Assignment 04

(knowledge about learning rate parameters have been summed at the end)

1) Learning rate of 0.01 and decay of 0.999 was used for this question. (to maintain consistency for the following questions)

```
lr = 1.0e-2
lr_decay= 0.999
reg =5e-6
```

Figure 1 parameters of item 1

Accuracy was defined in the following way:

```
def accuracy(labels,preds):
  real_cls=np.argmax(labels,axis=1)
  pred_cls=np.argmax(preds, axis=1)
  valid_pred=[pred_cls==real_cls]
  acc=100*np.sum(valid_pred)/len(real_cls)
  return acc
```

Figure 2 Accuracy definition used

Important part of the code

```
for t in range(iterations):
  # Forward pass
  y_pred=np.dot(x_train,w1)+b1
  train_loss=np.sum((y_pred-y_train)**2)/Ntr + reg*np.sum(w1**2)
  train_loss_history.append(train_loss)
  test_loss=np.sum((np.dot(x_test,w1)+b1-y_test)**2)/Nte + reg*np.sum(w1**2)
  test_loss_history.append(test_loss)

  # Backward pass
  dw1 = 2*(1/Ntr)*(x_train.T.dot(y_pred - y_train))  + 2*reg*w1
  w1 = w1 - lr*dw1
  I=np.ones((Ntr,1))
  db1=2*(1/Ntr)*(I.T.dot(y_pred - y_train))
  b1=b1-lr*db1

  #Accuracies
  train_acc = accuracy(y_train,y_pred)
  #train_acc=1.0-(1/Ntr)*(np.absolute(np.argmax(y_train,axis=1)-np.argmax(y_pred,axis=1))).sum()/K
  train_acc_history.append(train_acc)
  test_acc = accuracy( y_test, np.dot(x_test,w1)+b1)
  #test_acc=1.0-(1/Nte)*(np.absolute(np.argmax(y_test,axis=1)-np.argmax(np.dot(x_test,w1)+b1,axis=1))).sum()/K
  val_acc_history.append(test_acc)
  lr=lr*lr_decay
```

Figure 3 Forward propagation, backward propagation and accuracy definition

From the printed output we can see that the learning rate and the losses reduce with each epoch while the accuracies keep on increasing to reach a certain point.

```
0.01
0.999
Epoch 000 - Train_Loss 1.0000 - Training Accuracy: 10.312 - Testing Accuracy: 24.850 - Learning Rate: 0.00999
Epoch 001 - Train_Loss 0.9625 - Training Accuracy: 24.404 - Testing Accuracy: 28.600 - Learning Rate: 0.00997
Epoch 030 - Train_Loss 0.8403 - Training Accuracy: 37.400 - Testing Accuracy: 37.450 - Learning Rate: 0.00941
Epoch 060 - Train_Loss 0.8115 - Training Accuracy: 38.862 - Testing Accuracy: 38.760 - Learning Rate: 0.00886
Epoch 090 - Train_Loss 0.8007 - Training Accuracy: 39.616 - Testing Accuracy: 39.230 - Learning Rate: 0.00834
Epoch 120 - Train_Loss 0.7958 - Training Accuracy: 40.116 - Testing Accuracy: 39.510 - Learning Rate: 0.00786
Epoch 150 - Train_Loss 0.7930 - Training Accuracy: 40.452 - Testing Accuracy: 39.610 - Learning Rate: 0.00740
Epoch 180 - Train_Loss 0.7911 - Training Accuracy: 40.678 - Testing Accuracy: 39.610 - Learning Rate: 0.00697
Epoch 210 - Train_Loss 0.7897 - Training Accuracy: 40.920 - Testing Accuracy: 39.740 - Learning Rate: 0.00656
Epoch 240 - Train_Loss 0.7887 - Training Accuracy: 41.102 - Testing Accuracy: 39.840 - Learning Rate: 0.00618
Epoch 270 - Train_Loss 0.7878 - Training Accuracy: 41.208 - Testing Accuracy: 39.960 - Learning Rate: 0.00582
Epoch 299 - Train_Loss 0.7871 - Training Accuracy: 41.298 - Testing Accuracy: 40.030 - Learning Rate: 0.00549
```

Figure 4 Processing

Graphed Output

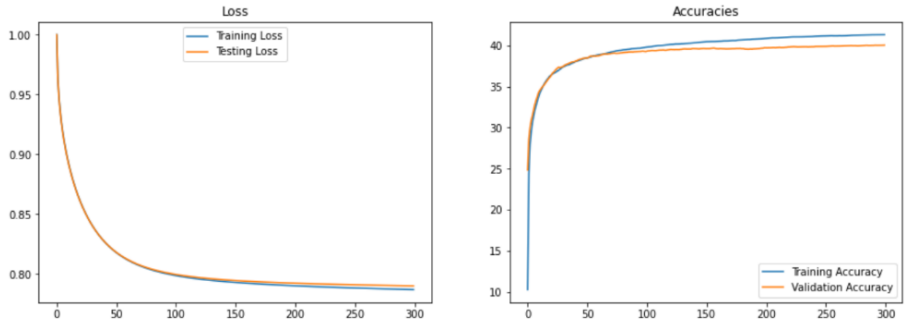

Figure 5 Losses and Accuracies

Following images were obtained for the weight matrix.

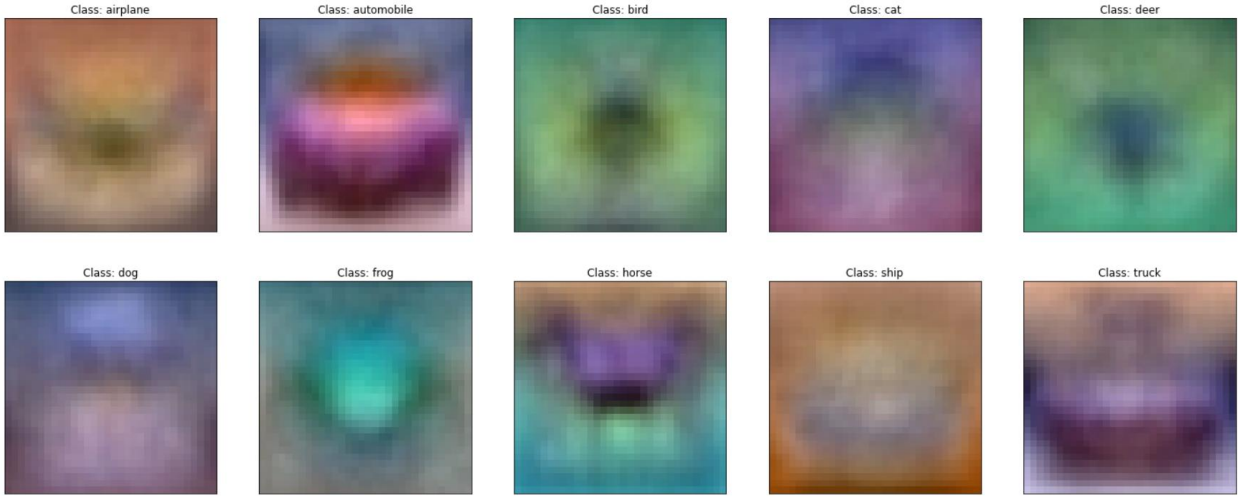

Figure 6 Weight matrix Images

2) Compared to item 1 this takes more time to train the model since two layers are used. And we can see that more accuracy could be gained here. Initial learning rate has been set 0.01 as the previous part for comparison purpose.

```
lr=1.0e-2
lr_decay = 0.999
reg = 5e-6
```

Figure 7 Parameters for item 2

```
h=1.0/(1.0+np.exp(-(x.dot(w1)+b1)))
y_pred = h.dot(w2)+b2
train_loss = (1./batch_size)*np.square(y_pred-y).sum() + reg*(np.sum(w2*w2)+np.sum(w1*w1))
```

Figure 8 Sigmoid

Output of the code of item 2.

```
Epoch 000 - Train Loss 1.0000 - Training Accuracy: 10.000 - Testing Accuracy: 10.000 - Learning Rate: 0.01000
Epoch 030 - Train Loss 0.8358 - Training Accuracy: 29.098 - Testing Accuracy: 28.970 - Learning Rate: 0.00970
Epoch 060 - Train Loss 0.8211 - Training Accuracy: 32.338 - Testing Accuracy: 32.380 - Learning Rate: 0.00942
Epoch 090 - Train Loss 0.7947 - Training Accuracy: 37.040 - Testing Accuracy: 37.000 - Learning Rate: 0.00914
Epoch 120 - Train Loss 0.7864 - Training Accuracy: 40.434 - Testing Accuracy: 40.080 - Learning Rate: 0.00887
Epoch 150 - Train Loss 0.7792 - Training Accuracy: 41.550 - Testing Accuracy: 40.780 - Learning Rate: 0.00861
Epoch 180 - Train Loss 0.7656 - Training Accuracy: 43.020 - Testing Accuracy: 41.670 - Learning Rate: 0.00835
Epoch 210 - Train Loss 0.7594 - Training Accuracy: 43.908 - Testing Accuracy: 42.550 - Learning Rate: 0.00810
Epoch 240 - Train Loss 0.7570 - Training Accuracy: 44.278 - Testing Accuracy: 42.690 - Learning Rate: 0.00787
Epoch 270 - Train Loss 0.7491 - Training Accuracy: 45.236 - Testing Accuracy: 43.230 - Learning Rate: 0.00763
```

Figure 9 Processing

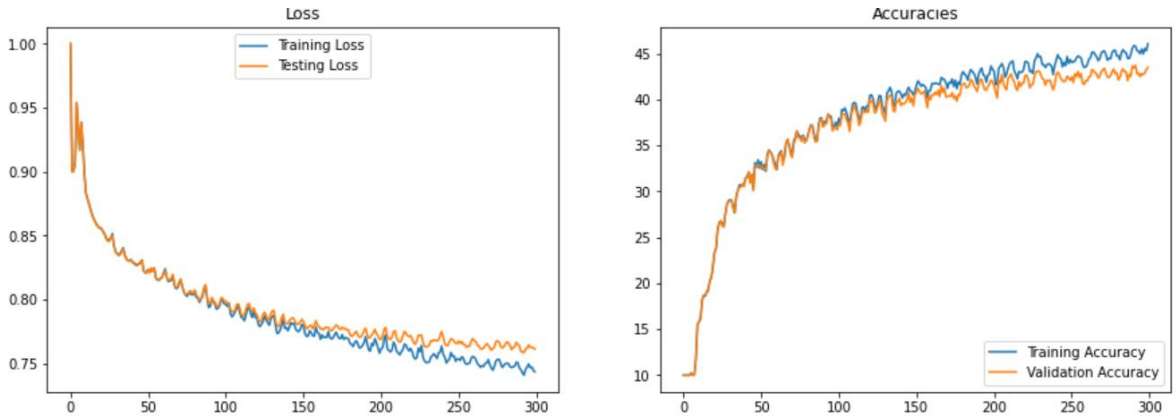

Figure 10 Losses and Accuracies of item 2

3) The learning rate was changed to 0.0001 to avoid the overshooting problem.

```
t0 = time.time()
for t in range(iterations):
    #indices= np.random.choice(Ntr,Ntr)
    indices= np.array(range(0,Ntr))
    x=x_train[indices]
    y=y_train[indices]
    for s in range(steps):
      x_mini=x[batch_size*s:batch_size*(s+1)]
      y_mini=y[batch_size*s:batch_size*(s+1)]
      h=1.0/(1.0+np.exp(-(x_mini.dot(w1)+b1)))
```

Figure 12 Batch defining

```
batch_size = 500
steps = int(Ntr / batch_size)
iterations = 300
lr=0.0001
lr_decay = 0.999
reg = 5e-6
```

Figure 11 Parameters of item 3

The output of the code

```
Epoch 000 - Train Loss 0.9132 - Training Accuracy: 10.000 - Testing Accuracy: 10.000 - Learning Rate: 0.00010
Epoch 030 - Train Loss 0.8358 - Training Accuracy: 30.372 - Testing Accuracy: 30.460 - Learning Rate: 0.00010
Epoch 060 - Train Loss 0.8036 - Training Accuracy: 37.288 - Testing Accuracy: 37.180 - Learning Rate: 0.00009
Epoch 090 - Train Loss 0.7789 - Training Accuracy: 40.718 - Testing Accuracy: 40.020 - Learning Rate: 0.00009
Epoch 120 - Train Loss 0.7647 - Training Accuracy: 42.798 - Testing Accuracy: 42.040 - Learning Rate: 0.00009
Epoch 150 - Train Loss 0.7546 - Training Accuracy: 44.306 - Testing Accuracy: 42.900 - Learning Rate: 0.00009
Epoch 180 - Train Loss 0.7455 - Training Accuracy: 45.638 - Testing Accuracy: 43.680 - Learning Rate: 0.00008
Epoch 210 - Train Loss 0.7370 - Training Accuracy: 46.748 - Testing Accuracy: 44.250 - Learning Rate: 0.00008
Epoch 240 - Train Loss 0.7294 - Training Accuracy: 47.756 - Testing Accuracy: 44.890 - Learning Rate: 0.00008
Epoch 270 - Train Loss 0.7223 - Training Accuracy: 48.656 - Testing Accuracy: 45.370 - Learning Rate: 0.00008
Epoch 299 - Train Loss 0.7159 - Training Accuracy: 49.506 - Testing Accuracy: 45.690 - Learning Rate: 0.00007
End of Training. time taken: 2982.78 seconds
```

Figure 14 Processing

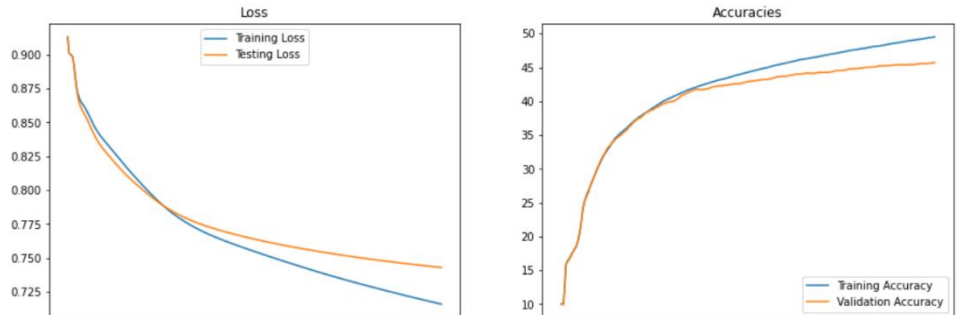

Figure 13 Losses and Accuracies of item 3

Processing time is almost same as the processing time of item 2. By comparing the result with item 2 we can see that the loss received at after a certain amount of epochs will be reached earlier in item 3. Which means that the converging is reached early when stochastic gradient descent (sgd) is implemented. To explain more, by considering only the first epoch by the time parameters of gradient descent gets updated once, the parameters of the sgd gets updated 100 times (considering batch size 500 and training samples 50000). Because of this, the loss we get with gradient descent after a certain number of epochs will be achieved quicker with sgd. Further, the accuracy obtained with sgd seems to be a little bit higher than the accuracy obtained by normal gradient descent.

4)

Activations used for C32, C64, C64, F64 was ReLU and for F10 softmax was used for the better performance of the code.

Parameters – Learning Rate – 0.001

Momentum – 0.1

When learning rate was set to a higher value, the output overshoots and didn't provide preferable result.

Learnable Parameters – 122,570 (provided in the image)

```
train_images.shape:  (50000, 32, 32, 3)
train_labels.shape:  (50000, 1)
test_images.shape: (10000, 32, 32, 3)
test_labels.shape: (10000, 1)
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 64)        18496
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 4, 4, 64)          0
_________________________________________________________________
flatten_3 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                65600
_________________________________________________________________
dense_7 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
```

Figure 15 Parameters of the model

Since this Convolutional Neural Network (CNN) is large, and the number of training samples is small considered to its capacity, I have reduced the epochs to 100 here to save some time.

```
training_history = model.fit(train_images, train_labels,epochs=100, batch_size=50, validation_data=(test_images,  test_labels), verbose = 2)
```

Figure 16 Processing code

Output

```
Epoch 1/100
1000/1000 - 4s - loss: 2.1127 - accuracy: 0.2436 - val_loss: 1.9864 - val_accuracy: 0.3065
Epoch 2/100
1000/1000 - 3s - loss: 1.9101 - accuracy: 0.3295 - val_loss: 1.8463 - val_accuracy: 0.3517
Epoch 3/100
1000/1000 - 3s - loss: 1.7834 - accuracy: 0.3762 - val_loss: 1.7253 - val_accuracy: 0.3958
- - ----
1000/1000 - 3s - loss: 0.7814 - accuracy: 0.7310 - val_loss: 0.9398 - val_accuracy: 0.6785
Epoch 98/100
1000/1000 - 3s - loss: 0.7793 - accuracy: 0.7324 - val_loss: 0.9469 - val_accuracy: 0.6798
Epoch 99/100
1000/1000 - 3s - loss: 0.7746 - accuracy: 0.7343 - val_loss: 0.9723 - val_accuracy: 0.6679
Epoch 100/100
1000/1000 - 3s - loss: 0.7711 - accuracy: 0.7344 - val_loss: 0.9365 - val_accuracy: 0.6803
```

Figure 17 First and the last part of the process

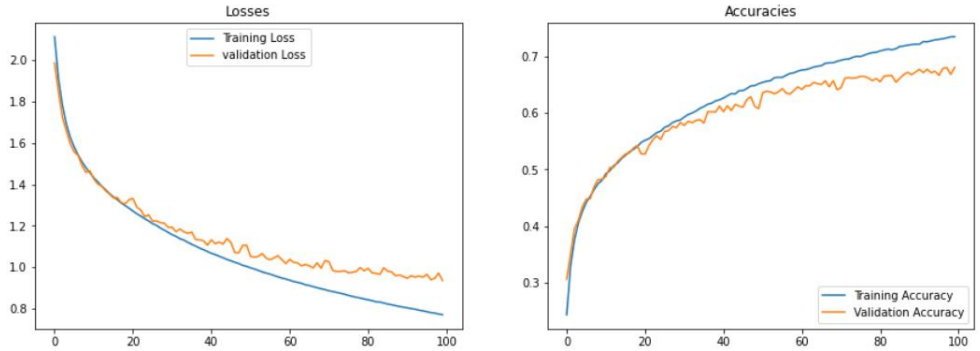

Figure 18 Losses and Accuracies of Item 4

Compared to the previous items on the same data set, the CNN produces a higher accuracy. To thoroughly analyze this CNN a larger data set would be helpful.

Some learnings from this assignment

- The learning rate (Lr) parameter should be carefully set for proper execution of the code. Normally it takes a value within the range 0.0 - 1.0. If the value is higher, then the model weights will be learnt faster and training will be executed fast. If the value is small, then it will take more time to learn the weights and the execution will be slower.
- But if the learning rate is set to a very high value then it will overshoot or the system will be unstable and it won't converge to a result. Similarly if the value is set to a very small value then the system could get stuck.
- For example, considering all other parameters are same, if the value is around 0.1, system may be unstable. If the value is around 0.01 then there might be some spikes. And at 0.001 the spikes will reduce. And at 0.0001 a proper curved line without spikes may be seen. At 1e-6 or lower system might be unable to learn properly.
- Note that we can also change the Lr delay parameter to avoid system being unstable.
- Also a case where the test loss graph takes the pattern of a tick was observed for certain Lrs. (i.e. both the test loss and test accuracy is increasing) In this case, the model has been overfit and it has started to predict only images in training and reduced generalizing.
- Sgd is more useful than gd when images with large size is used.