```python
# Do not import any additional 3rd party external libraries
from cProfile import label
from tkinter import W
from wsgiref.handlers import format_date_time
import numpy as np
import os
import matplotlib.pyplot as plt


class Activation(object):

    """
    Interface for activation functions (non-linearities).
    """

    # No additional work is needed for this class, as it acts like an abstract base class for the others

    def __init__(self):
        self.state = None

    def __call__(self, x):
        return self.forward(x)

    def forward(self, x):
        raise NotImplemented

    def derivative(self):
        raise NotImplemented


class Identity(Activation):

    """
    Identity function (already implemented).
    """

    # This class is a gimme as it is already implemented for you as an example (do not change)

    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        self.state = x
        return x

    def derivative(self):
        return 1.0


class Sigmoid(Activation):

    """
    Sigmoid non-linearity
    """
    def __init__(self):
        super(Sigmoid, self).__init__()

```

```python
    def forward(self, x):
        # hint: save the useful data for back propagation
        self.state = x
        return 1/(1+np.exp(-x))

    def derivative(self):
        return self.forward(self.state)*(1-self.forward(self.state))


class Tanh(Activation):

    """
    Tanh non-linearity
    """

    def __init__(self):
        super(Tanh, self).__init__()

    def forward(self, x):
        self.state = x
        return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))

    def derivative(self):
        return 1-np.power(self.forward(self.state),2)


class ReLU(Activation):

    """
    ReLU non-linearity
    """

    def __init__(self):
        super(ReLU, self).__init__()

    def forward(self, x):
        self.state = x
        out = np.maximum(0,x)
        return out

    def derivative(self):
        out = self.state
        out[out>0] = 1
        out[out<0] = 0
        return out



class Criterion(object):

    """
    Interface for loss functions.
    """

    # Nothing needs done to this class, it's used by the following Criterion classes

    def __init__(self):
        self.logits = None
        self.labels = None
```

```python
        self.loss = None

    def __call__(self, x, y):
        return self.forward(x, y)

    def forward(self, x, y):
        raise NotImplemented

    def derivative(self):
        raise NotImplemented


class SoftmaxCrossEntropy(Criterion):

    """
    Softmax loss
    """

    def __init__(self):
        super(SoftmaxCrossEntropy, self).__init__()
        # you can add variables if needed

    def softmax(self, x):
        ex = np.exp(x)
        return ex/np.sum(ex)

    def forward(self, x, y):
        self.logits = x
        self.labels = y
        out_shape = y.shape[0]
        soft = self.softmax(self.logits)
        self.loss = np.sum(-np.log(soft[range(out_shape),np.argmax(y,axis=1)]))/out_shape
        return self.loss

    def derivative(self):
        grad = self.softmax(self.logits)
        out_shape = self.labels
        out_shape = out_shape.shape[0]
        grad[range(out_shape),np.argmax(self.labels,axis=1)] -= 1
        return grad/out_shape


# randomly intialize the weight matrix with dimension d0 x d1 via Normal distribution
def random_normal_weight_init(d0, d1):
    return np.random.normal(size=(d0,d1))


# initialize a d-dimensional bias vector with all zeros
def zeros_bias_init(d):
    return np.zeros(d)


class MLP(object):

    """
    A simple multilayer perceptron
    (feel free to add class functions if needed)
    """
```

```python

    def __init__(self, input_size, output_size, hiddens, activations, weight_init_fn, bias_init_fn, criterion, lr):

        # Don't change this -->
        self.train_mode = True
        self.nlayers = len(hiddens) + 1
        self.input_size = input_size
        self.output_size = output_size
        self.activations = activations
        self.criterion = criterion
        self.lr = lr
        # <---------------------

        # Don't change the name of the following class attributes
        self.nn_dim = [input_size] + hiddens + [output_size]
        # list containing Weight matrices of each layer, each should be a np.array
        self.W = [weight_init_fn(self.nn_dim[i], self.nn_dim[i+1]) for i in range(self.nlayers)]
        # list containing derivative of Weight matrices of each layer, each should be a np.array
        self.dW = [np.zeros_like(weight) for weight in self.W]
        # list containing bias vector of each layer, each should be a np.array
        self.b = [bias_init_fn(self.nn_dim[i+1]) for i in range(self.nlayers)]
        # list containing derivative of bias vector of each layer, each should be a np.array
        self.db = [np.zeros_like(bias) for bias in self.b]

        # You can add more variables if needed
        self.f_data= np.zeros(output_size)
        self.Zn = [0]*len(self.W)
        self.An = [0]*len(self.W)
        self.input = 0

    def forward(self, x):
        out = x
        self.input = x
        for i,w in enumerate(self.W):
            out = out@w+self.b[i]
            self.Zn[i] = out
            out = self.activations[i](out)
            self.An[i] = out
        self.f_data = out


    def zero_grads(self):
        self.dW = np.zeros_like(self.dW)
        self.b = np.zeros_like(self.b)

    def step(self):
        self.W = [w - self.lr*dw for (w,dw) in zip(self.W,self.dW)]
        self.b = [b - self.lr*db for (b,db) in zip(self.b,self.db)]

    def backward(self, labels):
        if self.train_mode:
            # calculate dW and db only under training mode
            loss = self.get_loss(labels)
```

```python
            for l in range(self.nlayers-1,-1,-1):
                self.activations[l].state = self.Zn[l]
                if l!=(self.nlayers-1):
                    dl = self.activations[l].derivative()*(self.W[l+1]@dl.T).T
                else:
                    dl = (self.An[l]-labels)*self.activations[l].derivative()
                self.db[l] = np.mean(dl,axis=0,keepdims=False)
                if l!=0:
                    self.dW[l] = self.An[l-1].T @ dl
                else:
                    self.dW[l] = (self.input).T @ dl
        return

    def __call__(self, x):
        return self.forward(x)

    def train(self):
        # training mode
        self.train_mode = True

    def eval(self):
        # evaluation mode
        self.train_mode = False

    def get_loss(self, labels):
        # return the current loss value given labels
        return self.criterion(self.f_data,labels)

    def get_error(self, labels):
        # return the number of incorrect preidctions gievn labels
        pred = self.criterion.softmax(self.f_data)
        pred = np.argmax(pred,axis=1)
        labels_correct = np.argmax(labels,axis=1)
        return np.sum(pred!=labels_correct)

    def save_model(self, path='p1_model.npz'):
        # save the parameters of MLP (do not change)
        np.savez(path, self.W, self.b)


# Don't change this function
def get_training_stats(mlp, dset, nepochs, batch_size):
    train, val, test = dset
    trainx, trainy = train
    valx, valy = val
    testx, testy = test

    idxs = np.arange(len(trainx))

    training_losses = []
    training_errors = []
    validation_losses = []
    validation_errors = []

    for e in range(nepochs):
        print("epoch: ", e)
        train_loss = 0
        train_error = 0
        val_loss = 0
```

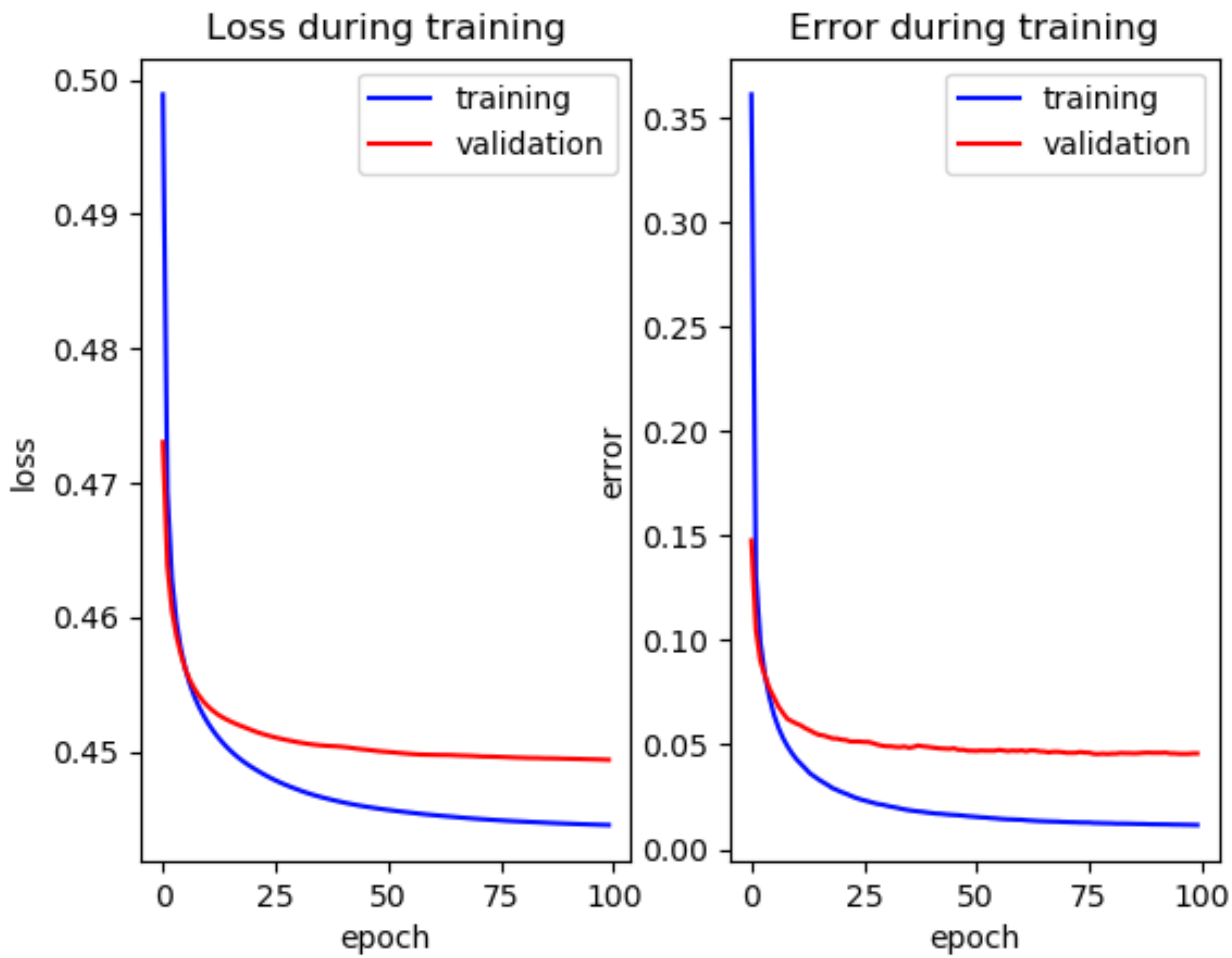
```
        val_error = 0
        num_train = len(trainx)
        num_val = len(valx)

        for b in range(0, num_train, batch_size):
            mlp.train()
            mlp(trainx[b:b+batch_size])
            mlp.backward(trainy[b:b+batch_size])
            mlp.step()
            train_loss += mlp.get_loss(trainy[b:b+batch_size])
            train_error += mlp.get_error(trainy[b:b+batch_size])
        training_losses += [train_loss/num_train]
        training_errors += [train_error/num_train]
        print("training loss: ", train_loss/num_train)
        print("training error: ", train_error/num_train)

        for b in range(0, num_val, batch_size):
            mlp.eval()
            mlp(valx[b:b+batch_size])
            val_loss += mlp.get_loss(valy[b:b+batch_size])
            val_error += mlp.get_error(valy[b:b+batch_size])
        validation_losses += [val_loss/num_val]
        validation_errors += [val_error/num_val]
        print("validation loss: ", val_loss/num_val)
        print("validation error: ", val_error/num_val)

    test_loss = 0
    test_error = 0
    num_test = len(testx)
    for b in range(0, num_test, batch_size):
        mlp.eval()
        mlp(testx[b:b+batch_size])
        test_loss += mlp.get_loss(testy[b:b+batch_size])
        test_error += mlp.get_error(testy[b:b+batch_size])
    test_loss /= num_test
    test_error /= num_test
    print("test loss: ", test_loss)
    print("test error: ", test_error)

    return (training_losses, training_errors, validation_losses, validation_errors)


# get ont hot key encoding of the label (no need to change this function)
def get_one_hot(in_array, one_hot_dim):
    dim = in_array.shape[0]
    out_array = np.zeros((dim, one_hot_dim))
    for i in range(dim):
        idx = int(in_array[i])
        out_array[i, idx] = 1
    return out_array


def main():
    # load the mnist dataset from csv files
    image_size = 28 # width and length of mnist image
    num_labels = 10 #  i.e. 0, 1, 2, 3, ..., 9
    image_pixels = image_size * image_size
    data_path = "mnist/"
    train_data = np.loadtxt(data_path + "mnist_train.csv", delimiter=",")
```

```
    test_data = np.loadtxt(data_path + "mnist_test.csv", delimiter=",")

    # rescale image from 0-255 to 0-1
    fac = 1.0 / 255
    train_imgs = np.asfarray(train_data[:50000, 1:]) * fac
    val_imgs = np.asfarray(train_data[50000:, 1:]) * fac
    test_imgs = np.asfarray(test_data[:, 1:]) * fac
    train_labels = np.asfarray(train_data[:50000, :1])
    val_labels = np.asfarray(train_data[50000:, :1])
    test_labels = np.asfarray(test_data[:, :1])

    # convert labels to one-hot-key encoding
    train_labels = get_one_hot(train_labels, num_labels)
    val_labels = get_one_hot(val_labels, num_labels)
    test_labels = get_one_hot(test_labels, num_labels)

    print(train_imgs.shape)
    print(train_labels.shape)
    print(val_imgs.shape)
    print(val_labels.shape)
    print(test_imgs.shape)
    print(test_labels.shape)

    dataset = [
        [train_imgs, train_labels],
        [val_imgs, val_labels],
        [test_imgs, test_labels]
    ]

    # These are only examples of parameters you can start with
    # you can tune these parameters to improve the performance of your MLP
    # this is the only part you need to change in main() function
    hiddens = [128, 64]
    activations = [Sigmoid(), Sigmoid(), Sigmoid()]
    lr = 0.05
    num_epochs = 100
    batch_size = 8

    # build your MLP model
    mlp = MLP(
        input_size=image_pixels,
        output_size=num_labels,
        hiddens=hiddens,
        activations=activations,
        weight_init_fn=random_normal_weight_init,
        bias_init_fn=zeros_bias_init,
        criterion=SoftmaxCrossEntropy(),
        lr=lr
    )

    # train the neural network
    losses = get_training_stats(mlp, dataset, num_epochs, batch_size)

    # save the parameters
    mlp.save_model()

    # visualize the training and validation loss with epochs
    training_losses, training_errors, validation_losses, validation_errors =
losses

```

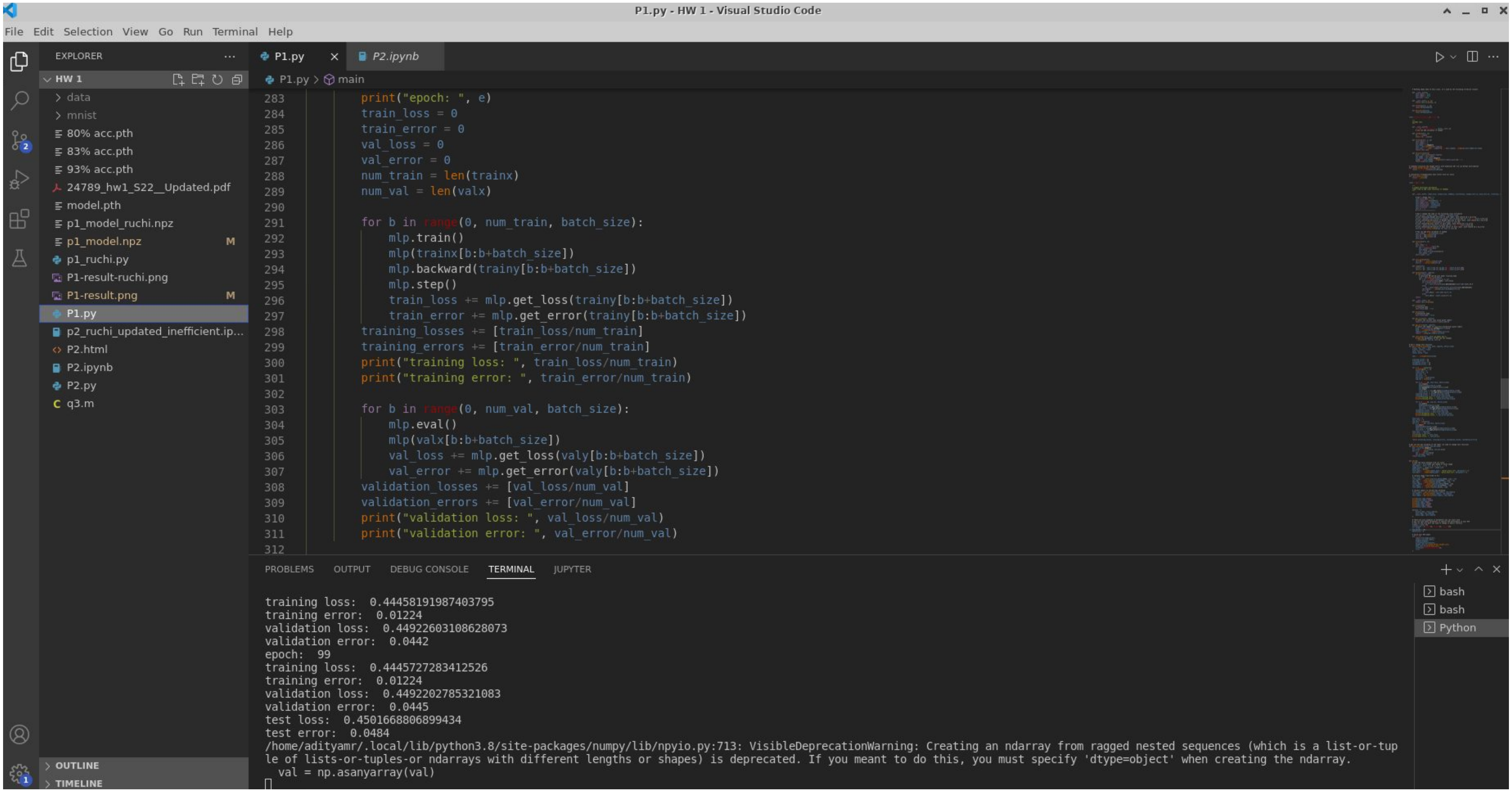
```
    fig, (ax1, ax2) = plt.subplots(1, 2)

    ax1.plot(training_losses, color='blue', label="training")
    ax1.plot(validation_losses, color='red', label='validation')
    ax1.set_title('Loss during training')
    ax1.set_xlabel('epoch')
    ax1.set_ylabel('loss')
    ax1.legend()

    ax2.plot(training_errors, color='blue', label="training")
    ax2.plot(validation_errors, color='red', label="validation")
    ax2.set_title('Error during training')
    ax2.set_xlabel('epoch')
    ax2.set_ylabel('error')
    ax2.legend()

    plt.show()


if __name__ == "__main__":
    main()
```

After training the model I got the following results:
test loss: 0.4501668806899434
test error: 0.0484

Hyperparameter tuning that I carried out to tune the model:
I Initially started with a model with only one hidden layer to test out if the accuracy could still be achieved but that did not work out. Then I reverted back to the starting model with two hidden layers. I slowly increased the epochs as I was getting increasing accuracy with increase in epochs. I managed to get good test results (<5% test error) for 100 epochs and hence I did not tweak other parameters such as learning rate or batch size to further try and tune the model.



Loss during training
training
validation
0.50
0.49
0.48
0.47
0.46
0.45
loss
0
25
50
75
100
epoch
Error during training
training
validation
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
error
0
25
50
75
100
epoch


P1.py - HW 1 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
HW 1
data
mnist
80% acc.pth
83% acc.pth
93% acc.pth
24789_hw1_S22__Updated.pdf
model.pth
p1_model_ruchi.npz
p1_model.npz M
p1_ruchi.py
P1-result-ruchi.png
P1-result.png M
P1.py
p2_ruchi_updated_inefficient.ip...
P2.html
P2.ipynb
P2.py
q3.m
OUTLINE
TIMELINE
P1.py
P2.ipynb
P1.py > main
        print("epoch: ", e)
        train_loss = 0
        train_error = 0
        val_loss = 0
        val_error = 0
        num_train = len(trainx)
        num_val = len(valx)

        for b in range(0, num_train, batch_size):
            mlp.train()
            mlp(trainx[b:b+batch_size])
            mlp.backward(trainy[b:b+batch_size])
            mlp.step()
            train_loss += mlp.get_loss(trainy[b:b+batch_size])
            train_error += mlp.get_error(trainy[b:b+batch_size])
        training_losses += [train_loss/num_train]
        training_errors += [train_error/num_train]
        print("training loss: ", train_loss/num_train)
        print("training error: ", train_error/num_train)

        for b in range(0, num_val, batch_size):
            mlp.eval()
            mlp(valx[b:b+batch_size])
            val_loss += mlp.get_loss(valy[b:b+batch_size])
            val_error += mlp.get_error(valy[b:b+batch_size])
        validation_losses += [val_loss/num_val]
        validation_errors += [val_error/num_val]
        print("validation loss: ", val_loss/num_val)
        print("validation error: ", val_error/num_val)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
training loss: 0.44458191987403795
training error: 0.01224
validation loss: 0.44922603108628073
validation error: 0.0442
epoch: 99
training loss: 0.4445727283412526
training error: 0.01224
validation loss: 0.4492202785321083
validation error: 0.0445
test loss: 0.45016688806899434
test error: 0.0484
/home/adityamr/.local/lib/python3.8/site-packages/numpy/lib/npyio.py:713: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  val = np.asanyarray(val)
bash
bash
Python