Data Mining Assignment 3

Zhengyuan Zhu
CSE5334 - Data Mining
University of Texas at Arlington

May 3, 2020

Problem 1: Logistic Regression

Solution for 1.1

For implement logistic regression, we need to use 'numpy' to make up a toy dataset which generates 1000 training instances in two sets of random data points(500 in each) from multivariate normal distribution with

$$\mu_1 = [1, 0], \mu_2 = [0, 1.5], \Sigma_1 = \begin{bmatrix} 1 & 0.75 \\ 0.75 & 1 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 1 & 0.75 \\ 0.75 & 1 \end{bmatrix}$$

and label them with 0 and 1. For the convenience of later steps, we also need to stack two distributions data vertically.

```
# build train set
mul1, sigma1 = [1, 0], [[1, 0.75], [0.75, 1]]
mul2, sigma2 = [0, 1], [[1, 0.75], [0.75, 1]]
train_size = 500
test_size = 250

train1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=
                                       train_size)
train1_label = np.zeros((train_size, 1))

train2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=
                                       train_size)
train2_label = np.ones((train_size, 1))
X_train = np.vstack([train1, train2])
y_train = np.vstack([train1_label, train2_label])
print("Train set samples: \n",X_train[:5], X_train[-5:])
print("Train set labels: \n", y_train[:5], y_train[-5:])

test1 = np.random.multivariate_normal(mean=mul1, cov=sigma1, size=
                                       test_size)
test1_label = np.zeros((test_size, 1))
test2 = np.random.multivariate_normal(mean=mul2, cov=sigma2, size=
                                       test_size)
test2_label = np.ones((test_size, 1))
X_test = np.vstack([test1, test2])
y_test = np.vstack([test1_label, test2_label])
```

Then, we implement the logistic regression through **pytorch**. The implementation of model is fairly simple that only contains one fully connected layer and takes two number as input and output a single number. Then using 'Sigmoid' as activation function and binary cross entropy as loss function.
We also need to define a class for loading the data. The codes are shown below:

```
class LogisticReg(torch.nn.Module):
    def __init__(self):
        super(LogisticReg, self).__init__()
        self.fc = torch.nn.Linear(2, 1)
    def forward(self, x):
        x = self.fc(x)
        return F.sigmoid(x)

class NormDataset(torch.utils.data.Dataset):
    def __init__(self, x, y):
        self.len = x.shape[0]
        self.device = 'cuda'if torch.cuda.is_available() else 'cpu'
        self.x_data = torch.as_tensor(x, device=self.device, dtype=torch.
                                                    float)
        self.y_data = torch.as_tensor(y, device=self.device, dtype=torch.
                                                    float)
    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]
    def __len__(self):
        return self.len
loss_func = torch.nn.BCELoss()
```

During the training process, firstly we create a object for normal distribution dataset. And use pytorch dataloader to transform numpy object of train set and test set to torch tensor object. Then use a for loop to iterate 4 different learning rate $\eta = \{1, 0.1, 0.01, 0.001\}$.
For the batch training part, we use stochastic gradient descent(SGD) as optimizer and calculate L1 normalization of gradients as a threshold to early stop the training process. If L1 normalization of model's weight and bias has not change in 10 epochs, then the train process will stop. The codes and experiment results are shown below:

```
train_set, test_set = NormDataset(X_train, y_train), NormDataset(X_test,
                                        y_test)
train_loader = torch.utils.data.DataLoader(dataset=train_set, batch_size=
                                        32, shuffle=False)
X_test_tsr, y_test_tsr = Variable(torch.from_numpy(X_test).float(),
                                        requires_grad=False), Variable(torch.
                                        from_numpy(y_test).float(),
                                        requires_grad=False)

writer = SummaryWriter()
for lr in learning_rates:
    model = LogisticReg()
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    prev_norm, norms, cnt = torch.tensor(0), torch.tensor(0), 0
    print("Parameters before training:")
    for name, param in model.named_parameters():
        if param.requires_grad:
```

Learning rate	1	0.1	0.01	0.001
Accuracy	0.864	0.914	0.912	0.912
Gradient Norm	0.0505	0.1222	0.1340	0.1354
Loss	0.0355	0.0923	0.1027	0.1039
Iterations	51	344	2440	151771

Table 1: Experiment results of batch training

```
            print(name, param.data)
    for epoch in tqdm(range(num_epochs)):
        early_stop = False
        for i, data in enumerate(train_loader):
            X_train_tsr, y_train_tsr = data
            y_pred = model(X_train_tsr)
            loss = loss_func(y_pred, y_train_tsr)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        norms = torch.norm(model.fc.weight.grad)+torch.norm(model.fc.bias.
                                                    grad)
        if prev_norm.data==norms.data and cnt<10:
            cnt += 1
        if cnt==10:
            print('Early stopping at {} epoch when norms={}'.format(epoch,
                                                        norms.data))
            break
        writer.add_scalar('Loss/lr='+str(lr), loss, epoch)
        writer.add_scalar('GradNorm/lr='+str(lr), norms, epoch)
        prev_norm = norms
    test_pred = model.forward(X_test_tsr).data.numpy()
    test_pred = np.where(test_pred>0.5, 1., 0.)
    acc = accuracy_score(test_pred, y_test_tsr.data.numpy())
    print("\nParameters after training:")
    for name, param in model.named_parameters():
        if param.requires_grad:
            print(name, param.data)
    print('\nWhen lr={}, the accuracy is {}'.format(lr, acc))
    print('------'*10)
```

According to Table 1, it shows that the model achieves the best performance(accuracy: 91.4%) when we set learning rate to 0.1. And the figures of changes of norm of gradient and losses are shown in the Figure 1. It will take more iterations to converge when set a small learning rate.

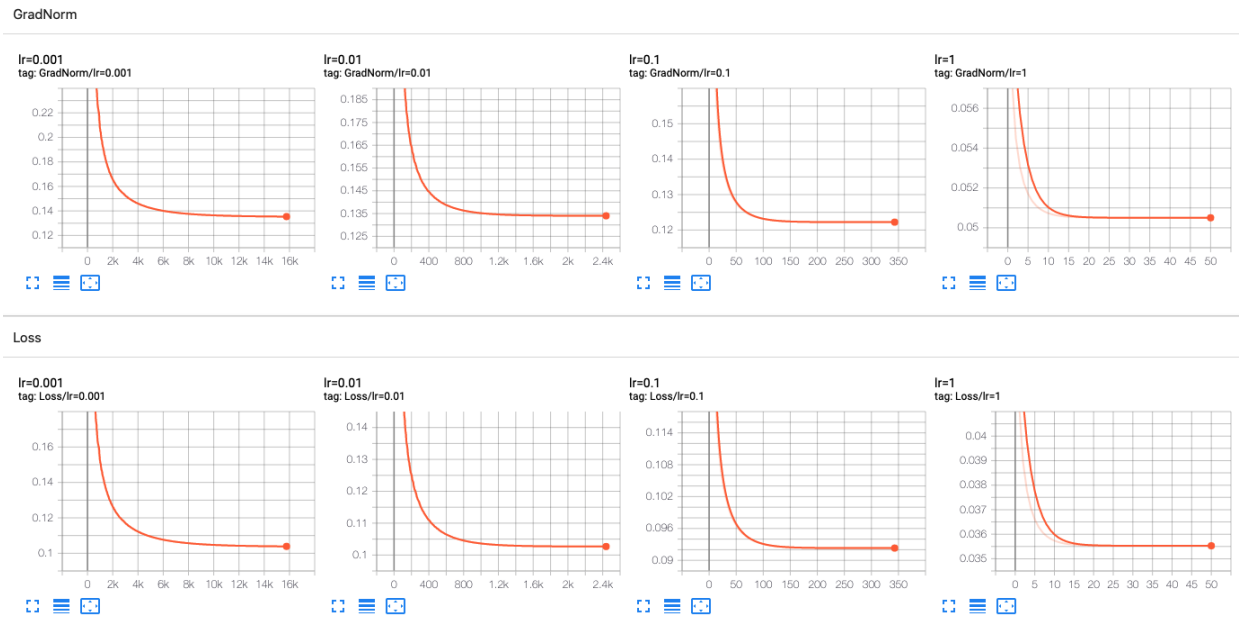

Figure 1: Figures show L1 normalization of gradients and losses w.r.t iteration.

We also visualise the scatter plot of the testing data and the trained decision boundary in Figure 2

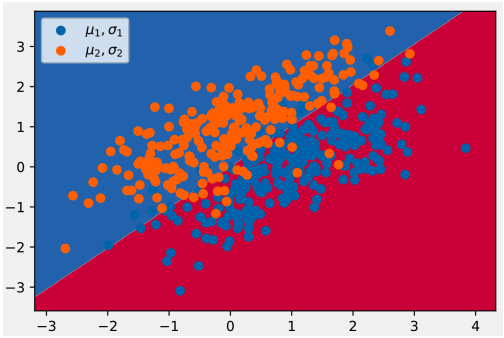

Figure 2: Scatter plot of the testing data and the trained decision boundary of batch training.

Then we do some visualizations of the probability density function. The figure below shows that the different densities with different bandwidth. The red line represents the ground truth distribution and the histogram represents the samples distribution. It shows that the distribution curve becomes more flat when bandwidths get larger and larger.

Solution for 1.2

For the problem 1.2, the methodology essentially is the same as problem 1.1. The only different is that we set batch size from 32 to 1 to make it a online training. The printed log records are shown below:

Learning rate	1	0.1	0.01	0.001
Accuracy	0.552	0.77	0.922	0.911
Gradient Norm	0.0003	0.0748	0.4951	0.6689
Loss	0.0001	0.0446	0.3404	0.4939
Iterations	15	27	103	726

Table 2: Experiment results of online training

```
Parameters before training:
fc.weight tensor([[-0.7055,  0.0255]])
fc.bias tensor([-0.2273])
Early stopping at 15 epoch when norms=0.003028314560651779
Parameters after training:
fc.weight tensor([[-2.8413,  3.3302]])
fc.bias tensor([5.6395])
When lr=1, the accuracy is 0.552
------------------------------------------------------------
Parameters before training:
fc.weight tensor([[-0.0246, -0.4560]])
fc.bias tensor([-0.1986])
Early stopping at 27 epoch when norms=0.0748012363910675
Parameters after training:
fc.weight tensor([[-2.6917,  3.3682]])
fc.bias tensor([2.3081])
When lr=0.1, the accuracy is 0.77
------------------------------------------------------------
Parameters before training:
fc.weight tensor([[ 0.2296, -0.2112]])
fc.bias tensor([-0.6161])
Early stopping at 103 epoch when norms=0.49514836072921753
Parameters after training:
fc.weight tensor([[-3.1573,  3.4086]])
fc.bias tensor([0.2957])
When lr=0.01, the accuracy is 0.922
------------------------------------------------------------
Parameters before training:
fc.weight tensor([[0.4855, 0.6238]])
fc.bias tensor([0.4780])
Early stopping at 726 epoch when norms=0.6689199209213257
Parameters after training:
fc.weight tensor([[-3.3658,  3.3036]])
fc.bias tensor([-0.0130])
When lr=0.001, the accuracy is 0.91
------------------------------------------------------------
```

The experiment results in Table 2 shows the same conclusion that smaller learning rate will lead to more iterations to converge. And the best performance(Accuracy 92.2%) appears when setting the learning rate to 0.01 during online training. We should notice that although the loss of $learningrate = 1$ is pretty low(0.0001), the accuracy is the worst among all the experiments.

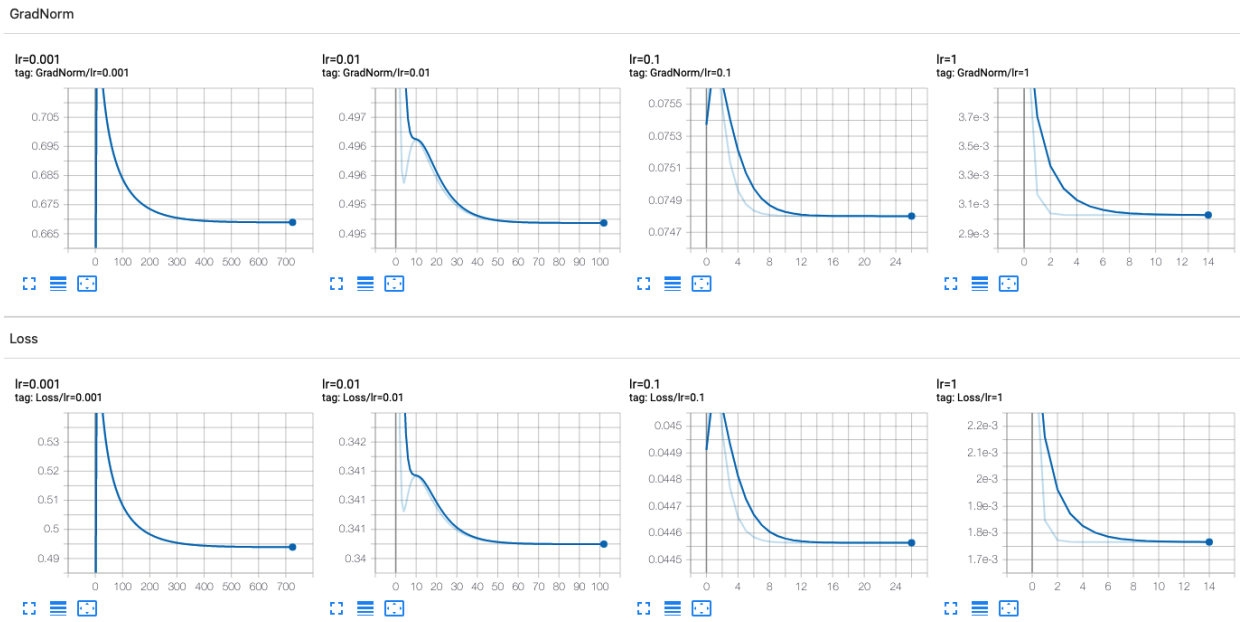

Figure 3: Figures show L1 normalization of gradients and losses w.r.t iteration.

We also visualise the scatter plot of the testing data and the trained decision boundary in Figure 3

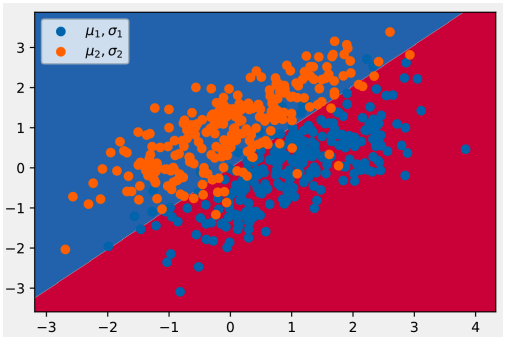

Figure 4: Scatter plot of the testing data and the trained decision boundary of online training.

Problem 2 Artificial Neural Network

Solution 2.1

Use *PyTorch* to design a Neural Network for Image Classification on $CIFAR-10$ *dataset*. First, we need to convert color images to gray scale and make sure that the pixel intensities are normalized to stay in [0, 1]. It is very convenient to implement transform converter via *torch.vision*, but do not forget there are three channels(RGB) in $CIFAR-10$ *datasets* and

only one channel in $fashion\ MNIST$. As a result, we will use two different transformers that take images as input and transform them into tensor then normalize those tensor:

```
cifar10_transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
    )
fashion_transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize([0.5], [0.5])])
```

We modify the sample codes to complete the first problem. For initiating the Net class of problem 2.1, the first fully connected layer should change is input size from $image\_cifar10\_width*image\_cifar10\_height$ to $image\_cifar10\_width*image\_cifar10\_height*color\_channels$. We don't need to modify too much sample codes to finish the first task. The result are shown in Figure 5 and Figure 6.

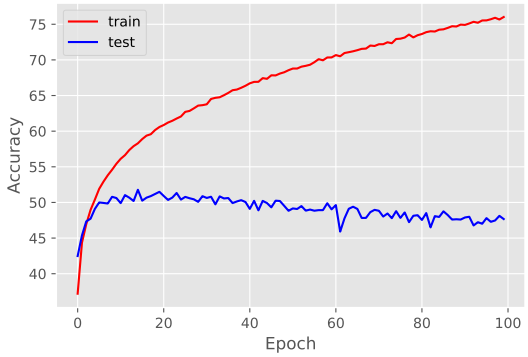

Figure 5: Accuracy of vanilla neural network

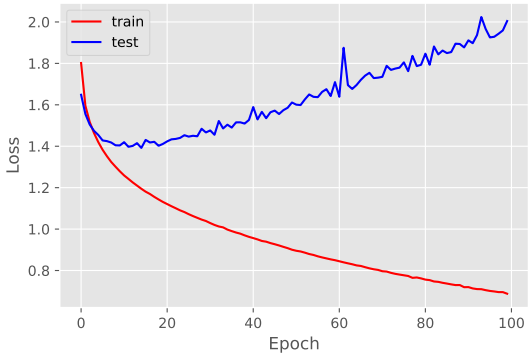

Figure 6: Loss of vanilla neural network

Solution 2.2

We can simply add a hidden layer through 'torch.nn.Linear()' function to '__init__' in the Net class. The results in Figure 7 and Figure 8 show that the one more hidden layer model outperforms the vanilla one. The shape of figures pretty much look the same, and the accuracy increased by 5% and although loss value is higher at the early stage.

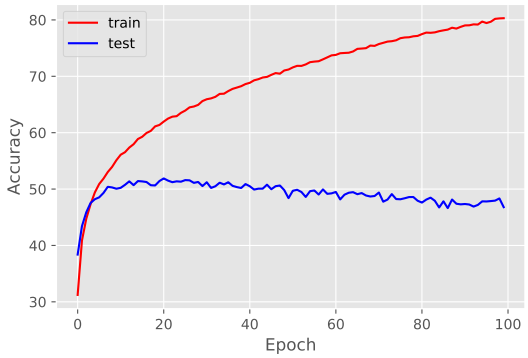

Figure 7: Accuracy of neural network has one hidden layer

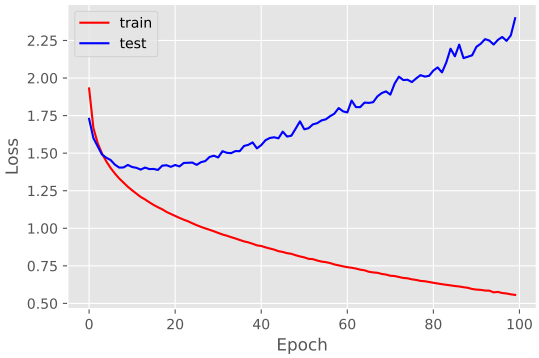

Figure 8: Losses of neural network has one hidden layer

Solution 2.3

After add dropout at the rate of 0.2 for the hidden layer. The idea under the hood is simple. It deactivates the neurons randomly at each training step instead of training the data on the original network, train the data on the network with dropped out nodes. In the next iteration of the training step, the hidden neurons which are deactivated by dropout changes because of its probabilistic behavior. In this way, by applying dropout i.e. . . deactivating certain individual nodes at random during training we can simulate an ensemble of neural network with different architectures.
The result shows that although it solves the over-fitting to some extent. But the performance of model is effected by dropout. It seems like the drop rate should be adjust to a reasonable value to make it better performance.

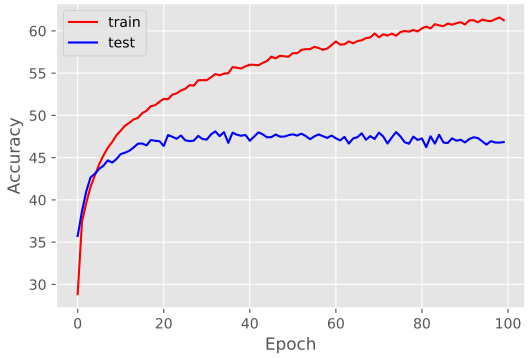

Figure 9: Accuracy of vanilla neural network of sub-task 3

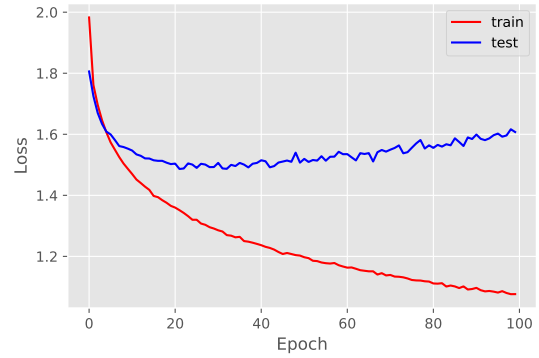

Figure 10: Loss of vanilla neural network of sub-task 3

Solution 2.4

In this problem, we freely change the architecture of ANN. So that we add two more hidden layer and *BatchNormalization* to the *Net* class. But unfortunately the performance have not improved. The final modified *Net* class are shown below:

```
class Net(torch.nn.Module):
    def __init__(self, one_more_hid=False, freestyle=False, vanilla=False,
                                                logistic=False):
        super(Net, self).__init__()
        self.vanilla = vanilla
        self.logistic = logistic
        self.dropout = dropout
        self.fc1 = torch.nn.Linear(image_cifar10_width*
                                                image_cifar10_height*
                                                color_channels, hidden_nodes)
        self.fc1_fashion = torch.nn.Linear(image_fashion_mnist_width*
                                                image_fashion_mnist_height,
                                                hidden_nodes)
        self.fc2 = None
        self.fc3 = None
        self.bn1 = None
        self.bn2 = None
        self.bn3 = None
        self.non_hidden = torch.nn.Linear(image_fashion_mnist_width*
                                                image_fashion_mnist_height,
                                                class_number_fashion_mnist)
        if one_more_hid:
            self.fc2 = torch.nn.Linear(hidden_nodes, hidden_nodes)
        if freestyle:
            self.bn1 = torch.nn.BatchNorm1d(num_features=64)
            self.fc2 = torch.nn.Linear(hidden_nodes, 128)
            self.bn2 = torch.nn.BatchNorm1d(num_features=128)
            self.fc3 = torch.nn.Linear(128, 64)
            self.bn3 = torch.nn.BatchNorm1d(num_features=64)
        if logistic:
            self.logi_fc = torch.nn.Linear(image_fashion_mnist_width*
                                                    image_fashion_mnist_height
                                                    , 1)
        self.out_cifar10 = torch.nn.Linear(hidden_nodes,
                                                class_number_cifar10)
        self.out_fashion_mnist = torch.nn.Linear(hidden_nodes,
                                                class_number_fashion_mnist)

    def forward(self, x):
        if dataset_name == "cifar10":
            x = x.view(-1, image_cifar10_width*image_cifar10_height*
                                                    color_channels)
            x = self.fc1(x)
            if self.bn1:
                x = self.bn1(x)
            x = torch.nn.functional.relu(x)
            x = torch.nn.Dropout(self.dropout)(x)
            if self.fc2:
```

```
                x = self.fc2(x)
                if self.bn2:
                    x = self.bn2(x)
                x = torch.nn.functional.relu(x)
                x = torch.nn.Dropout(self.dropout)(x)
            if self.fc3:
                x = self.fc3(x)
                if self.bn3:
                    x = self.bn3(x)
                x = torch.nn.functional.relu(x)
                x = torch.nn.Dropout(self.dropout)(x)
            return self.out_cifar10(x)
```

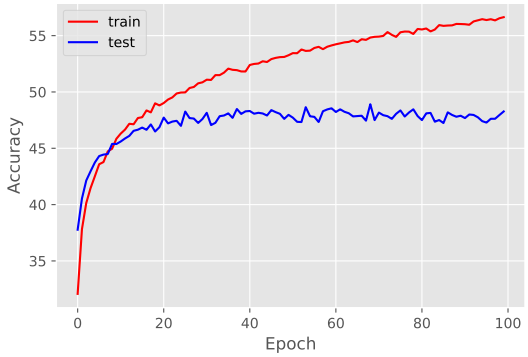

Figure 11: Accuracy of freestyle neural network

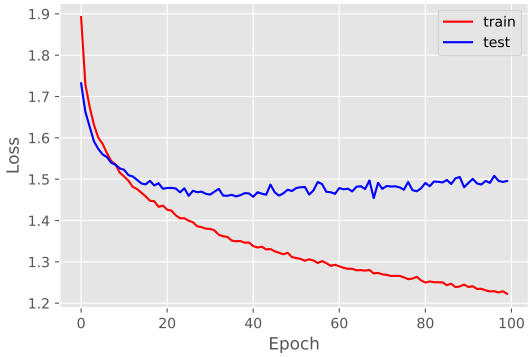

Figure 12: Loss of freestyle neural network

Solution 2.5

When we dive into *fashion MNIST* dataset. There are nothing much to change for the codes. The results shown in Figure 13 and 14.

Figure 13: Accuracy of neural network for fashion MNIST

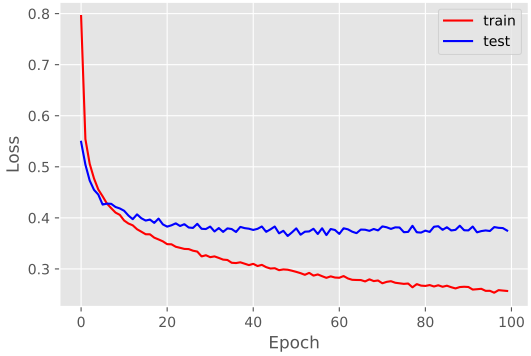

Figure 14: Losses of vanilla neural network for fashion MNIST

Solution 2.6

If we remove all the hidden layer and train the model. The results shown in Figure 13 and 14.

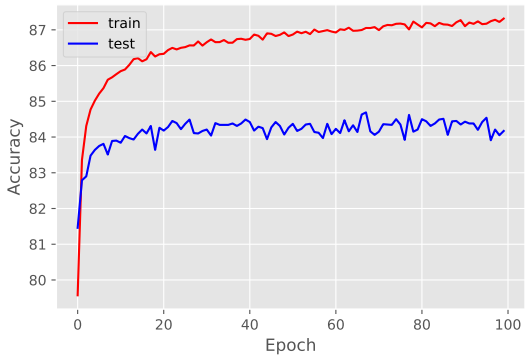

Figure 15: Accuracy of none layer neural network

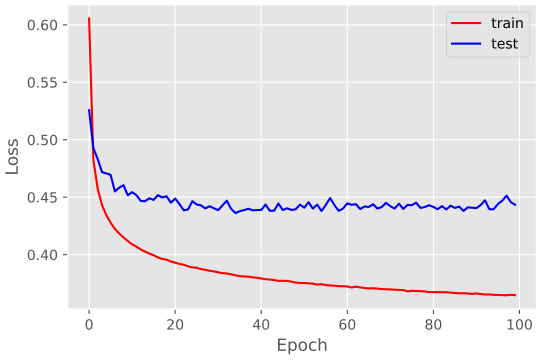

Figure 16: Loss of none layer neural network

The trained model contains the weight directly from input to output nodes. The visualization of the learned weights to the image dimension show that the weights have pretty much the same shape with train set images.

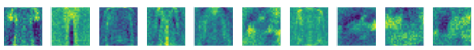

Figure 17: Network weights at epoch 0

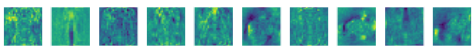

Figure 18: Network weights at epoch 25

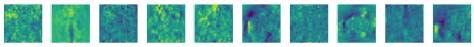

Figure 19: Network weights at epoch 75

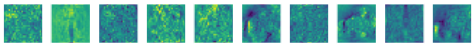

Figure 20: Network weights at epoch 95

Solution 2.7

For this problem, we select the first two classes and apply ANN model. The ROC curve along with the other results are shown below separately.

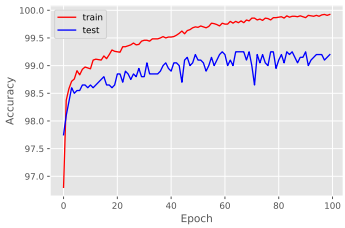

Figure 21: Accuracy of ANN

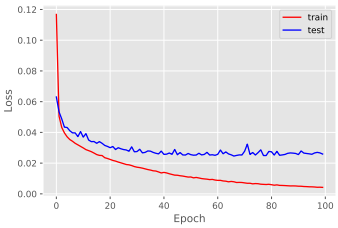

Figure 22: Loss of ANN

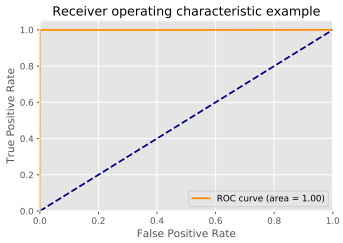

Figure 23: ROC curve of ANN

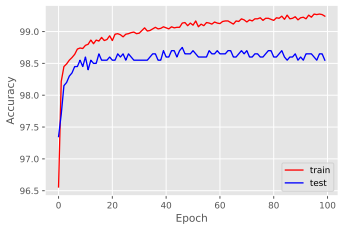

Figure 24: Accuracy of Logistic Regression

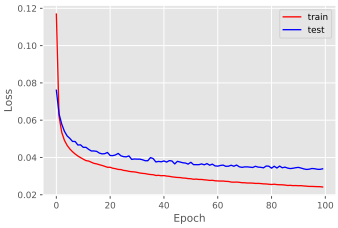

Figure 25: Loss of Logistic Regression

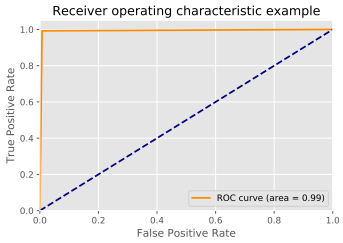

Figure 26: ROC curve of Logistic Regression

From those figures, we can easily obtain that the performance of both two models have good validity. The neural network method has better accuracy compare to logistic regression.

Reference

- https://medium.com/biaslyai/pytorch-linear-and-logistic-regression-models-5c5f0da2c
- https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html#sphx-glr-auto-examples-model-selection-plot-roc-py
- https://scikit-learn.org/stable/modules/model_evaluation.html#roc-metrics