss function (mean loss value for 10 batches) and accuracies, and plots the loss function vs #batch curve at the end of each epoch.

```
model=Model()
model.cuda()
max_epochs=10
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
training_accuracy=[]
validation_accuracy=[]
prev_accuracy=0.0
curr_accuracy=0.0
epoch_count=0
learning_rate=0.001
for epoch in range(max_epochs):
    epoch_count+=1
    loss_list=[]
    num_batch_list=[]
    num_batches=0
    final_loss=[]
    model.train()
    for local_batch, local_labels in train_loader:
      # Transfer to GPU
      local_batch, local_labels =  torch.autograd.Variable(local_batch.cuda()),
torch.autograd.Variable(local_labels.cuda())
      output =model(local_batch)
      loss = criterion(output, local_labels)
      loss_list.append(loss.item())
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      num_batches+=1
      if num_batches % 10 == 0:
            num_batch_list.append(num_batches)
            final_loss.append(np.mean(loss_list))
            loss_list.clear()
    # plotting training loss
    fig2 = pyplot.figure()
    pyplot.plot(num_batch_list, final_loss, 'r')
    pyplot.xlabel("#Batch")
    pyplot.ylabel("Training Loss")
    pyplot.show(fig2)
training_accuracy.append(accuracy/num_batches)
print("Training:::for epoch", epoch, "accuracy is:",accuracy/num_batches)
```

Measuring the performance of the Model for each epoch

For every epoch, once the training of the model is done, the following code runs the model (with the updated weights) for the validation dataset. It also keeps track of the loss for validation set and plots at the end of the code.

```
    accuracy=0.0
    num_batch_list=[]
    num_batches=0
    final_loss=[]
    loss_list=[]
    net.eval()
    for local_batch, local_labels in valid_loader:
      # Transfer to GPU
      local_batch=  torch.autograd.Variable(local_batch.cuda())

      # Model computations
      predicted_val =model(local_batch)
      loss = criterion(predicted_val, torch.autograd.Variable(local_labels.cuda()))
      loss_list.append(loss.item())
      predicted_val = predicted_val.cpu().data.numpy()
      predicted_val = np.argmax(predicted_val, axis = 1)  # retrieved max_values along every
row
      # accuracy
      current_accuracy=accuracy_score(local_labels.numpy(), predicted_val)
      accuracy += current_accuracy
      num_batches +=1
      if num_batches % 10 == 0:
            num_batch_list.append(num_batches)
            final_loss.append(np.mean(loss_list))
            loss_list.clear()
    # plotting validation loss
    fig2 = pyplot.figure()
    pyplot.plot(num_batch_list, final_loss, 'r')
    pyplot.xlabel("#Batch")
    pyplot.ylabel("Validation Loss")
    pyplot.show(fig2)
    curr_accuracy=accuracy/num_batches
    validation_accuracy.append(curr_accuracy)
    print("Validation:::for epoch", epoch, "accuracy is:",accuracy/num_batches)
```

Visualizing the accuracy vs #epochs curve

The following code is to visualize how accuracy changes with each epoch for training and validation set. This graph helps in deciding the number of epochs to use for training.

```
epochs = list(range(max_epochs))
# plotting training and validation accuracies
fig2 = pyplot.figure()
pyplot.plot(epochs, training_accuracy, 'r')
pyplot.plot(epochs, validation_accuracy, 'g')
pyplot.xlabel("Epochs")
pyplot.ylabel("Accuracy")
pyplot.show(fig2)
```

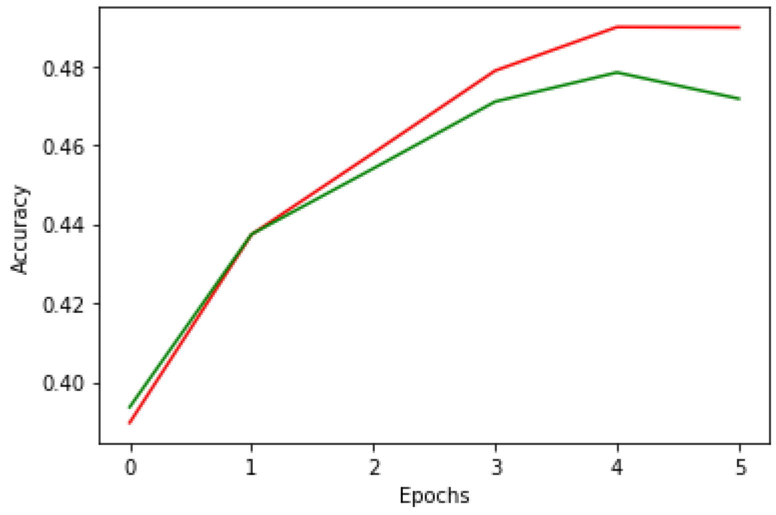
Finding accuracy for test dataset

Once the model is trained, following code runs the model to classify data for test dataset to get the final accuracy.

```
correct = 0
total = 0
accuracy=0.0
predicted_values=[]
real_values=[]
model.eval()
for inputs, actual_val in test_loader:
    total += 1
    # perform classification
    predicted_val =model(torch.autograd.Variable(inputs.cuda()))
    # convert 'predicted_val' GPU tensor to CPU tensor and extract the column with max_score
    predicted_val = predicted_val.cpu().data.numpy()
    predicted_val = np.argmax(predicted_val, axis = 1)  # retrieved max_values along every row
    predicted_values.extend(predicted_val)
    real_values.extend(actual_val.numpy())
    current_accuracy=accuracy_score(actual_val.numpy(), predicted_val)
    accuracy += current_accuracy

print("Classifier Accuracy on Test Dataset: ", accuracy/total * 100)
```

Following graph shows one of the results for the above code. Green curve is for validation and red for training.



Creating confusion matrix

```
import sklearn.metrics as metrics
conf_matrix = metrics.confusion_matrix(predicted_values, real_values)
```

Limitations

1. Given the images of 32x32 size, it is difficult to apply many layers of convolutions to the network since the size was decreasing rapidly.
2. Current model is unable to differentiate between similar categories very well.
3. Performance of the model is not comparable to other well-known image classification models. The reason could be less number of layers in LeNet architecture.

Discussion

1. Experimenting the model with different dataset, probably with higher resolution images would be interesting.
2. In the current experiments, train data set was increased by ~40%. Furthermore, I used only a few methods like cropping and flipping for augmentation. It will be insightful to see how the performance of the model changes as we increase the percentage of data augmentation and try various other methods (rotation by some angle).
3. We know that Lenet-5 gives really high accuracy for images of digits. The images which we used for our experiments are complex. This could be the reasoning for performance boosting as we increase the feature maps. The model would be able to learn more features with increased number of features.
4. The least accuracies were found for dog and cat classes. The probable reason is that since there are many feature similarities between dog and cat images, model is classifying dogs as cats and cats as dogs many times.