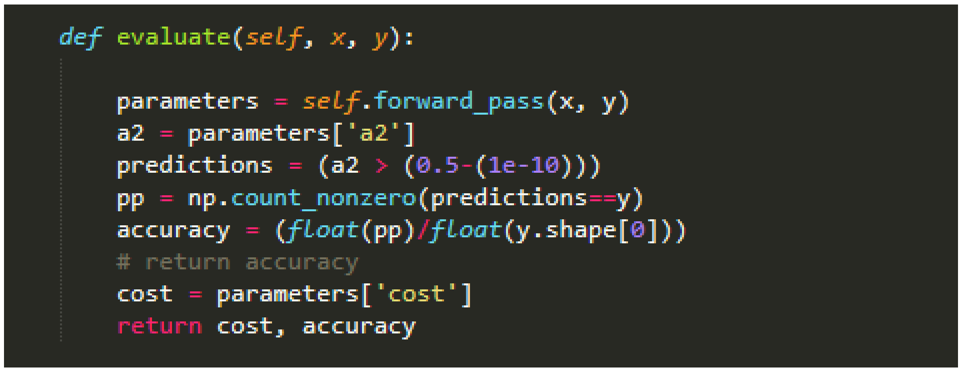
Assignment 2 Report

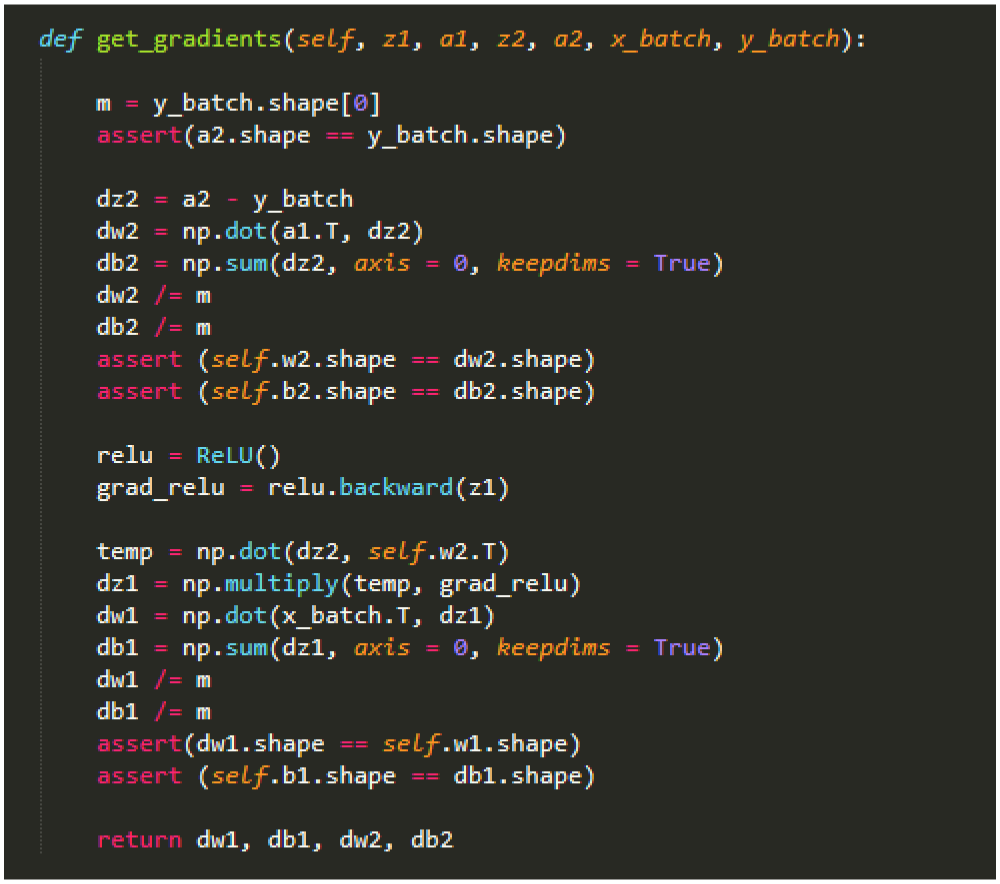
*1) Write a function that evaluates the trained network (5 points), as well as computes all the sub gradients of **W1** and **W2** using backpropagation (5 points).*



```
def evaluate(self, x, y):

    parameters = self.forward_pass(x, y)
    a2 = parameters['a2']
    predictions = (a2 > (0.5-(1e-10)))
    pp = np.count_nonzero(predictions==y)
    accuracy = (float(pp)/float(y.shape[0]))
    # return accuracy
    cost = parameters['cost']
    return cost, accuracy
```

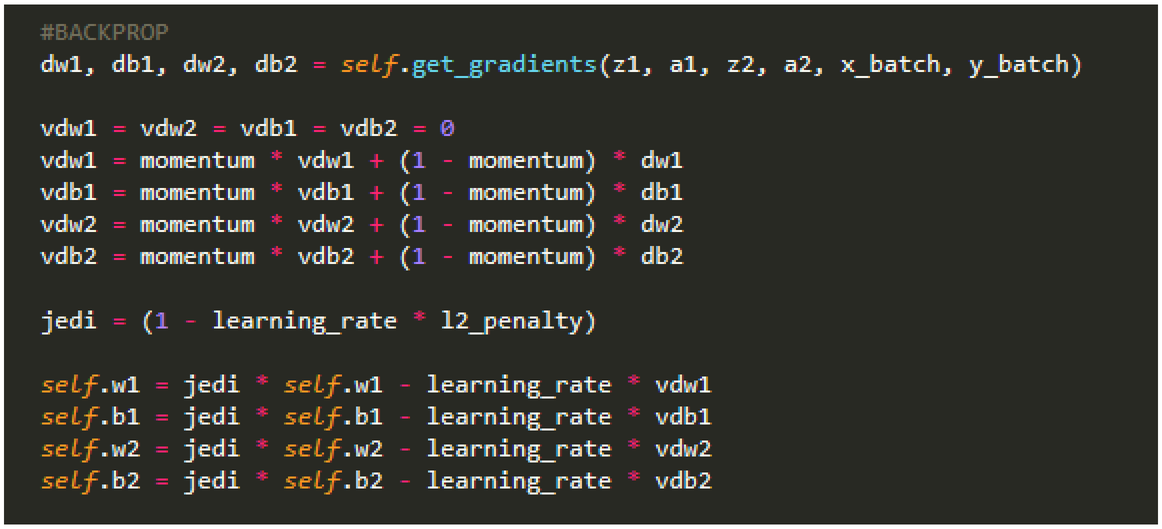
Figure: 1

This above piece of code evaluates the trained network give the input and corresponding output. It runs a forward pass through the neural network and matches the resulting output with Y. As the neural network provides the output in form of the result of sigmoid function the results are in probability between 0 and 1 which are then converted into binary results. If the probability is less than 0.5 we are considering it as a negative else positive.



```
def get_gradients(self, z1, a1, z2, a2, x_batch, y_batch):

    m = y_batch.shape[0]
    assert(a2.shape == y_batch.shape)

    dz2 = a2 - y_batch
    dw2 = np.dot(a1.T, dz2)
    db2 = np.sum(dz2, axis = 0, keepdims = True)
    dw2 /= m
    db2 /= m
    assert (self.w2.shape == dw2.shape)
    assert (self.b2.shape == db2.shape)

    relu = ReLU()
    grad_relu = relu.backward(z1)

    temp = np.dot(dz2, self.w2.T)
    dz1 = np.multiply(temp, grad_relu)
    dw1 = np.dot(x_batch.T, dz1)
    db1 = np.sum(dz1, axis = 0, keepdims = True)
    dw1 /= m
    db1 /= m
    assert(dw1.shape == self.w1.shape)
    assert (self.b1.shape == db1.shape)

    return dw1, db1, dw2, db2
```

This function is used to calculate the gradient of the parameters of the neural network. We have parameters w1, w2, b1 and b2. In the backpropagation step we need perform modification on our parameters. The gradient dw1, dw2, db1, db2 was calculated and used in the following way in the training method to modify the parameters.

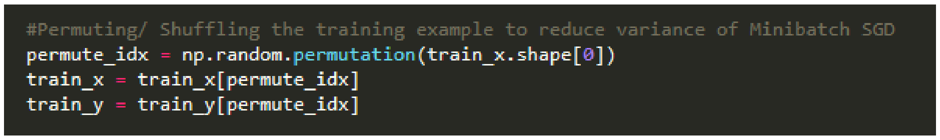


```
#BACKPROP
dw1, db1, dw2, db2 = self.get_gradients(z1, a1, z2, a2, x_batch, y_batch)

vdw1 = vdw2 = vdb1 = vdb2 = 0
vdw1 = momentum * vdw1 + (1 - momentum) * dw1
vdb1 = momentum * vdb1 + (1 - momentum) * db1
vdw2 = momentum * vdw2 + (1 - momentum) * dw2
vdb2 = momentum * vdb2 + (1 - momentum) * db2

jedi = (1 - learning_rate * l2_penalty)

self.w1 = jedi * self.w1 - learning_rate * vdw1
self.b1 = jedi * self.b1 - learning_rate * vdb1
self.w2 = jedi * self.w2 - learning_rate * vdw2
self.b2 = jedi * self.b2 - learning_rate * vdb2
```

Figure: 3

2) Write a function that performs stochastic mini-batch gradient descent training (5 points). You may use the deterministic approach of permuting the sequence of the data. Use the momentum approach described in the course slides.

Stochastic mini-batch gradient descent training was implemented which is shown in figure 3. To reduce variance in the minibatch SGD the training data was permuted before training in each epoch using the following piece of code:

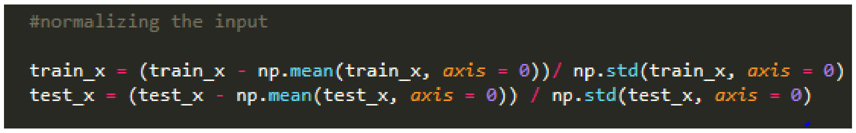


```
#Permuting/ Shuffling the training example to reduce variance of Minibatch SGD
permute_idx = np.random.permutation(train_x.shape[0])
train_x = train_x[permute_idx]
train_y = train_y[permute_idx]
```

Figure: 4

3) Train the network on the attached 2-class dataset extracted from CIFAR-10: (data can be found in the cifar-2class-py2.zip file on Canvas.). The data has 10,000 training examples in 3072 dimensions and 2,000 testing examples. For this assignment, just treat each dimension as uncorrelated to each other. Train on all the training examples, tune your parameters (number of hidden units, learning rate, mini-batch size, momentum) until you reach a good performance on the testing set. What accuracy can you achieve? (20 points based on the report).

The training examples were normalized 1st using the Z-score normalization using the following piece of code:



```
#normalizing the input

train_x = (train_x - np.mean(train_x, axis = 0))/ np.std(train_x, axis = 0)
test_x = (test_x - np.mean(test_x, axis = 0)) / np.std(test_x, axis = 0)
```

Figure: 5

Without normalizing the training and test data the sigmoid and cross-entropy loss function was exploding. Also, the parameters W1, W2 was declared in a way so that they don't explode the sigmoid activation and cross entropy loss function.

Without doing any sort of tuning (not using any momentum, l2 regularization) using the vanilla NN at the beginning of code testing I was able to reach around 82% of test accuracy and over 90% of training accuracy. After testing that my vanilla NN was performing in a stable manner I implemented momentum and l2 regularization (Figure 3 and 8).

At first, I was using 0.1 as my learning rate and my NN was not converging and when I increased the epoch significantly I saw some convergence that reflected that my learning rate wasn't set properly so it was dangling around the critical point. Then I saw significant improvement when I set my learning rate to 0.02. Later I've tested my system with different learning rates which is shown in figure 8. And the best test accuracy was 84% using 5e-3 using learning rate.

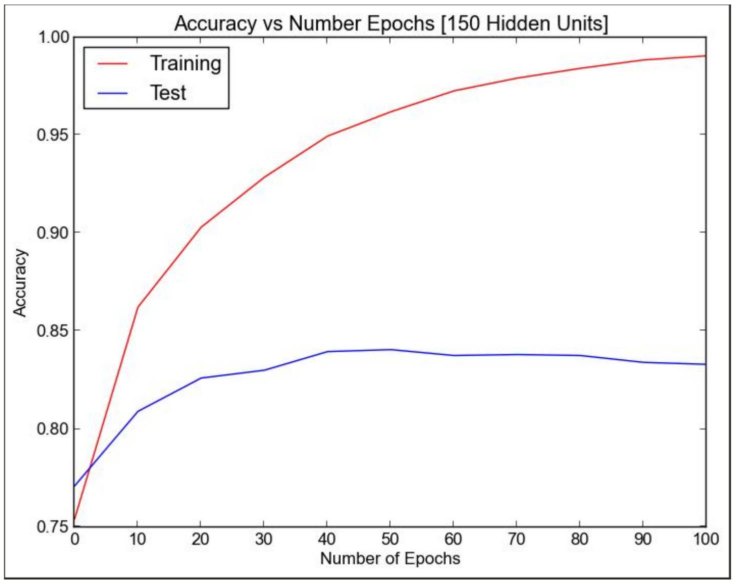


Figure: 6

Setting the number of epochs was also part of my experiment. Figure 6 relates to my finding that after 40 epochs test accuracy reached 83.95% and it didn't change that much later. Best test accuracy was found while number of epochs was 50 and the accuracy was 84.05%. Although with the increment of number of epochs training accuracy is increasing gradually and after 100 epochs it reached 99.05% accuracy which overfits the training data but doesn't have any impact on the test accuracy.

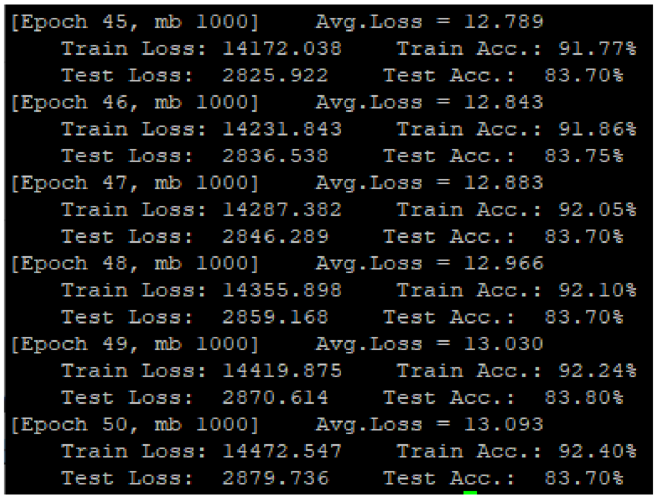
I've also tested my system with different number of hidden units which is replicated in Figure 13. I started my testing with 100 hidden units and later tested on different hidden units. I saw that using more than 100 hidden units showed significant improvement than using less than 50 hidden units. Though from

Figure 13 it appears that best accuracy was achieved using 150 hidden units but there wasn't major improvement among number of hidden units among 100 and 150.

For the whole experiment I set my batch size to 1000 which mean each minibatch has 10 tuples. After fixing the best learning rate to be 5e-3 and hidden unit to 150 I tested my system with different batch size which is shown in figure 10. It seems that for minibatch size of 10 best accuracy was found. Using minibatch greater less than 10 there was performance drop.

Using momentum 0.9 after 50 epochs maximum 83.80% test accuracy was achieved. And using momentum 0.8 after 50 epochs maximum of accuracy of 83.25% was reached. (Figure 7 and 8). Also, in Figure 7 I used 0.001 as l2 penalty (92.4% Training accuracy after 50 epochs) and in Figure 8 I used 0.000001 as l2 penalty (97.06% Training Accuracy after 50 epochs). We can see from the figures that when l2 penalty is large there is less overfitting in the training data. Using momentum of 0.85 and l2 penalty as 0.000001 maximum 83% test accuracy was achieved. <u>More details about achieved accuracy is written in question 6.</u>

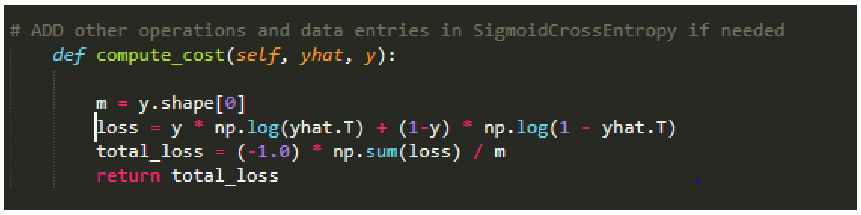
(4) Training Monitoring: For each epoch in training, your function should evaluate the training objective, testing objective, training misclassification error rate (error is 1 for each example if misclassifies, 0 if correct), testing misclassification error rate (5 points).



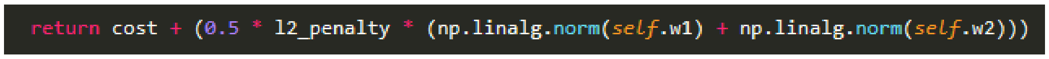
```
[Epoch 45, mb 1000]     Avg.Loss = 12.789
    Train Loss: 14172.038    Train Acc.: 91.77%
    Test Loss:  2825.922     Test Acc.:  83.70%
[Epoch 46, mb 1000]     Avg.Loss = 12.843
    Train Loss: 14231.843    Train Acc.: 91.86%
    Test Loss:  2836.538     Test Acc.:  83.75%
[Epoch 47, mb 1000]     Avg.Loss = 12.883
    Train Loss: 14287.382    Train Acc.: 92.05%
    Test Loss:  2846.289     Test Acc.:  83.70%
[Epoch 48, mb 1000]     Avg.Loss = 12.966
    Train Loss: 14355.898    Train Acc.: 92.10%
    Test Loss:  2859.168     Test Acc.:  83.70%
[Epoch 49, mb 1000]     Avg.Loss = 13.030
    Train Loss: 14419.875    Train Acc.: 92.24%
    Test Loss:  2870.614     Test Acc.:  83.80%
[Epoch 50, mb 1000]     Avg.Loss = 13.093
    Train Loss: 14472.547    Train Acc.: 92.40%
    Test Loss:  2879.736     Test Acc.:  83.70%
```

Figure: 7 [Momentum 0.9, Hidden Units 150, Epochs 50]

For each epoch of training my function evaluates the training and testing objecting and represents the training misclassification error rate. Following is the cross-entropy loss function that returns the loss:



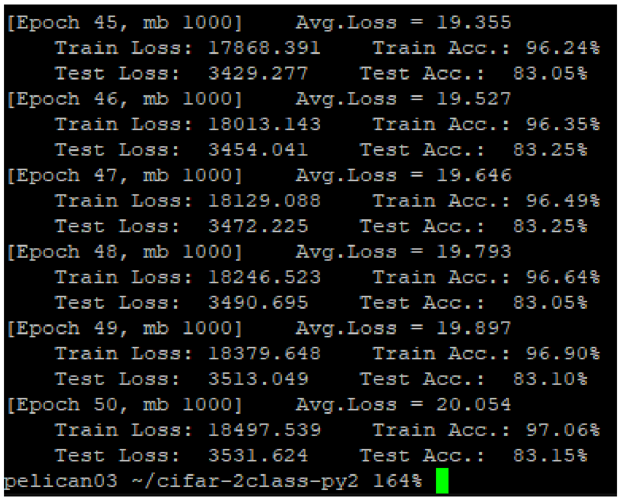
```
# ADD other operations and data entries in SigmoidCrossEntropy if needed
    def compute_cost(self, yhat, y):

        m = y.shape[0]
        loss = y * np.log(yhat.T) + (1-y) * np.log(1 - yhat.T)
        total_loss = (-1.0) * np.sum(loss) / m
        return total_loss
```



```
    return cost + (0.5 * l2_penalty * (np.linalg.norm(self.w1) + np.linalg.norm(self.w2)))
```

Figure: 8

Also, regularization was added to the cost returned by the cost function. To make sure that our loss function doesn't generate divide by zero error for the log function which appeared to occur often, I added a small error(1e-10) inside the sigmoid function so that the sigmoid function doesn't return 0.



```
[Epoch 45, mb 1000]     Avg.Loss = 19.355
    Train Loss: 17868.391    Train Acc.: 96.24%
    Test Loss:  3429.277    Test Acc.:  83.05%
[Epoch 46, mb 1000]     Avg.Loss = 19.527
    Train Loss: 18013.143    Train Acc.: 96.35%
    Test Loss:  3454.041    Test Acc.:  83.25%
[Epoch 47, mb 1000]     Avg.Loss = 19.646
    Train Loss: 18129.088    Train Acc.: 96.49%
    Test Loss:  3472.225    Test Acc.:  83.25%
[Epoch 48, mb 1000]     Avg.Loss = 19.793
    Train Loss: 18246.523    Train Acc.: 96.64%
    Test Loss:  3490.695    Test Acc.:  83.05%
[Epoch 49, mb 1000]     Avg.Loss = 19.897
    Train Loss: 18379.648    Train Acc.: 96.90%
    Test Loss:  3513.049    Test Acc.:  83.10%
[Epoch 50, mb 1000]     Avg.Loss = 20.054
    Train Loss: 18497.539    Train Acc.: 97.06%
    Test Loss:  3531.624    Test Acc.:  83.15%
pelican03 ~/cifar-2class-py2 164%
```

Figure: 9 [Momentum 0.8, Hidden Units 150, Epochs 50]

While using momentum 0.8 accuracy of maximum accuracy achieved within 50 epochs was 83.25%.

In Figure 1 the code of evaluating the network is given along with description in question 1.

(5) Tuning Parameters: please create three figures with following requirements. Save them into jpg format:

i) test accuracy with different number of batch size

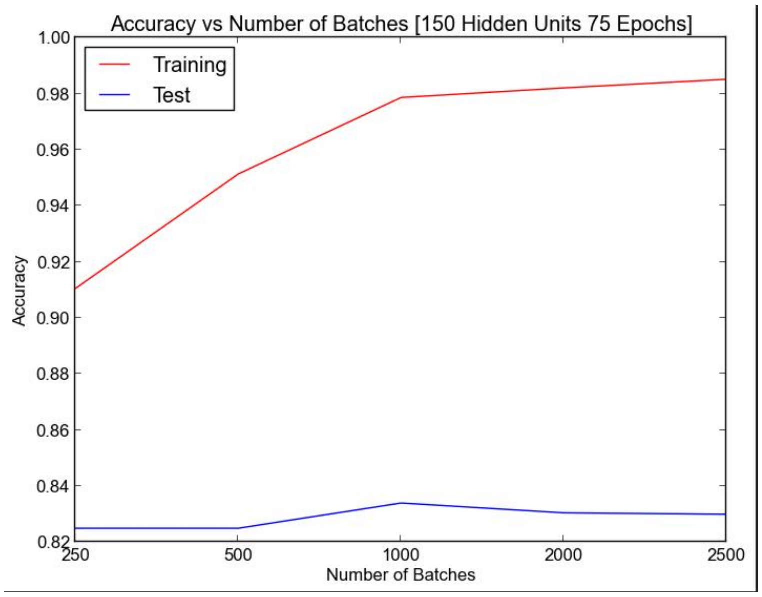


Figure: 10

The above plot shows the test and training accuracy for different number of batches. Using the hyper-parameters: learning rate = 5e-3, momentum = 0.8 in the experiment the best result was found while I set the number of batches to 1000. That means each minibatch size was set to 10. We can see that when batch size is below 1000, accuracy is relatively lower than when batch size is greater than 1000.

ii)test accuracy with different learning rate

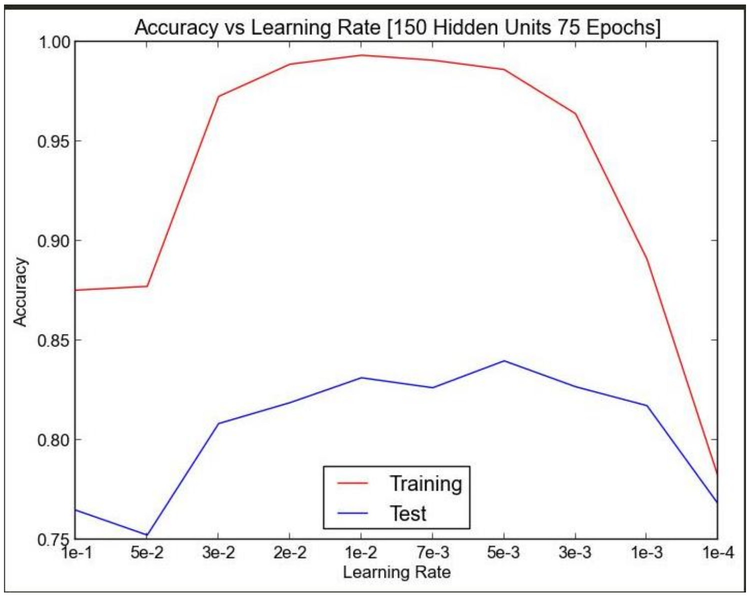


Figure: 11

Above plot shows test and training accuracies for different learning rate. This plot was generated keeping hyperparameter: Number of hidden units and epochs fixed to 150 and 75 respectively. Maximum accuracy 84% was achieved by setting learning rate to 5e-3. And from the plot we can also see that when the learning rate becomes very small 1e-4 it is not it takes much longer time to reach a higher accuracy.

iii) test accuracy with different number of hidden units

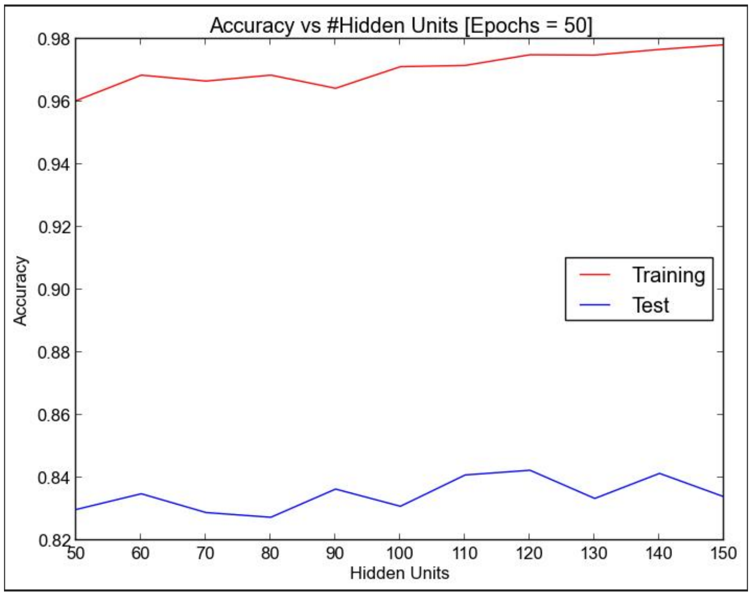


Figure: 12

The above plot displays test and training accuracy for different number of hidden units in our 1st hidden layer. We can see that as the number of hidden units is increasing the training data is overfitting a lot. For 150 hidden units the training accuracy reaches 97.83% and we get the best result for testing accuracy for 120 hidden units which is 84.25%. The batch size was fixed to 1000 and learning rate was 1e-2.

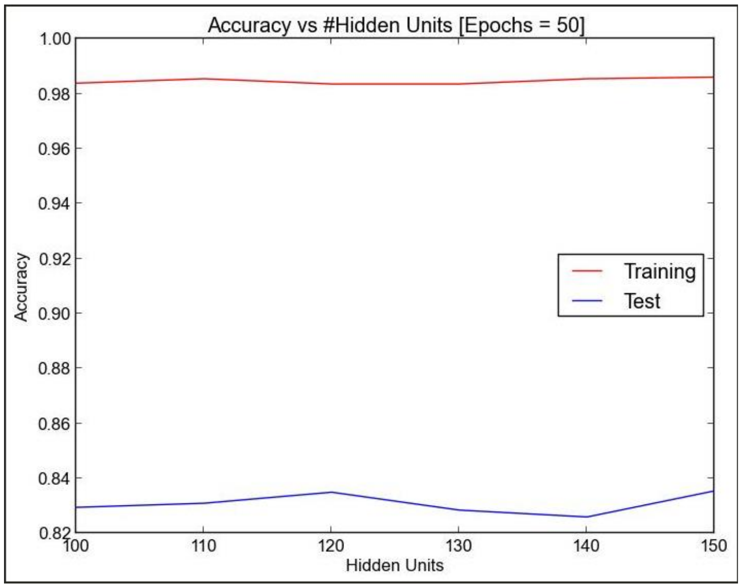


Figure: 13

The above plot also displays the test and training accuracy for different number of hidden units with the same hyper-parameters, but permutation was added to the code. Training data was permuted in each epoch for reducing variance in minibatch stochastic gradient descent.

(6) Discussion about the performance of your neural network.

It was easy to achieve around 80% of testing accuracy without any sort of training using the neural network I implemented. Then it required some tuning to reach above 84%. Most of the time it was around 83%-84%. Maximum of 84.25% training accuracy was achieved using my neural network. But increasing the number of epochs didn't help that much. It was easy to reach above 80% testing accuracy just after 6-7 epochs reaching more than 84% accuracy took a long time most of the case and completely depended on the initial values of hyperparameters. For example, using momentum 0.75, Hidden Units = 150 and l2 penalty as 0.0001 after 6 iterations testing accuracy reached 80%, after 30 epochs accuracy reached 82% and even after 70 epochs accuracy didn't reach 83%. Whereas in the above discussion it has already been mentioned it 83% testing accuracy was clearly reachable. So, the performance of the system was quite dependent on the hyperparameters. And as there are multiple hyperparameters there can be various combinations of them that can be used on our system. But due to high running time it wasn't possible to try all possible combinations, but it seemed that system wasn't able to cross 85% accuracy.