All the functions are that are called in the code snippet of each solution are defined in the appendix. Alternately, the entire code suite can be found here - ***https://github.com/payalmohapatra/Deep-Learning-EE-435/tree/main/HW_1***

13.4 Nonlinear Autoencoder using neural networks Repeat the Autoencoder experiment described in Example 13.6 beginning with the implementation outlined in Section 13.2.6. You need not reproduce the projection map shown in the bottom-right panel of Figure 13.11.

Solution :

The autoencoder implementation details are,

- The encoder has 3 layers with tanh activation function. The input here is 2-dimensional and the encoder output is 1-dimension.
- The decoder uses the encoded lower dimensional output to decode back to the original input dimension. Below if the snapshot of the model.

```
AE(
  (encoder): Sequential(
    (0): Linear(in_features=2, out_features=10, bias=True)
    (1): Tanh()
    (2): Linear(in_features=10, out_features=10, bias=True)
    (3): Tanh()
    (4): Linear(in_features=10, out_features=1, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=1, out_features=10, bias=True)
    (1): Tanh()
    (2): Linear(in_features=10, out_features=10, bias=True)
    (3): Tanh()
    (4): Linear(in_features=10, out_features=2, bias=True)
  )
)
```

- Figure 1 shows the cost function for the autoencoder.

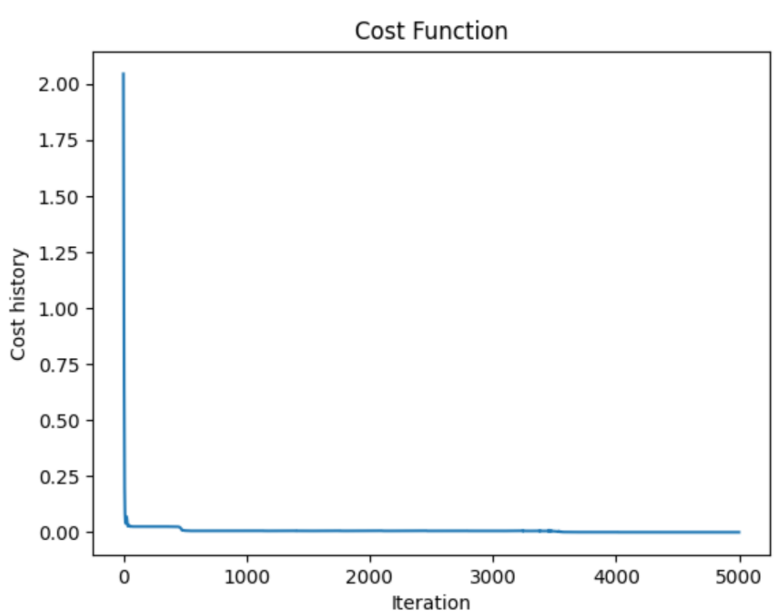


Figure 1 Cost History vs Iterations for Q.13.4.

Figure 2 shows the plot of the original input data and the decoded values from the autoencoder.

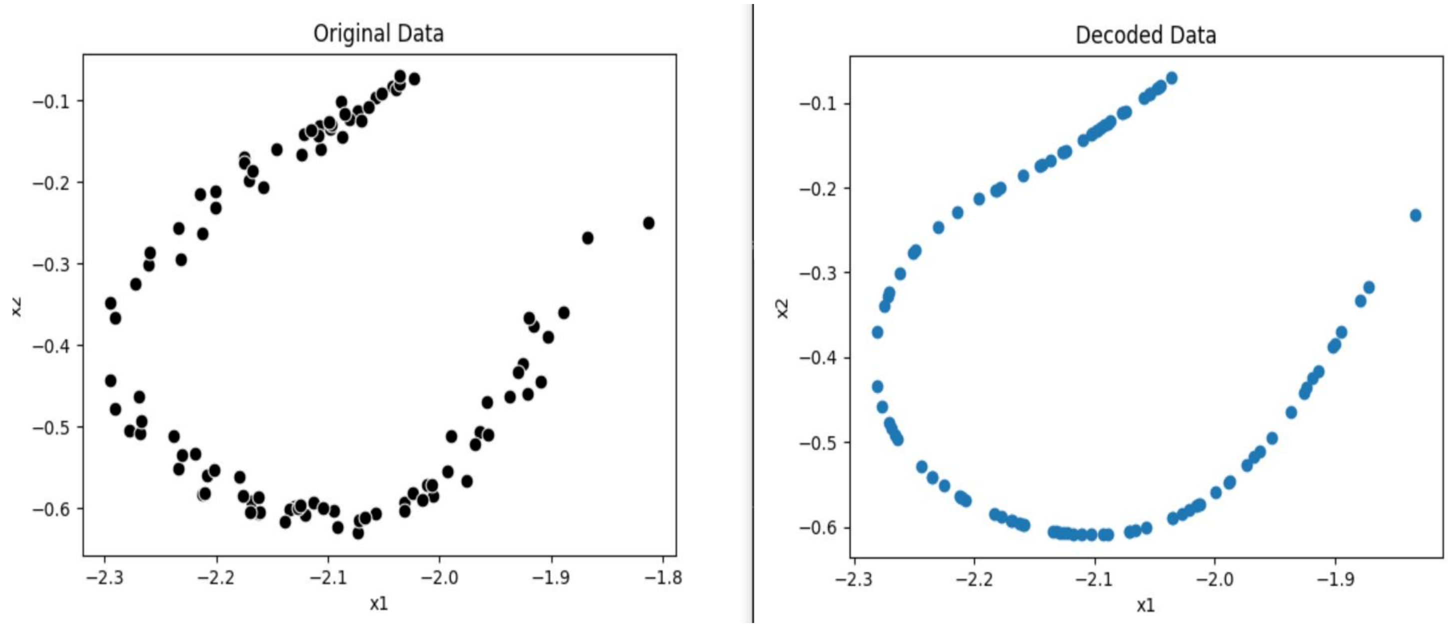


Figure 2 The left side plot shows the original data. On the left is the plot of the decoded input by the autoencoder.

Code :

```
writer = SummaryWriter()
# import data
datapath = 'Data/'
X = np.loadtxt(datapath + 'universal_autoencoder_samples.txt', delimiter=',')
print(np.shape(X))
plt.scatter(X[0,:], X[1,:], c = 'k', s = 60, linewidth = 0.75, edgecolor = 'w')
plt.title('Original Data')
plt.xlabel('x1')
plt.ylabel('x2')
#plt.show()
batch_size = 100
X = X.T
x = torch.from_numpy(X)

class GenDataset(Dataset) :
    def __init__(self,x) :
        # data loading
        self.x = x

    def __getitem__(self,index) :
        return self.x[index]

    def __len__(self) :
        return len(self.x)

dataset = GenDataset(x)

loader = DataLoader(dataset=dataset,
                          batch_size=batch_size,
                          shuffle=True,
                          num_workers=0)

# Creating a PyTorch class
# 1 * 2 ==> 1 ==> 1 * 2
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Building an linear encoder with Linear
        # layer followed by Relu activation function
        # 2 ==> 1
        self.encoder = torch.nn.Sequential(
```

```
            torch.nn.Linear(2, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 1)
        )

        # Building an linear decoder with Linear
        # layer followed by Relu activation function
        # The Sigmoid activation function
        # outputs the value between 0 and 1
        # 1 ==> 2
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(1, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 2)
            #torch.nn.ReLU()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        #print('Encoded =', encoded)
        decoded = self.decoder(encoded)
        #print('Decoded =',decoded)
        return decoded
# Model Initialization
model = AE()
print(model)

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss().float()
loss_function = loss_function.float()

# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(),
                             lr = 1e-2)
epochs = 5000
outputs = []
losses = []
#reconstructed_hist = []
# Train the model
for epoch in range(epochs):
    n_correct = 0
    reconstructed_hist = []
    for (features) in loader:
        # Forward pass
        reconstructed = model(features.float())
        loss = loss_function(reconstructed.float(), features.float())
        loss = loss.float()
        writer.add_scalar("Loss/train", loss, epoch)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Storing the losses in a list for plotting
        losses.append(loss.detach().numpy())
        reconstructed_np = reconstructed.detach().numpy()
        reconstructed_hist.append(reconstructed_np)

    outputs.append((epochs, reconstructed))

print(np.shape(reconstructed_hist))
reconstructed_arr = np.array(reconstructed_hist)
reconstructed_arr = np.reshape(reconstructed_arr, (100,2))
print(np.shape(reconstructed_arr))
plt.figure(3)
plt.scatter(reconstructed_arr[:,0], reconstructed_arr[:,1])
plt.title('Decoded Data')
plt.xlabel('x1')
plt.ylabel('x2')

plt.figure(2)
plt.plot(losses)
plt.title('Cost Function')
plt.xlabel('Iteration')
plt.ylabel('Cost history')
plt.show()
reconstrcuted_test_hist = []
with torch.no_grad():
```

```
    n_correct = 0
    for (features) in loader:
        reconstructed_test = model(features.float())
        reconstrcuted_test_hist.append(reconstructed_test)

print(len(reconstrcuted_test_hist))
writer.close()
```

13.8 Batch normalization Repeat the experiment described in Example 13.13, and produce plots like those shown in Figure 13.20. Your plots may not look precisely like those shown in this figure (but they should look similar).

Solution :

The model's cost function has not yet converged to the global minima. But we can observe the trend that with batch normalization the optimization is significantly faster and accuracy is also higher as seen in Figure 3 for the same number of iterations.

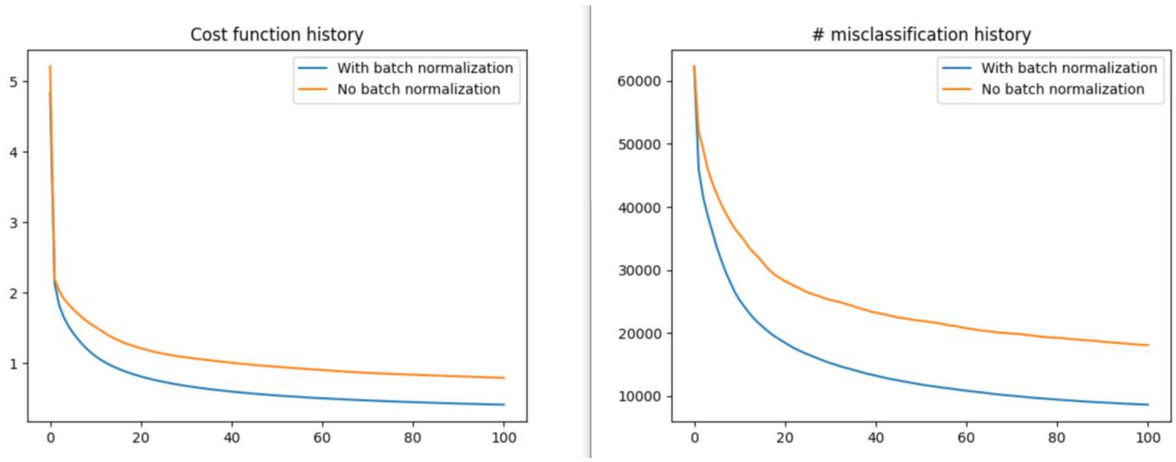


Figure 3 Left side plot shows the cost history for standard and batch normalized models and the plot on the right shows the respective cost history.

Code :

```
x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
# convert string labels to integers
y = np.array([int(v) for v in y])[:,np.newaxis]
x = DataFrame(x).to_numpy()
x = x.T
y = y.astype(int)
for i in range(np.shape(x)[0]):
    x[i,:] = (x[i,:] - np.mean(x[i,:]))/(np.std(x[i,:]))

x = DataFrame(x)
x = x.dropna()
x = DataFrame(x).to_numpy()

np.random.seed(0)

# create initial weights for a neural network model
def network_initializer(layer_sizes, scale):
    # container for all tunable weights
    weights = []
    # create appropriately -sized initial
    # weight matrix for each layer of network
    for k in range(len(layer_sizes) -1):

        # get layer sizes for current weight matrix
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k +1]

        # make weight matrix
```

```
        weight = scale* np. random. randn(U_k+ 1, U_k_plus_1)
        weights. append(weight)

        # repackage weights so that theta_init[0] contains all
        # weight matrices internal to the network, and theta_init[1]
        # contains final linear combination weights
        theta_init = [weights[:-1], weights[-1]]

    return theta_init

N = np.shape(x)[0]
# print('N is :', N)
U_1 = 10
U_L = 10
C = 10  ## if C is > 0 it is +1 class and if it is < 0 then -1 class
layer_sizes = [N, U_1, U_1, U_1, U_1, C]
scale = 1

theta_init = network_initializer(layer_sizes, scale)

def multisoftmax(a, y):

    index = np.array([range(np.size(y))]).T
    exp_a = np.reshape(np.log(np.sum(np.exp(a), axis=1)), (np.size(y),1)) - a[index,y.T]
    cost = np.sum(exp_a)

    return cost/float(np.size(y))

# # neural network feature transformation
def feature_transforms(a, w):
    # loop through each layer
    for W in w:
    # compute inner -product with current layer weights
        a = W[0] + np. dot(a.T , W[1:])
        # pass through activation
        a = np.maximum(a,0).T
        all_mean = np.reshape(np.mean(a, axis=1), (np.shape(a)[0],1))
        all_std = np.reshape(np.std(a,axis=1), (np.shape(a)[0],1))
        a = a - all_mean
        a = a/(all_std+10**-15)

    return a

def feature_transforms_wo(a, w):
    # loop through each layer
    for W in w:
    # compute inner -product with current layer weights
        a = W[0] + np. dot(a.T , W[1:])
        # pass through activation
        a = np.tanh(a). T
    return a

# neural network model
def model(theta,x,y,batch_inds):
    # print(np.size(batch_inds))
    y = np.reshape(y,(1, np.size(y)))
    # print('Again the size of x', np.shape(x))
    # print('Again the size of y', np.size(y))
    x_p = x[:,batch_inds]
    y_p = y[:,batch_inds]
    # print('size of y_p is', np.shape(y_p))
    # compute feature transformation
    f = feature_transforms(x_p, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])
    # a = a.T
    # # print('shape of a is:', np.shape(a))
    # all_mean = np.reshape(np.mean(a, axis=1), (np.shape(a)[0],1))
    # # print('shape of mean is', np.shape(all_mean))
    # all_std = np.reshape(np.std(a,axis=1), (np.shape(a)[0],1))
    # a = a - all_mean
    # a = a/all_std
    # a = a.T
    cost = multisoftmax(a, y_p)

    return cost

def model_wo(theta,x,y,batch_inds):
    # print(np.size(batch_inds))
    y = np.reshape(y,(1, np.size(y)))
    # print('Again the size of x', np.shape(x))
```

```
    # print('Again the size of y', np.size(y))
    x_p = x[:,batch_inds]
    y_p = y[:,batch_inds]
    # print('size of y_p is', np.shape(y_p))
    # compute feature transformation
    f = feature_transforms_wo(x_p, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])
    cost = multisoftmax(a, y_p)
    return cost

def prediction(theta):
    # compute feature transformation
    f = feature_transforms(x, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])

    return a

def prediction_wo(theta):
    # compute feature transformation
    f = feature_transforms_wo(x, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])

    return a

# a =  model(theta_init)
# # print('Size of a is', np.shape(a))

max_its = 100
alpha_choice = 0.1
batch_size = 600
# weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,theta_init)
weight_history,cost_history = gradient_descent_batch(model,theta_init,x,y,alpha_choice,max_its,batch_size)
print(cost_history)

# theta_init = network_initializer(layer_sizes, scale)
weight_history_wo,cost_history_wo = gradient_descent_batch(model_wo,theta_init,x,y,alpha_choice,max_its,batch_size)
print(cost_history_wo)

# print('shape of weight_history is:', np.shape(weight_history))

misclassification = np.zeros((np.size(cost_history),1))
misclassification_wo = np.zeros((np.size(cost_history_wo),1))

# for i in range(1):
for i in range(np.size(cost_history)):
    pred = prediction(weight_history[i])
    pred_wo = prediction_wo(weight_history_wo[i])
    # print('size of pred is:', np.shape(pred))
    pred = np.reshape(np.argmax(pred, axis=1), (1,np.size(y)))
    pred_wo = np.reshape(np.argmax(pred_wo, axis=1), (1,np.size(y)))

    # print('checking pred again', np.)
    # print('Shape oy pred is', np.shape(pred))
    pred = (pred == y.T)
    pred_wo = (pred_wo == y.T)
    misclassification[i] = np.size(y) - np.sum(pred)
    misclassification_wo[i] = np.size(y) - np.sum(pred_wo)
    # print(misclassification[i])

# print(misclassification)
# print('At the end of the analysis, the # of misclassifications are:', misclassification[-1])
# print('End weight is:', weight_history[-1])




plt.figure(1)
plt.plot(cost_history)
plt.plot(cost_history_wo)
plt.legend(["With batch normalization", "No batch normalization"])
plt.title('Cost function history')

plt.figure(2)
plt.plot(misclassification)
plt.plot(misclassification_wo)
```

```
plt.legend(["With batch normalization", "No batch normalization"])
plt.title('# misclassification history')

plt.show()
```

13.9 Early stopping cross-validation Repeat the experiment described in Example 13.14. You need not reproduce all the panels shown in Figure 13.21. However, you should plot the fit provided by the weights associated with the minimum validation error on top of the entire dataset.

Solution :

Early stopping is done when the validation error is the minimum. This helps is avoiding the scenario of overfitting the model to the input data. For this example the model is run in steps of 1000 iterations for 7 steps. Validation error is plotted as shown in Figure 4.

The Validation error is calculated as a mean-squared error for this example.

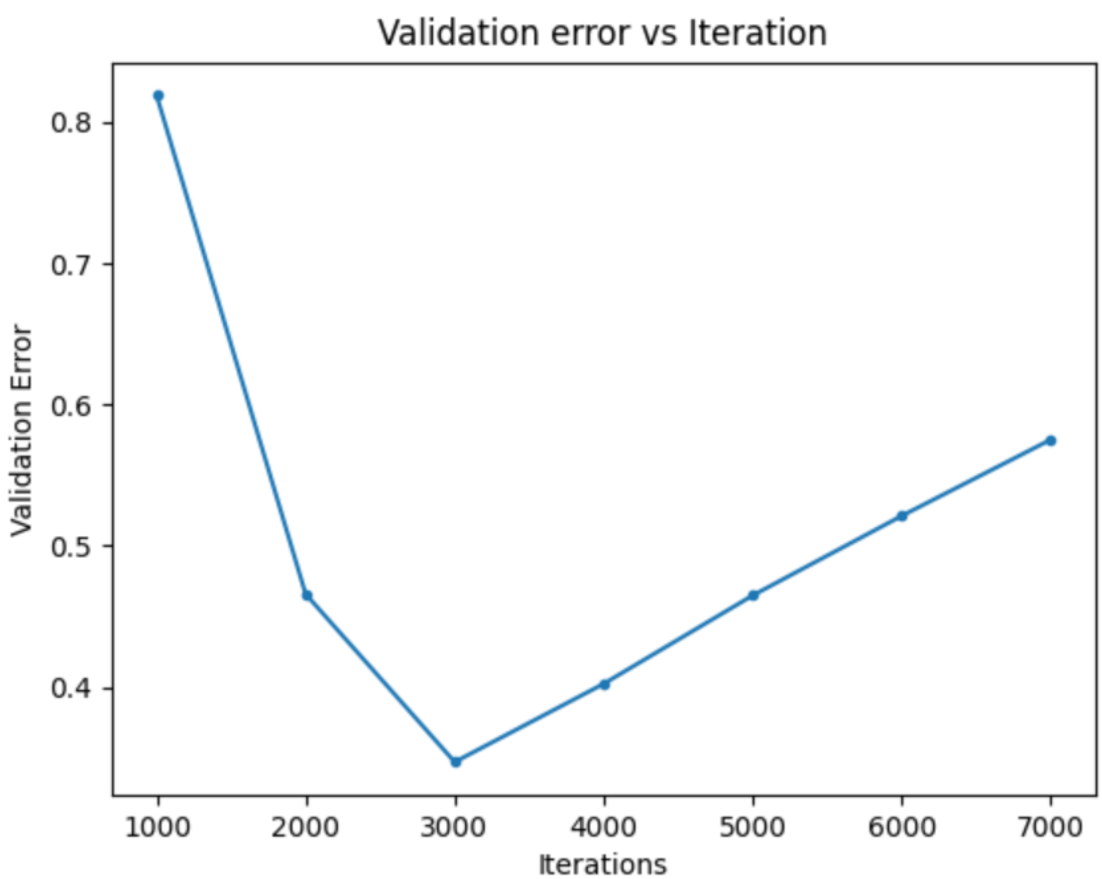


Figure 4 Validation error vs number of iterations

As we can see, the validation error is minimum for 3000 iterations.

Figure 5, Figure 6 and Figure 7 show the plot of the original validation, training and total dataset vs ther predicted counterpart.

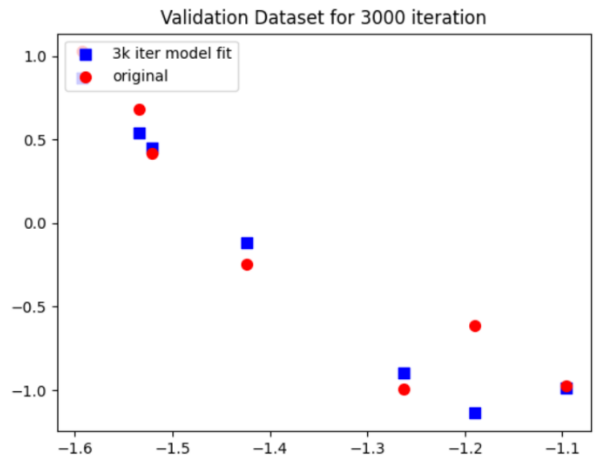


Figure 5 Plot of validation dataset vs. the predicted values after model is trained for 3000 iterations.

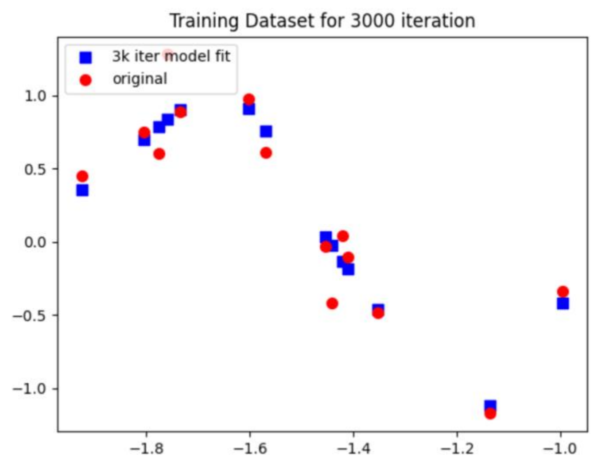


Figure 6 Plot of training dataset vs. the predicted values after model is trained for 3000 iterations.

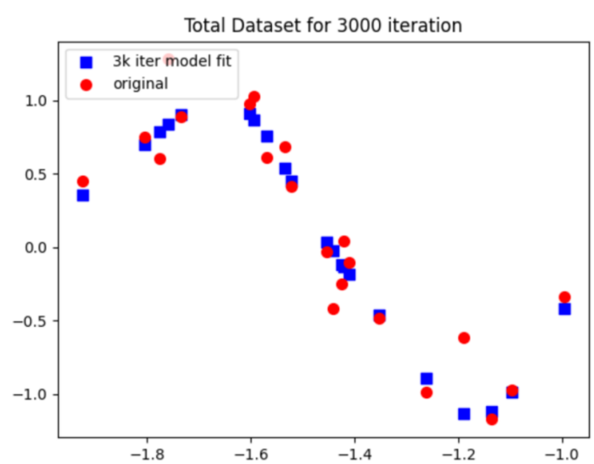


Figure 7 Plot of etire dataset vs. the predicted values after model is trained for 3000 iterations.

Payal Mohapatra
NetID : PMS4829
Student ID : 3334086 EE-435 – Homework 3

Code :

```
np.random.seed(27)
# load in dataset
datapath = 'Data/'
csvname = datapath + 'noisy_sin_sample.txt'
data = np.loadtxt(csvname, delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]
x_orig = x
y_orig = y
print(np.shape(x))
print(np.shape(y))



# Normalise inputs
for i in range(np.size(x,0)):
    row_mean = np.mean(x[i,:])
    row_std = np.std(x[i,:])
    x[i,:] = x[i,:]-row_mean/row_std

x_train, x_valid, y_train, y_valid = train_test_split(x.T, y.T, test_size=0.33)
#print(np.shape(x_train))
#print(np.shape(x_valid))
#breakpoint()
x = x_train.T
y = y_train.T
x_valid = x_valid.T
y_valid = y_valid.T
# neural network feature transformation
def feature_transforms(a, w):
    # loop through each layer
    for W in w:
        # compute inner -product with current layer weights
        a = W[0] + np. dot(a.T , W[1:])
        # pass through activation
        a = np.tanh(a). T
    return a

# neural network model
#def model(x, theta):
def model(theta,x):
    # compute feature transformation
    f = feature_transforms(x, theta[0])
    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])
    return a.T

# create initial weights for a neural network model
def network_initializer(layer_sizes, scale):
    # container for all tunable weights
    weights = []
    # create appropriately -sized initial
    # weight matrix for each layer of network
    for k in range(len(layer_sizes) -1):
        # get layer sizes for current weight matrix
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k +1]
        # make weight matrix
        weight = scale* np. random. randn(U_k+ 1, U_k_plus_1)
        weights. append(weight)

    # repackage weights so that theta_init[0] contains all
    # weight matrices internal to the network, and theta_init[1]
    # contains final linear combination weights
    theta_init = [weights[:-1], weights[-1]]

    return theta_init


def rmse_func(w) :
    cost = 0
    y_pred = model(w,x)
    #y_pred = y_pred._value
```

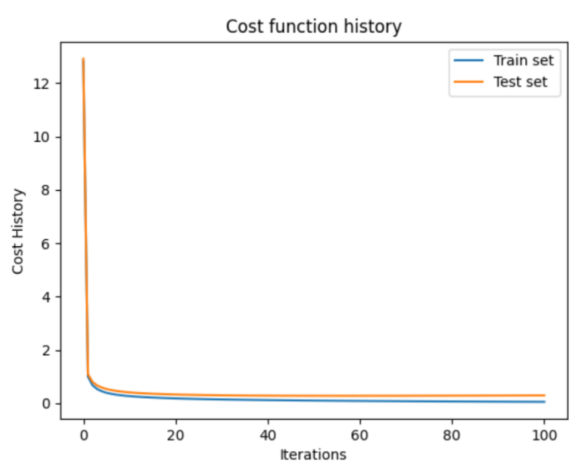
```
    for p in range(y.size):
       # get pth prediction/ output pair
       y_p = y[:,p]
       y_model = y_pred[:,p]
       ## add to current cost
       cost += (y_p - y_model)**2
    return cost/ float(np.size(y))



# Initialise as per Example 13.4 from text
layer_sizes = [np.shape(x)[0],10,10,10,1]
w_init = network_initializer(layer_sizes,1.0)
max_its = 1000
alpha_choice = 0.05
weight_history_1k,cost_history_1k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 2000
alpha_choice = 0.05
weight_history_2k,cost_history_2k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 3000
alpha_choice = 0.05
weight_history_3k,cost_history_3k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 4000
alpha_choice = 0.05
weight_history_4k,cost_history_4k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 5000
alpha_choice = 0.05
weight_history_5k,cost_history_5k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 6000
alpha_choice = 0.05
weight_history_6k,cost_history_6k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)

max_its = 7000
alpha_choice = 0.05
weight_history_7k,cost_history_7k = gradient_descent_nn(rmse_func,alpha_choice,max_its,w_init)




#plt.figure(4)
#plt.plot(cost_history)
#plt.title('Cost history vs Iteration')
#plt.xlabel('Iteration')
#plt.ylabel('Cost')
#plt.figure()
##plt.show()



#y_p = model(weight_history[-1],x_valid)
y_p_1k = model(weight_history_1k[-1],x_valid)
y_p_2k = model(weight_history_2k[-1],x_valid)
y_p_3k = model(weight_history_3k[-1],x_valid)
y_p_4k = model(weight_history_4k[-1],x_valid)
y_p_5k = model(weight_history_5k[-1],x_valid)
y_p_6k = model(weight_history_6k[-1],x_valid)
y_p_7k = model(weight_history_7k[-1],x_valid)
print('Accuracy calc')
print(np.shape(y_p_1k))
print(np.shape(y_valid))
breakpoint()

def error_calc(y_pred_tmp) :
    error = (y_pred_tmp - y_valid)**2
    err_acc = np.sum(error)
    return err_acc

err_1k = error_calc(y_p_1k)
err_2k = error_calc(y_p_2k)
err_3k = error_calc(y_p_3k)
err_4k = error_calc(y_p_4k)
err_5k = error_calc(y_p_5k)
```

```
err_6k = error_calc(y_p_6k)
err_7k = error_calc(y_p_7k)

err_all_iter = [err_1k, err_2k, err_3k, err_4k, err_5k, err_6k, err_7k]
iter_plt = [1000, 2000, 3000, 4000, 5000, 6000, 7000]
plt.figure(4)
plt.plot(iter_plt, err_all_iter,marker=".",)
plt.title('Validation error vs Iteration')
plt.xlabel('Iterations')
plt.ylabel('Validation Error')
#plt.show()
print(err_all_iter)
plt.figure(1)
plt.scatter(x_valid[0,:],y_p_3k[0,:], s=50, c='b', marker="s", label='3k iter model fit')
plt.scatter(x_valid[0,:],y_valid[0,:], s=50, c='r', marker="o", label='original')
plt.legend(loc='upper left')
plt.title ('Validation Dataset for 3000 iteration')

plt.figure(2)
y_p = model(weight_history_3k[-1],x)
print(np.shape(y_p))
plt.scatter(x[0,:],y_p[0,:], s=50, c='b', marker="s", label='3k iter model fit')
plt.scatter(x[0,:],y[0,:], s=50, c='r', marker="o", label='original')
plt.legend(loc='upper left')
plt.title ('Training Dataset for 3000 iteration')

plt.figure(3)
y_p = model(weight_history_3k[-1],x_orig)
print(np.shape(y_p))
plt.scatter(x_orig[0,:],y_p[0,:], s=50, c='b', marker="s", label='3k iter model fit')
plt.scatter(x_orig[0,:],y_orig[0,:], s=50, c='r', marker="o", label='original')
plt.legend(loc='upper left')
plt.title ('Total Dataset for 3000 iteration')

plt.show()
```

13.10 Handwritten digit recognition using neural networks Repeat the experiment described in Example 13.15, and produce cost/accuracy history plots like the ones shown in Figure 13.22. You may not reproduce exactly what is reported basedon your particular implementation. However, you should be able to achieve similar results as reported in Example 13.15.

Solution :

For 100 iterations, this implementation achieves training accuracy of 98.48% and a test accuracy of 94% as shown in Figure 8.



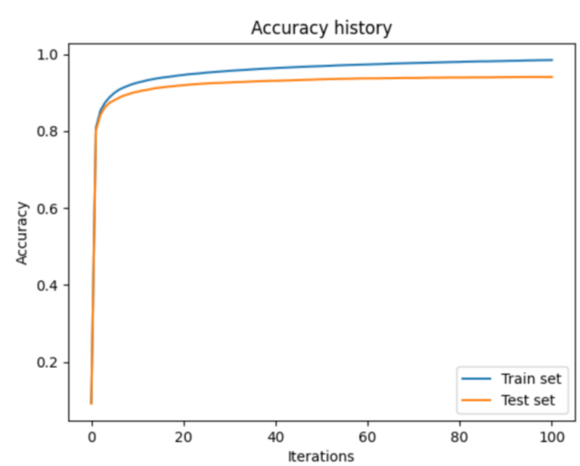


Figure 8 Left plot shows the cost history of the training and test datasets and the right plot shows the respective accuracy.

Payal Mohapatra
NetID : PMS4829
Student ID : 3334086 EE-435 – Homework 3

Code :

```
x, y = fetch_openml('mnist_784', version=1, return_X_y=True)
# convert string labels to integers
y = np.array([int(v) for v in y])[:,np.newaxis]
x = DataFrame(x).to_numpy()
x = x.T
y = y.astype(int)
for i in range(np.shape(x)[0]):
    x[i,:] = (x[i,:] - np.mean(x[i,:]))/(np.std(x[i,:]))

x = DataFrame(x)
x = x.dropna()
x = DataFrame(x).to_numpy()

x_train, x_test, y_train, y_test = model_selection.train_test_split(x.T, y, test_size=0.3, random_state=42)
x_train = x_train.T
x_test = x_test.T
print('Size of train input is:', np.shape(x_test))
print('Size of train output is:', np.shape(y_test))
print(type(x))
print(type(y))


np.random.seed(0)

# create initial weights for a neural network model
def network_initializer(layer_sizes, scale):
    # container for all tunable weights
    weights = []
    # create appropriately -sized initial
    # weight matrix for each layer of network
    for k in range(len(layer_sizes) -1):

        # get layer sizes for current weight matrix
        U_k = layer_sizes[k]
        U_k_plus_1 = layer_sizes[k +1]

        # make weight matrix
        weight = scale* np. random. randn(U_k+ 1, U_k_plus_1)
        weights. append(weight)

        # repackage weights so that theta_init[0] contains all
        # weight matrices internal to the network, and theta_init[1]
        # contains final linear combination weights
        theta_init = [weights[:-1], weights[-1]]

    return theta_init

N = np.shape(x)[0]
# print('N is :', N)
U_1 = 100
U_L = 100
C = 10  ## if C is > 0 it is +1 class and if it is < 0 then -1 class
layer_sizes = [N, U_1, U_1, C]
scale = 1

theta_init = network_initializer(layer_sizes, scale)
# print('Shape of theta_init is:', np.shape(theta_init))

def multisoftmax(a, y):

    index = np.array([range(np.size(y))]).T
    exp_a = np.reshape(np.log(np.sum(np.exp(a), axis=1)), (np.size(y),1)) - a[index,y.T]
    cost = np.sum(exp_a)

    return cost/float(np.size(y))

def feature_transforms(a, w):
    for W in w:
    # compute inner -product with current layer weights
        a = W[0] + np. dot(a.T , W[1:])
        # pass through activation
        a = np.maximum(a,0).T
```

```
        all_mean = np.reshape(np.mean(a, axis=1), (np.shape(a)[0],1))
        all_std = np.reshape(np.std(a,axis=1), (np.shape(a)[0],1))
        a = a - all_mean
        a = a/(all_std+10**-15)
    return a

# neural network model
def model(theta,x,y,batch_inds):
    # print(np.size(batch_inds))
    y = np.reshape(y,(1, np.size(y)))
    x_p = x[:,batch_inds]
    y_p = y[:,batch_inds]
    f = feature_transforms(x_p, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])

    cost = multisoftmax(a, y_p)

    return cost


def prediction(x, theta):
    # compute feature transformation
    f = feature_transforms(x, theta[0])

    # compute final linear combination
    a = theta[1][0] + np. dot(f.T, theta [1][1:])

    return a



max_its = 300
alpha_choice = 1
batch_size = 700
weight_history,cost_history = gradient_descent_batch(model,theta_init,x_train,y_train,alpha_choice,max_its,batch_size)


accuracy_train = np.zeros((np.size(cost_history),1))

for i in range(np.size(cost_history)):
    pred = prediction(x_train, weight_history[i])

    pred = np.reshape(np.argmax(pred, axis=1), (1,np.size(y_train)))
    pred = (pred == y_train.T)

    accuracy_train[i] = np.sum(pred)/np.size(y_train)


cost_test = np.zeros((np.size(cost_history),1))
accuracy_test = np.zeros((np.size(cost_history),1))

for i in range(np.size(cost_history)):
    pred = prediction(x_test, weight_history[i])
    batch_inds = np.arange(np.size(y_test))
    cost = model( weight_history[i],x_test,y_test,batch_inds)

    pred = np.reshape(np.argmax(pred, axis=1), (1,np.size(y_test)))
    pred = (pred == y_test.T)

    accuracy_test[i] = np.sum(pred)/np.size(y_test)
    cost_test[i] = cost



plt.figure(1)
plt.plot(cost_history)
plt.plot(cost_test)
plt.legend(["Train set", "Test set"])
plt.title('Cost function history')

plt.figure(2)
```

```
plt.plot(accuracy_train)
plt.plot(accuracy_test)
plt.legend(["Train set", "Test set"])
plt.title('# misclassification history')

plt.show()
```