Importing libraries

```
import argparse
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import pickle
from sklearn.decomposition import PCA
import sys
from pathlib import Path
```

Creating the skeleton of a network

```
def createnetwork(num_hidden, activation_func, sizes, inputsize = 784, outputsize = 10):
    # Appending the first element(number of inputs=784) and the last element(number of of
    # Sizes is a list representing the number of units in each layer.
    sizes = [inputsize] + sizes
    sizes = sizes + [outputsize]
    # Ex output of the above 2 lines: [784, 50, 100, 150, 10]. If there are 3 hidden layer
    np.random.seed(1234)
    # creating a dictionary for all the weights and biases
    parameters = {}
    if activation_func == "relu":
        for i in range(1, num_hidden+2):
          # Kaiming initialization for W, b i set to zeros
            parameters["W" + str(i)] = 0.01*np.random.randn(sizes[i], sizes[i-1])*(np.sqrt
            parameters["b" + str(i)] = np.zeros((sizes[i],1))
    else:
        for i in range(1, num_hidden+2):
            parameters["W" + str(i)] = np.random.randn(sizes[i], sizes[i-1])
            parameters["b" + str(i)] = np.random.randn(sizes[i],1)

    return parameters
```

Some mathematical functions

```
def sigmoid(z):
  return 1/(1 + np.exp(-z))

def tanh(z):
  return np.tanh(z)

def softmax(z):
    z = z-np.max(z)
    numer = np.exp(z)
```

```
    denom = np.sum(numer, axis = 0)
    return numer/denom

def convert_to_onehot(indices, num_classes):
    output = np.eye(num_classes)[np.array(indices).reshape(-1)]
    # each target vector is converted to a row vector. Each label is now a 10 dimensional
    return output.reshape(list(np.shape(indices))+[num_classes])

def squared_loss(X, Y):
    x = np.array(X)
    y = np.array(Y)
    # to hold off broadcasting behaviour in case of 1D arrays
    if x.ndim == 1:
        x = x[:,np.newaxis]
    if y.ndim == 1:
        y = y[:, np.newaxis]
    loss = 0.5*np.sum((y-x)**2)
    return loss


def cross_entropy_loss(X, Y):
    x = np.array(X)
    y = np.array(Y)
    # to hold off broadcasting behaviour in case of 1D arrays
    if x.ndim == 1:
        x = x[:,np.newaxis]
    if y.ndim == 1:
        y = y[:, np.newaxis]
    loss_vec = -1*y*np.log(x)
    return np.sum(loss_vec)
    # alternate implementation
    # x = np.array(X).reshape(-1)
    # y = np.array(Y).reshape(-1)
    # logx = np.log(x)
    # loss_vec = (-1)*(y*logx)
    # loss = np.sum(loss_vec)
    # return loss
```

Function for reading data and function for activating a neuro

```
def read_data(path_to_csv):
    df = pd.read_csv(path_to_csv)
    data = df.to_numpy()
    X = data[:,1:-1]
    y = data[:,-1]
    Y = [int(i) for i in y]
    return data, X.T, Y

def activate(z, activation):
    if activation == "sigmoid":
        return sigmoid(z)
    elif activation == "tanh":
```

```
        return tanh(z)
    elif activation == "relu":
        return (z>0)*(z) + 0.01*((z<0)*z)
```

Forward pass function

```
def forward_pass(X, parameters, activation, num_hidden):
    A = {}
    # To prevent broadcasting when a single input vector is given
    if X.ndim == 1:
        X = X[:, np.newaxis]

    #  Initializing the pre-activation to inputs
    H = {"h0":X}

    for l in range(1, num_hidden + 2):
        Wl = parameters["W" + str(l)]
        bl = parameters["b" + str(l)]

        hprev = H["h" + str(l-1)]
        al = np.dot(Wl,hprev) + bl
        A["a" + str(l)] = al

        if l != num_hidden + 1:
            hl = activate(al, activation)
        elif l == num_hidden + 1:
            hl = softmax(al)
        H["h" + str(l)] = hl

    yhat = H["h" + str(num_hidden + 1)]
    return yhat, A, H
```

Functions for generating gradients

```
def creategrads(num_hidden, sizes, inputsize = 784, outputsize = 10):
    sizes = [inputsize] + sizes
    sizes = sizes + [outputsize]
    grads = {"dh0":np.zeros((inputsize,1)),
            "da0":np.zeros((inputsize,1))}
    for i in range(1, num_hidden+2):
        grads["dW" + str(i)] = np.zeros((sizes[i], sizes[i-1]))
        grads["db" + str(i)] = np.zeros((sizes[i],1))
        grads["da" + str(i)] = np.zeros((sizes[i],1))
        grads["dh" + str(i)] = np.zeros((sizes[i],1))
    return grads


def grad_sigmoid(z):
    return (sigmoid(z))*(1 - sigmoid(z))
```

```
    return (sigmoid(z))*(1 - sigmoid(z))

def grad_tanh(z):
    return (1 - (np.tanh(z))**2)

def grad_relu(z):
    return (z>0)*(np.ones(np.shape(z))) + (z<0)*(0.01*np.ones(np.shape(z)))
```

Function for back propagation

```
def back_prop(H, A, parameters, num_hidden, sizes, Y, Yhat, loss, activation, inputsize, o
    grad_one_eg = creategrads(num_hidden, sizes, inputsize = 784, outputsize = 10)
    A["a0"] = np.zeros((inputsize,1))
    if Y.ndim == 1:
        Y = Y[:, np.newaxis]
    if Yhat.ndim == 1:
        Yhat = Yhat[:, np.newaxis]

    if loss == "ce":
      # Derivative of loss function with respect to the pre-activations of the output laye
        grad_one_eg["da" + str(num_hidden + 1)] = Yhat - Y
    elif loss == "sq":
        grad_one_eg["da" + str(num_hidden + 1)] = (Yhat - Y)*Yhat - Yhat*(np.dot((Yhat-Y).

    for i in np.arange(num_hidden + 1, 0, -1):
            # a = b + W*h.
            # grad_one_eg["da" + str(i)] this part of the below dot product is calculated

            # Derivative of loss function with respect to the weight matrix
            grad_one_eg["dW" + str(i)] = np.dot(grad_one_eg["da" + str(i)], (H["h" + str(i
            # Derivative of loss function with respect to the weight matrix
            grad_one_eg["db" + str(i)] = grad_one_eg["da" + str(i)]
            # Derivative of loss function wrt activation 'h'.
            grad_one_eg["dh" + str(i-1)] = np.dot((parameters["W" + str(i)]).T, grad_one_e

            if activation == "sigmoid":
                derv = grad_sigmoid(A["a" + str(i-1)])
            elif activation == "tanh":
                derv = grad_tanh(A["a" + str(i-1)])
            elif activation == "relu":
                derv = grad_relu(A["a" + str(i-1)])
            if derv.ndim == 1:
                derv = derv[:, np.newaxis]
            # Derivative of w.r.t to pre-activations('a') of the hidden layers.
            grad_one_eg["da" + str(i-1)] = (grad_one_eg["dh" + str(i-1)])*derv

    return grad_one_eg
```

Generating momenta

```
def createmomenta(num_hidden, sizes, inputsize = 784, outputsize = 10):
    sizes = [inputsize] + sizes
    sizes = sizes + [outputsize]
    momenta = {}
    for i in range(1, num_hidden+2):
        momenta["vW" + str(i)] = np.zeros((sizes[i], sizes[i-1]))
        momenta["vb" + str(i)] = np.zeros((sizes[i],1))
    return momenta

def createmomenta_squared(num_hidden, sizes, inputsize = 784, outputsize = 10):
    sizes = [inputsize] + sizes
    sizes = sizes + [outputsize]
    momenta = {}
    for i in range(1, num_hidden+2):
        momenta["mW" + str(i)] = np.zeros((sizes[i], sizes[i-1]))
        momenta["mb" + str(i)] = np.zeros((sizes[i],1))
    return momenta
```

Functions for finding the accuracy and reading the test data

```
def find_accuracy(yhat,y):
    a = np.argmax(yhat, axis = 0)
    b = np.argmax(y, axis = 0)
    return 100*(np.sum(a == b)/len(a))

def read_data_test(path_to_csv):
    df = pd.read_csv(path_to_csv)
    data = df.to_numpy()
    X = data[:,1:]
    indices = data[:,0]
    return data, X.T, indices
```

Measuring the performance and displaying output informatic

```
def measure_performance(X, Y, X_val, Y_val, params, activation_func, num_hidden, loss):
    # Performing forward pass on training set
    Yhat, _, _ = forward_pass(X, params, activation_func, num_hidden)
    train_acc = find_accuracy(Yhat, Y)
    train_err = 100 - train_acc
    if loss == "ce":
        train_loss = (cross_entropy_loss(Yhat,Y))
    elif loss == "sq":
        train_loss = (squared_loss(Yhat,Y))

    # Performing forward pass on validation set
    Yhat_val, _, _ = forward_pass(X_val, params, activation_func, num_hidden)
    val_acc = find_accuracy(Yhat_val, Y_val)
    val_err = 100 - val_acc
```

```
    if loss == "ce":
        valid_loss = cross_entropy_loss(Yhat_val, Y_val)
    elif loss == "sq":
        valid_loss = (squared_loss(Yhat_val, Y_val))

    return train_err, train_loss, val_err, valid_loss

def display_info(epoch, train_err, train_loss, val_err, valid_loss):
    print("epoch:" + str(epoch))
    print("train error: ", "%.2f" % train_err, " train loss: ", "%.2f" % train_loss,
        " validation error: ", "%.2f" % val_err, "valid loss: ", "%.2f" % valid_loss)
    print("\n")
```

Functions for creating submissions and log files

```
def create_submission(X_test, indices, params, activation_func, num_hidden, submission_pat
    # Prediction on the test data set
    Yhat_test, _, _ = forward_pass(X_test, params, activation_func, num_hidden)
    Yhat_test_classes = np.argmax(Yhat_test, axis = 0)
    output = np.array([indices, Yhat_test_classes])
    output = output.T
    sub = pd.DataFrame({"id": output[:,0], "label": output[:,1]})
    _ = sub.to_csv(submission_path, index = False)
    print("Created submission at " + submission_path)


def create_log_files(path_expt_dir, step_data):
    try:
        os.mkdir(path_expt_dir)
        print("expt_dir created at " + path_expt_dir)
    except FileExistsError:
        print("expt_dir already exists at " + path_expt_dir)

    f = open(path_expt_dir + str("log_train.txt"), "w")
    for step in step_data:
        line = "Epoch " + str(step[0]) + ", Step " + str(step[1])
        line = line +  ", Loss: " + str(np.round(step_data[step][0], decimals = 2)) # Trai
        line = line + ", Error: " + str(np.round(step_data[step][1], decimals = 2)) # Trai
        line = line + ", lr: " + str(step_data[step][4])
        line = line + "\n"
        f.write(line)
    f.close()

    f = open(path_expt_dir + str("log_val.txt"), "w")
    for step in step_data:
        line = "Epoch " + str(step[0]) + ", Step " + str(step[1])
        line = line +  ", Loss: " + str(np.round(step_data[step][2], decimals = 2)) # Vali
        line = line + ", Error:" + str(np.round(step_data[step][3], decimals = 2))  # Vali
        line = line + ", lr: " + str(step_data[step][4])
        line = line + "\n"
        f.write(line)
    f.close()
    print("log files created")
```

```
        print( log files created )


    # Function to create a file with all the hyper parameters used for modelling.
    def create_readme(path_expt_dir, run_details):
        try:
            os.mkdir(path_expt_dir)
            print("expt_dir created at " + path_expt_dir)
        except FileExistsError:
            print("expt_dir already exists at " + path_expt_dir)
        f = open(path_expt_dir + str("readme.txt"), "w")
        f.write(run_details + "\n")
        f.write
        f.close()
```

Function for Adam

```
    def adam(X, Y, X_val, Y_val, activation_func, loss_func, eta, num_epochs, num_hidden, size
     inputsize = 784, outputsize = 10, beta1 = 0.9, beta2 = 0.999, eps = 1e-8, anneal = True,
        print("Adam optimizer is being used.")
        step_data = {}
        epoch_data = []

        if pretrain == False:
            params = createnetwork(num_hidden, activation_func, sizes, inputsize, outputsize)
        elif pretrain == True:
            params = load_params(path_save_dir, state)

        prev_momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)
        momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)
        momenta_hat = createmomenta(num_hidden, sizes, inputsize, outputsize)

        prev_momenta_squared = createmomenta_squared(num_hidden, sizes, inputsize, outputsize)
        momenta_squared = createmomenta_squared(num_hidden, sizes, inputsize, outputsize)
        momenta_squared_hat = createmomenta_squared(num_hidden, sizes, inputsize, outputsize)
        pointsseen = 0
        epoch = 0
        while epoch < (num_epochs):
            grads = creategrads(num_hidden, sizes, inputsize, outputsize)
            step = 0
            for j in range(0, 55000):
                x = X[:,j]
                y = Y[:,j]
                yhat, A, H = forward_pass(x, params, activation_func, num_hidden)
                grad_current = back_prop(H, A, params, num_hidden, sizes, y, yhat, loss_func,
                for key in grads:
                    grads[key] = grads[key] + grad_current[key]
                pointsseen = pointsseen + 1

                if pointsseen% batch_size == 0:
                    step = step + 1
```

```
                for newkey in params:
                  #  My stuff
                    momenta["v" + newkey] = beta2*prev_momenta["v" + newkey] + (1 - beta2)
                    momenta_squared["m" + newkey] = beta1*prev_momenta_squared["m" + newke

                    momenta_hat["v" + newkey] = momenta["v" + newkey]/(1 - np.power(beta2,
                    momenta_squared_hat["m" + newkey] = momenta_squared["m" + newkey]/(1 -

                    params[newkey] = params[newkey] - (eta/np.sqrt(momenta_hat["v" + newke

                    prev_momenta["v" + newkey] = momenta["v" + newkey]
                    prev_momenta_squared["m" + newkey] = momenta_squared["m" + newkey]


                  # His stuff
                    # momenta["v" + newkey] = beta1*prev_momenta["v" + newkey] + (1 - beta
                    # momenta_squared["m" + newkey] = beta2*prev_momenta_squared["m" + new

                    # momenta_hat["v" + newkey] = momenta["v" + newkey]/(1 - np.power(beta
                    # momenta_squared_hat["m" + newkey] = momenta_squared["m" + newkey]/(1

                    # params[newkey] = params[newkey] - (eta/np.sqrt(momenta_squared_hat["

                    # prev_momenta["v" + newkey] = momenta["v" + newkey]
                    # prev_momenta_squared["m" + newkey] = momenta_squared["m" + newkey]

                grads = creategrads(num_hidden, sizes, inputsize, outputsize)

                if step%100 == 0:
                    train_err, train_loss, val_err, valid_loss = measure_performance(X, Y,
                    step_data[(epoch, step)] = [train_loss, train_err, valid_loss, val_err


        train_err, train_loss, val_err, valid_loss = measure_performance(X, Y, X_val, Y_va

        if anneal and epoch >=1 and epoch_data[epoch - 1][2] <= valid_loss:
            eta = eta/2
            params = load_params(path_save_dir, epoch - 1)
            epoch = epoch - 1
            print("anneal")
        else:
            display_info(epoch, train_err, train_loss, val_err, valid_loss)
            epoch_data.append([epoch, train_loss, valid_loss])
            pickle_params(params, epoch, path_save_dir)

        epoch = epoch + 1
    return params, step_data, epoch_data
```

Function for Momentum GD

```
def mgd(X, Y, X_val, Y_val, activation_func, loss_func, eta, gamma, num_epochs, num_hidden
```

```
    inputsize = 784, outputsize = 10, anneal = True, pretrain = False, state = 0):
        print("momentum gradient descent")
        step_data = {}
        epoch_data = []

        if pretrain == False:
            params = createnetwork(num_hidden, activation_func, sizes, inputsize, outputsize)
        elif pretrain == True:
            params = load_params(path_save_dir, state)

        prev_momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)
        momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)
        pointsseen = 0

        epoch = 0
        while epoch < num_epochs:
            grads = creategrads(num_hidden, sizes, inputsize, outputsize)
            step = 0
            for j in range(0, 55000):
                x = X[:,j]
                y = Y[:,j]
                yhat, A, H = forward_pass(x, params, activation_func, num_hidden)
                grad_current = back_prop(H, A, params, num_hidden, sizes, y, yhat, loss_func,
                for key in grads:
                    grads[key] = grads[key] + grad_current[key]
                pointsseen = pointsseen + 1

                if pointsseen% batch_size == 0:
                    for newkey in params:
                        momenta["v" + newkey] = gamma*prev_momenta["v" + newkey] + eta*grads["
                        params[newkey] = params[newkey] - momenta["v" + newkey]
                        prev_momenta["v" + newkey] = momenta["v" + newkey]
                    grads = creategrads(num_hidden, sizes, inputsize, outputsize)

                    step = step + 1
                    if step%100 == 0:
                        train_err, train_loss, val_err, valid_loss = measure_performance(X, Y,
                        step_data[(epoch, step)] = [train_loss, train_err, valid_loss, val_err

            train_err, train_loss, val_err, valid_loss = measure_performance(X, Y, X_val, Y_va

            if anneal and epoch >=1 and epoch_data[epoch - 1][2] <= valid_loss:
                eta = eta/2
                params = load_params(path_save_dir, epoch - 1)
                epoch = epoch - 1
            else:
                display_info(epoch, train_err, train_loss, val_err, valid_loss)
                epoch_data.append([epoch, train_loss, valid_loss])
                pickle_params(params, epoch, path_save_dir)

            epoch = epoch + 1

        return params, step_data, epoch_data
```

Function for SGD

```
def sgd(X, Y, X_val, Y_val, activation_func, loss_func, eta, num_epochs, num_hidden, sizes
 inputsize = 784, outputsize = 10, anneal = True, pretrain = False, state = 0):
    print("Gradient decent with minibatch is used.")
    step_data = {}
    epoch_data = []

    if pretrain == False:
        params = createnetwork(num_hidden, activation_func, sizes, inputsize, outputsize)
    elif pretrain == True:
        params = load_params(path_save_dir, state)


    pointsseen = 0
    epoch = 0
    while epoch < num_epochs:
        # creation of zero grad vectors at the start of every epoch
        grads = creategrads(num_hidden, sizes, inputsize, outputsize)
        step = 0

        # iterate through every data point
        for j in range(0, 55000):
            x = X[:,j]
            y = Y[:,j]

            # perform forward pass and getting a prediction
            yhat, A, H = forward_pass(x, params, activation_func, num_hidden)
            # performing back propagation and generating new gradients
            grad_current = back_prop(H, A, params, num_hidden, sizes, y, yhat, loss_func,
            # cumulating the gradient values
            for key in grads:
                grads[key] = grads[key] + grad_current[key]

            pointsseen = pointsseen + 1

            # Check if a batch is complete. If so, then perform GD, update the parameters
            if pointsseen % batch_size == 0:
                for newkey in params:
                    params[newkey] = params[newkey] - eta*(grads["d" + newkey]/batch_size)
                grads = creategrads(num_hidden, sizes, inputsize, outputsize)

                # entering the if implies one step(batch) is done, so update step
                step = step + 1
                # store data for log files if 100 steps are done
                if step%100 == 0:
                    train_err, train_loss, val_err, valid_loss = measure_performance(X, Y,
                    step_data[(epoch, step)] = [train_loss, train_err, valid_loss, val_err

        train_err, train_loss, val_err, valid_loss = measure_performance(X, Y, X_val, Y_va

        # Start anneling learning rate if the validation loss of the previous epoch is les
```

```
        if anneal and epoch >=1 and epoch_data[epoch - 1][2] <= valid_loss:
            eta = eta/2
            params = load_params(path_save_dir, epoch - 1)
            epoch = epoch - 1
        else:
            display_info(epoch, train_err, train_loss, val_err, valid_loss)
            epoch_data.append([epoch, train_loss, valid_loss])
            pickle_params(params, epoch, path_save_dir)

        epoch = epoch + 1
    return params, step_data, epoch_data
```

Function for NAG

```
def nag(X, Y, X_val, Y_val, activation_func, loss_func, eta, gamma, num_epochs, num_hidden
 inputsize = 784, outputsize = 10, anneal = True, pretrain = False, state = 0):
    print("NAG")
    step_data = {}
    epoch_data = []

    if pretrain == False:
        params = createnetwork(num_hidden, activation_func, sizes, inputsize, outputsize)
    elif pretrain == True:
        params = load_params(path_save_dir, state)

    prev_momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)
    momenta = createmomenta(num_hidden, sizes, inputsize, outputsize)

    pointsseen = 0
    epoch = 0
    while epoch < (num_epochs):
        grads = creategrads(num_hidden, sizes, inputsize, outputsize)
        step = 0
        for j in range(0, 55000):
            x = X[:,j]
            y = Y[:,j]
            yhat, A, H = forward_pass(x, params, activation_func, num_hidden)
            grad_current = back_prop(H, A, params, num_hidden, sizes, y, yhat, loss_func,
            for key in grads:
                grads[key] = grads[key] + grad_current[key]
            pointsseen = pointsseen + 1
            if pointsseen% batch_size == 0:
                step = step + 1

                for newkey in params:
                    momenta["v" + newkey] = gamma*prev_momenta["v" + newkey] + eta*grads["
                    params[newkey] = params[newkey] - momenta["v" + newkey]
                    prev_momenta["v" + newkey] = momenta["v" + newkey]
                grads = creategrads(num_hidden, sizes, inputsize, outputsize)

                if step%100 == 0:
                    train_err, train_loss, val_err, valid_loss = measure_performance(X, Y,
                    step_data[(epoch, step)] = [train_loss, train_err, valid_loss, val_err
```

```
                    step_data[(epoch, step)] = [train_loss, train_err, valid_loss, val_err

                for next_key in params:
                    # Try to place this in the above for loop----------------------------
                    momenta["v" + next_key] = gamma*prev_momenta["v" + next_key]
                    params[next_key] = params[next_key] - momenta["v" + next_key]

        train_err, train_loss, val_err, valid_loss = measure_performance(X, Y, X_val, Y_va

        if anneal and epoch >=1 and epoch_data[epoch - 1][2] <= valid_loss:
            eta = eta/2
            params = load_params(path_save_dir, epoch - 1)
            epoch = epoch - 1
            print("anneal")
        else:
            display_info(epoch, train_err, train_loss, val_err, valid_loss)
            epoch_data.append([epoch, train_loss, valid_loss])
            pickle_params(params, epoch, path_save_dir)


        epoch = epoch + 1
    return params, step_data, epoch_data
```

Function for reading train validation and test data

```
# Read train, validation and test data
def init_data(path_train, path_val, path_test):
    data, X, y = read_data(path_train)

    Y = (convert_to_onehot(y, 10)).T
    X = (X.T/255)
    pca = PCA(n_components=50)
    pca.fit(X)
    X = pca.transform(X)
    X = X.T
    print(np.shape(X), np.shape(y), np.shape(data))

    data, X_val, y_val = read_data(path_val)

    Y_val = (convert_to_onehot(y_val, 10)).T
    X_val = (X_val.T/255)
    X_val = pca.transform(X_val)
    X_val = X_val.T
    print(np.shape(X_val), np.shape(y_val), np.shape(data))


    data, X_test, indices = read_data_test(path_test)

    X_test = X_test.T/255
    X_test = pca.transform(X_test)
    X_test = X_test.T
    #print(np.shape(X_test), "shape of test data")
```

```
    #np.savetxt("test_2.csv",X_test, delimiter = ",")

    return X, Y, X_val, Y_val, X_test, indices


def pickle_params(params, epoch, path_save_dir):
    try:
        os.mkdir(path_save_dir)
        print("save_dir created at " + path_save_dir)
    except FileExistsError:
        print("save_dir already exists at " + path_save_dir)
    filename = path_save_dir + "weights_" + str(epoch) + ".pickle"
    with open(filename, 'wb') as handle:
        pickle.dump(params, handle, protocol=pickle.HIGHEST_PROTOCOL)

def load_params(path_save_dir, epoch):
    if os.path.isdir(path_save_dir):
        filename = path_save_dir + "weights_" + str(epoch) + ".pickle"
        with open(filename, 'rb') as handle:
            parameters = pickle.load(handle)
        return parameters
    else:
        print("No directory at " + path_save_dir)
```

Using all the above functions wih different test cases

```
from pathlib import Path
# setting variables globally
train = Path('/content/train.csv')
val = Path('/content/valid.csv')
test = Path('/content/test.csv')
save_dir= '/content/save_dir'
expt_dir= '/content/expt_dir'

path_save_dir = save_dir
path_expt_dir = expt_dir
path_train = train
path_val = val
path_test = test
pretrain = False
testing = False


def run_model(eta, gamma, num_hidden, sizes, activation_func, loss_func, optim, batch_size
  # Reading data
  if testing == False:
      X, Y, X_val, Y_val, X_test, indices = init_data(path_train, path_val, path_test)
  elif testing == True:
      data = pd.read_csv(path_test, header = None)
      X_test = data.to_numpy()
      indices = np.arange(np.shape(data)[1])
```

```
    # Training part
    if testing == False:
        if optim == "gd":
            params, step_data, epoch_data = sgd(X, Y, X_val, Y_val, activation_func, loss_fu
        elif optim == "momentum":
            params, step_data, epoch_data = mgd(X, Y, X_val, Y_val, activation_func, loss_fu
        elif optim == "adam":
            params, step_data, epoch_data = adam(X, Y, X_val, Y_val, activation_func, loss_f
        elif optim == "nag":
            params, step_data, epoch_data = nag(X, Y, X_val, Y_val, activation_func, loss_fu

        # Testing part
        create_log_files(path_expt_dir, step_data)
        # create_readme(path_expt_dir, run_details)
        submission_path = path_expt_dir + "test_submission.csv"
    else:
        params = load_params(path_save_dir, state)
        num_hidden = int(len(params.keys())/2 - 1)
        print("num_hidden", num_hidden)
        activation_func = "relu"
        submission_path = path_expt_dir + "predictions_" + str(state) + ".csv"


    create_submission(X_test, indices, params, activation_func, num_hidden, submission_path)
    return epoch_data


def plot_stuff(result_list, color_list, description_list, title):
    plt.figure(figsize=(10,8))
    for i in range(0,len(result_list)):
      color = color_list[i]
      des = description_list[i]
      epoch_data = result_list[i]
      epoch_data_np = np.asarray(epoch_data)
      plt.plot(epoch_data_np.T[0], epoch_data_np.T[1], color,label=des)

    plt.xlabel("Number of Iterations")
    plt.ylabel("Loss values")
    plt.grid()
    plt.title(title[0])
    plt.legend()
    plt.show()

    plt.figure(figsize=(10,8))
    for i in range(0,len(result_list)):
      color = color_list[i]
      des = description_list[i]
      epoch_data = result_list[i]
      epoch_data_np = np.asarray(epoch_data)
      plt.plot(epoch_data_np.T[0], epoch_data_np.T[2], color,label=des)

    plt.xlabel("Number of Iterations")
    plt.ylabel("Loss values")
    plt.grid()
    plt.title(title[1])
    plt.legend()
```

```
    plt.legend()
    plt.show()

    return
```

For all the above 4 cases you will use sigmoid activation, cross entropy loss, Adam, batch size 20 a
For each of the 4 questions above you need to draw the following plots:

Training and Validation loss for different sizes with two hidd

```
result_50 = run_model(0.005, 0.5, 2, [50,50], 'relu', 'ce', 'adam', 20, 10, path_save_dir,
result_100 = run_model(0.005, 0.5, 2, [100,100], 'relu', 'ce', 'adam', 20, 10, path_save_d
result_200 = run_model(0.005, 0.5, 2, [200,200], 'relu', 'ce', 'adam', 20, 10, path_save_d
result_300 = run_model(0.005, 0.5, 2, [300,300], 'relu', 'ce', 'adam', 20, 10, path_save_d

result_list = [result_50, result_100, result_200, result_300]
color_list = ['r', 'b', 'g', 'm']
description_list = ['50 hidden units', '100 hidden units', '200 hidden units', '300 hidden
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different sizes with one hidd

```
# # (eta, gamma, num_hidden, sizes, activation_func, loss_func, optim, batch_size, num_epo
result_50_1 = run_model(0.005, 0.5, 1, [50], 'relu', 'ce', 'adam', 20, 10, path_save_dir,
result_100_1 = run_model(0.005, 0.5, 1, [100], 'relu', 'ce', 'adam', 20, 10, path_save_dir
result_200_1 = run_model(0.005, 0.5, 1, [200], 'relu', 'ce', 'adam', 20, 10, path_save_dir
result_300_1 = run_model(0.005, 0.5, 1, [300], 'relu', 'ce', 'adam', 20, 10, path_save_dir

result_list = [result_50_1, result_100_1, result_200_1, result_300_1]
color_list = ['r', 'b', 'g', 'm']
description_list = ['50 hidden units', '100 hidden units', '200 hidden units', '300 hidden
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different sizes with three hid

```
# # (eta, gamma, num_hidden, sizes, activation_func, loss_func, optim, batch_size, num_epo
result_50_3 = run_model(0.005, 0.5, 3, [50, 50, 50], 'relu', 'ce', 'adam', 20, 15, path_sa
result_100_3 = run_model(0.005, 0.5, 3, [75, 75, 75], 'relu', 'ce', 'adam', 20, 15, path_s
result_200_3 = run_model(0.005, 0.5, 3, [100, 100, 100], 'relu', 'ce', 'adam', 20, 15, pat
result_300_3 = run_model(0.005, 0.5, 3, [125, 125, 125], 'relu', 'ce', 'adam', 20, 15, pat

result_list = [result_50_3, result_100_3, result_200_3, result_300_3]
color_list = ['r', 'b', 'g', 'm']
```

```
description_list = ['50 hidden units', '100 hidden units', '200 hidden units', '300 hidden
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different sizes with four hidd

```
# # (eta, gamma, num_hidden, sizes, activation_func, loss_func, optim, batch_size, num_epo
result_50_4 = run_model(0.005, 0.5, 4, [50, 50, 50, 50], 'relu', 'ce', 'adam', 20, 15, pat
result_100_4 = run_model(0.005, 0.5, 4, [75, 75, 75, 75], 'relu', 'ce', 'adam', 20, 15, pa
result_200_4 = run_model(0.005, 0.5, 4, [100, 100, 100, 100], 'relu', 'ce', 'adam', 20, 15
result_300_4 = run_model(0.005, 0.5, 4, [125, 125, 125, 125], 'relu', 'ce', 'adam', 20, 15

result_list = [result_50_4, result_100_4, result_200_4, result_300_4]
color_list = ['r', 'b', 'g', 'm']
description_list = ['50 hidden units', '75 hidden units', '100 hidden units', '125 hidden
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different learning algorithms

```
# # (eta, gamma, num_hidden, sizes, activation_func, loss_func, optim, batch_size, num_epo
result_adam = run_model(0.005, 0.5, 2, [50, 50], 'relu', 'ce', 'adam', 20, 15, path_save_d
result_nag = run_model(0.005, 0.5, 2, [50, 50], 'relu', 'ce', 'nag', 20, 15, path_save_dir
result_mgd = run_model(0.005, 0.5, 2, [50, 50], 'relu', 'ce', 'momentum', 20, 15, path_sav



result_list = [result_adam, result_nag, result_mgd]
color_list = ['r', 'b', 'g']
description_list = ['Adam', 'Nestrov accelerated GD', 'Momentum GD']
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different activation functions

```
result_sig = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 20, 15, path_sa
result_tanh = run_model(0.005, 0.5, 2, [100, 100], 'tanh', 'ce', 'adam', 20, 15, path_save

result_list = [result_sig, result_tanh]
color_list = ['r', 'b']
description_list = ['Sigmoid activation', 'Tanh activation']
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different Loss functions

```
result_cross = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 20, 15, path_
result_square = run_model(0.005, 0.5, 2, [100, 100], 'tanh', 'sq', 'adam', 20, 15, path_sa

result_list = [result_cross, result_square]
color_list = ['r', 'b']
description_list = ['Cross entropy loss', 'Squared error loss']
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Training and Validation loss for different batch sizes

```
result_1 = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 1, 15, path_save_
result_20 = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 20, 15, path_sav
result_100 = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 100, 15, path_s
result_1000 = run_model(0.005, 0.5, 2, [100, 100], 'sigmoid', 'ce', 'adam', 1000, 15, path

result_list = [result_1, result_20, result_100, result_1000]
color_list = ['r', 'b', 'g', 'm']
description_list = ['Batch size = 1', 'Batch size = 20', 'Batch size = 100', 'Batch size =
title = ['Training loss', 'Validation loss']

plot_stuff(result_list, color_list, description_list, title)
```

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.