Lab 2 Task 3 Joel willén - joewil-2 Deborah Aittoklllio debait-2

```
import numpy as np
import sys

#functions of non-linear activations
def f_sigmoid(X, deriv=False):
    if not deriv:
        return 1 / (1 + np.exp(-X))
    else:
        return f_sigmoid(X)*(1 - f_sigmoid(X))


def f_softmax(X):
    Z = np.sum(np.exp(X), axis=1)
    Z = Z.reshape(Z.shape[0], 1)
    return np.exp(X) / Z

def f_relu(X, deriv=False):
    if not deriv:
        return np.maximum(0, X)
    else:
        # derivative: 1 if X > 0, else 0
        return (X > 0).astype(float)

def exit_with_err(err_str):
    print(sys.stderr, err_str)
    sys.exit(1)

#Functionality of a single hidden layer
class Layer:
    def __init__(self, size, batch_size, is_input=False,
is_output=False,
                 activation=f_sigmoid):
        self.is_input = is_input
        self.is_output = is_output

        # Z is the matrix that holds output values
        self.Z = np.zeros((batch_size, size[0]))
        # The activation function is an externally defined function
(with a
        # derivative) that is stored here
        self.activation = activation

        # W is the outgoing weight matrix for this layer
        self.W = None
        # S is the matrix that holds the inputs to this layer
        self.S = None
        # D is the matrix that holds the deltas for this layer
        self.D = None
        # Fp is the matrix that holds the derivatives of the
activation function
        self.Fp = None

        if not is_input:
            self.S = np.zeros((batch_size, size[0]))
            self.D = np.zeros((batch_size, size[0]))

        if not is_output:
            self.W = np.random.normal(size=size, scale=1E-4)

        if not is_input and not is_output:
            self.Fp = np.zeros((size[0], batch_size))

    def forward_propagate(self):
        if self.is_input:
            return self.Z.dot(self.W)

        self.Z = self.activation(self.S)
        if self.is_output:
            return self.Z
        else:
            # For hidden layers, we add the bias values here
            self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)),
axis=1)
            self.Fp = self.activation(self.S, deriv=True).T
            return self.Z.dot(self.W)

class MultiLayerPerceptron:

    def __init__(self, layer_config, batch_size=100,
hidden_activation=f_sigmoid):
        self.layers = []
        self.num_layers = len(layer_config)
        self.minibatch_size = batch_size

        for i in range(self.num_layers-1):
            if i == 0:
                print ("Initializing input layer with size
{0}.".format(layer_config[i]))
                # Input layer (we add one unit for bias in Z, but no
activation here)
                self.layers.append(Layer([layer_config[i]+1,
layer_config[i+1]],
                                         batch_size,
                                         is_input=True))
            else:
                print ("Initializing hidden layer with size
{0}.".format(layer_config[i]))
                # Hidden layers use the chosen activation (sigmoid or
ReLU)
                self.layers.append(Layer([layer_config[i]+1,
layer_config[i+1]],
                                         batch_size,

activation=hidden_activation))

        print ("Initializing output layer with size
{0}.".format(layer_config[-1]))
        # Output layer still uses softmax
        self.layers.append(Layer([layer_config[-1], None],
                                 batch_size,
                                 is_output=True,
                                 activation=f_softmax))
        print ("Done!")

    def forward_propagate(self, data):
        #We need to be sure to add bias values to the input
        self.layers[0].Z = np.append(data, np.ones((data.shape[0],
1)), axis=1)

        for i in range(self.num_layers-1):
            self.layers[i+1].S = self.layers[i].forward_propagate()
        return self.layers[-1].forward_propagate()

    def backpropagate(self, yhat, labels):

        #exit_with_err("FIND ME IN THE CODE, What is computed in the
next line of code?\n")
        #This is the where delta (yhat-labels) is calculated. The
start of backpropagation

        self.layers[-1].D = (yhat - labels).T
        for i in range(self.num_layers-2, 0, -1):
            # We do not calculate deltas for the bias values
            W_nobias = self.layers[i].W[0:-1, :]

            #exit_with_err("FIND ME IN THE CODE, What does this 'for'
loop do?\n")
            #This loop iterates through each hidden layer and
implements the backpropogation step for the layer


            self.layers[i].D = W_nobias.dot(self.layers[i+1].D) *
self.layers[i].Fp #This being the backpropogation formula

    def update_weights(self, eta):
        for i in range(0, self.num_layers-1):
            W_grad = -eta*(self.layers[i+1].D.dot(self.layers[i].Z)).T
            self.layers[i].W += W_grad
    def evaluate(self, train_data, train_labels, test_data, 
test_labels,
                 num_epochs=70, eta=0.05, eval_train=False, 
eval_test=True):

        N_train = len(train_labels)*len(train_labels[0])
        N_test = len(test_labels)*len(test_labels[0])

        print ("Training for {0} epochs...".format(num_epochs))
        for t in range(0, num_epochs):
            out_str = "[{0:4d}] ".format(t)

            for b_data, b_labels in zip(train_data, train_labels):
                output = self.forward_propagate(b_data)
                self.backpropagate(output, b_labels)

                #exit_with_err("FIND ME IN THE CODE, How does weight 
update is implemented? What is eta?\n")
                #eta is the learning rate and the weight update is 
implemented with the update weights function which contains the 
formula for gradient descent

                self.update_weights(eta=eta)

            if eval_train:
                errs = 0
                for b_data, b_labels in zip(train_data, train_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-
b_labels[np.arange(len(b_labels)), yhat])

                out_str = ("{0} Training error: 
{1:.5f}".format(out_str,

float(errs)/N_train))

            if eval_test:
                errs = 0
                for b_data, b_labels in zip(test_data, test_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-
b_labels[np.arange(len(b_labels)), yhat])

                out_str = ("{0} Test error: {1:.5f}").format(out_str,

float(errs)/N_test)
```

```
            print (out_str)

def label_to_bit_vector(labels, nbits):
    bit_vector = np.zeros((labels.shape[0], nbits))
    for i in range(labels.shape[0]):
        bit_vector[i, labels[i]] = 1.0

    return bit_vector

def create_batches(data, labels, batch_size, create_bit_vector=False):
    N = data.shape[0]
    print ("Batch size {0}, the number of examples
{1}.".format(batch_size,N))

    if N % batch_size != 0:
        print ("Warning in create_minibatches(): Batch size {0} does
not " \
              "evenly divide the number of examples
{1}.".format(batch_size,N))
    chunked_data = []
    chunked_labels = []
    idx = 0
    while idx + batch_size <= N:
        chunked_data.append(data[idx:idx+batch_size, :])
        if not create_bit_vector:
            chunked_labels.append(labels[idx:idx+batch_size])
        else:
            bit_vector =
label_to_bit_vector(labels[idx:idx+batch_size], 10)
            chunked_labels.append(bit_vector)

        idx += batch_size

    return chunked_data, chunked_labels

def prepare_for_backprop(batch_size, Train_images, Train_labels,
Valid_images, Valid_labels):

    print ("Creating data...")
    batched_train_data, batched_train_labels =
create_batches(Train_images, Train_labels,
                                              batch_size,
                                              create_bit_vector=True)
    batched_valid_data, batched_valid_labels =
create_batches(Valid_images, Valid_labels,
                                              batch_size,
                                              create_bit_vector=True)
    print ("Done!")
    return batched_train_data, batched_train_labels,
batched_valid_data, batched_valid_labels


from keras.datasets import mnist

(Xtr, Ltr), (X_test, L_test)=mnist.load_data()

Xtr = Xtr.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
Xtr = Xtr.astype('float32')
X_test = X_test.astype('float32')
Xtr /= 255
X_test /= 255
print(Xtr.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

60000 train samples
10000 test samples
```

1.a) The principle of the backpropogation algorithm; The backpropogation can be explained by three components that make use of the chain rule. The first being the forward propogation which stores the activations, pre activations and the derivative of activations. The Backpropogation component which is the calculation of the delta at the output layer and then iterates backwards over each hidden layers and computes their deltas using that output delta and then lastly the weight update which updates the weights using gradiant descent.

1.b) Softmax meaning is to take a vector of scores or logits and turn these to probability distribution over classes. Which means all outputs sum to 1. As such its well suited for multiclass classification tasks. In the given MLP code its role is to take logits and turn them into probabilities over classes. ANd even more so it is part of the cross entropy loss used.

1.c) In this code softmax is used on the output to get multi class classification and sigmoid activation functions in the hidden layers which add nonlinearity. Other examples are a linear output or Relu and Mean square error for regression. Which ones you choose to use in the code is very tied to the problem itself and choosing the correct activation functions for the task at hand is vital for the network to be able to suceed.

1.d) Comments have been added next to the raised errors to answer there questions. See the MLP class.

```
batch_size=100;

train_data, train_labels, valid_data,
valid_labels=prepare_for_backprop(batch_size, Xtr, Ltr, X_test,
L_test)

mlp = MultiLayerPerceptron(layer_config=[784, 100, 100, 10],
batch_size=batch_size)
```

```python
mlp.evaluate(train_data, train_labels, valid_data, valid_labels,
             eval_train=True)

#Training accuracy
train_correct = 0
train_total = 0
for b_data, b_labels in zip(train_data, train_labels):
    output = mlp.forward_propagate(b_data)
    yhat = np.argmax(output, axis=1)
    ytrue = np.argmax(b_labels, axis=1)
    train_correct += np.sum(yhat == ytrue)
    train_total += len(ytrue)
train_accuracy = train_correct / train_total

#Validation accuracy
valid_correct = 0
valid_total = 0
for b_data, b_labels in zip(valid_data, valid_labels):
    output = mlp.forward_propagate(b_data)
    yhat = np.argmax(output, axis=1)
    ytrue = np.argmax(b_labels, axis=1)
    valid_correct += np.sum(yhat == ytrue)
    valid_total += len(ytrue)
valid_accuracy = valid_correct / valid_total

print("\nFINAL ACCURACIES:")
print(f"Training accuracy:   {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {valid_accuracy*100:.2f}%")


print("Done:)\n")
```

```
Creating data...
Batch size 100, the number of examples 60000.
Batch size 100, the number of examples 10000.
Done!
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
Training for 70 epochs...
[   0]  Training error: 0.45165 Test error: 0.45930
[   1]  Training error: 0.07577 Test error: 0.07490
[   2]  Training error: 0.07840 Test error: 0.07970
[   3]  Training error: 0.04012 Test error: 0.04340
[   4]  Training error: 0.03302 Test error: 0.03950
[   5]  Training error: 0.03097 Test error: 0.03980
[   6]  Training error: 0.02888 Test error: 0.03890
[   7]  Training error: 0.02595 Test error: 0.03770
[    8]  Training error: 0.02448 Test error: 0.03580
[    9]  Training error: 0.02137 Test error: 0.03400
[   10]  Training error: 0.02015 Test error: 0.03330
[   11]  Training error: 0.01828 Test error: 0.03340
[   12]  Training error: 0.01862 Test error: 0.03270
[   13]  Training error: 0.01652 Test error: 0.03160
[   14]  Training error: 0.01535 Test error: 0.03230
[   15]  Training error: 0.01865 Test error: 0.03260
[   16]  Training error: 0.01767 Test error: 0.03460
[   17]  Training error: 0.01153 Test error: 0.03340
[   18]  Training error: 0.01383 Test error: 0.03290
[   19]  Training error: 0.01013 Test error: 0.03010
[   20]  Training error: 0.00890 Test error: 0.03030
[   21]  Training error: 0.01597 Test error: 0.03430
[   22]  Training error: 0.01552 Test error: 0.03510
[   23]  Training error: 0.01143 Test error: 0.03360
[   24]  Training error: 0.00888 Test error: 0.03120
[   25]  Training error: 0.00690 Test error: 0.02890
[   26]  Training error: 0.00632 Test error: 0.02870
[   27]  Training error: 0.00717 Test error: 0.02960
[   28]  Training error: 0.00660 Test error: 0.02890
[   29]  Training error: 0.00648 Test error: 0.02850
[   30]  Training error: 0.01148 Test error: 0.03280
[   31]  Training error: 0.01110 Test error: 0.03260
[   32]  Training error: 0.00697 Test error: 0.02990
[   33]  Training error: 0.00810 Test error: 0.03110
[   34]  Training error: 0.00825 Test error: 0.03230
[   35]  Training error: 0.00680 Test error: 0.03030
[   36]  Training error: 0.00572 Test error: 0.02980
[   37]  Training error: 0.00540 Test error: 0.02850
[   38]  Training error: 0.00357 Test error: 0.02840
[   39]  Training error: 0.00253 Test error: 0.02800
[   40]  Training error: 0.00227 Test error: 0.02750
[   41]  Training error: 0.00323 Test error: 0.02730
[   42]  Training error: 0.00165 Test error: 0.02660
[   43]  Training error: 0.00185 Test error: 0.02700
[   44]  Training error: 0.00870 Test error: 0.03140
[   45]  Training error: 0.00673 Test error: 0.03000
[   46]  Training error: 0.00790 Test error: 0.03030
[   47]  Training error: 0.00495 Test error: 0.02940
[   48]  Training error: 0.01010 Test error: 0.03170
[   49]  Training error: 0.00328 Test error: 0.02940
[   50]  Training error: 0.00272 Test error: 0.02800
[   51]  Training error: 0.00220 Test error: 0.02850
[   52]  Training error: 0.00207 Test error: 0.02880
[   53]  Training error: 0.00385 Test error: 0.03000
[   54]  Training error: 0.00143 Test error: 0.02830
[   55]  Training error: 0.00122 Test error: 0.02710
[   56]  Training error: 0.00267 Test error: 0.02840
[  57]  Training error: 0.00063 Test error: 0.02630
[  58]  Training error: 0.00123 Test error: 0.02760
[  59]  Training error: 0.00247 Test error: 0.02900
[  60]  Training error: 0.00208 Test error: 0.02800
[  61]  Training error: 0.00060 Test error: 0.02610
[  62]  Training error: 0.00087 Test error: 0.02800
[  63]  Training error: 0.00052 Test error: 0.02660
[  64]  Training error: 0.00010 Test error: 0.02670
[  65]  Training error: 0.00002 Test error: 0.02630
[  66]  Training error: 0.00002 Test error: 0.02650
[  67]  Training error: 0.00000 Test error: 0.02650
[  68]  Training error: 0.00000 Test error: 0.02670
[  69]  Training error: 0.00000 Test error: 0.02630

FINAL ACCURACIES:
Training accuracy:   100.00%
Validation accuracy: 97.37%
Done:)
```

Part 2. We get a training accuracy of 100% and validation accuracy of 97.32%. This took 3 minutes.

Part 3. For an lr of 0.005 we get a Training accuracy of 99.97% and Validation of 97.31%. This took 6 minutes. Lastly for an lr of 0.5 we get a Training accuracy of 9.75% and a validation accuracy of 9.74%. This took 6 minutes.

What we can extrapolate from these results is that an lr of 0.005 is able to get the same results as lr 0.05 but with double the time. This is because the weight updates are considerably smaller making convergence slower but with time is able to get almost the same accuracy as our best case. In the case of lr of 0.5 we can see that the MLP "collpases" as the optimal values for weights are overshot at each step and our gradents vanish. This leaves the accuracy up to essentially random guess. Giving us the very bad accuracy.

Part 4. With the same parameters as the best case previously, we get the same problem the lr of 0.5 got earlier. Around 10%. As such it is clear we must lower the lr to get any meaningful result. As such we lowered it by a considerable margin to 0.005. Which turned out to be the lucky number as it gave us the same accuracy value as that of the best case of Sigmoid,

```
batch_size = 100
train_data, train_labels, valid_data, valid_labels =
prepare_for_backprop(batch_size, Xtr, Ltr, X_test, L_test)

#ReLU MLP
mlp_relu = MultiLayerPerceptron(layer_config=[784, 100, 100,
10],batch_size=batch_size,hidden_activation=f_relu)

mlp_relu.evaluate(train_data, train_labels, valid_data,
valid_labels,num_epochs=70, eta=0.005, eval_train=True)
```

```python
#Training accuracy
train_correct = 0
train_total = 0
for b_data, b_labels in zip(train_data, train_labels):
    output = mlp_relu.forward_propagate(b_data)
    yhat = np.argmax(output, axis=1)
    ytrue = np.argmax(b_labels, axis=1)
    train_correct += np.sum(yhat == ytrue)
    train_total += len(ytrue)
train_accuracy = train_correct / train_total

#Validation accuracy
valid_correct = 0
valid_total = 0
for b_data, b_labels in zip(valid_data, valid_labels):
    output = mlp_relu.forward_propagate(b_data)
    yhat = np.argmax(output, axis=1)
    ytrue = np.argmax(b_labels, axis=1)
    valid_correct += np.sum(yhat == ytrue)
    valid_total += len(ytrue)
valid_accuracy = valid_correct / valid_total

print("\nReLU FINAL ACCURACIES:")
print(f"Training accuracy:   {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {valid_accuracy*100:.2f}%")

print("Done:)\n")
```

```
Creating data...
Batch size 100, the number of examples 60000.
Batch size 100, the number of examples 10000.
Done!
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
Training for 70 epochs...
[    0]  Training error: 0.89558 Test error: 0.89720
[    1]  Training error: 0.89558 Test error: 0.89720
[    2]  Training error: 0.89558 Test error: 0.89720
[    3]  Training error: 0.89558 Test error: 0.89720
[    4]  Training error: 0.89558 Test error: 0.89720
[    5]  Training error: 0.89558 Test error: 0.89720
[    6]  Training error: 0.89558 Test error: 0.89720
[    7]  Training error: 0.89558 Test error: 0.89720
[    8]  Training error: 0.89558 Test error: 0.89720
[    9]  Training error: 0.89558 Test error: 0.89720
[   10]  Training error: 0.89558 Test error: 0.89720
[   11]  Training error: 0.89558 Test error: 0.89720
[  12]  Training error: 0.89558 Test error: 0.89720
[  13]  Training error: 0.89558 Test error: 0.89720
[  14]  Training error: 0.89558 Test error: 0.89720
[  15]  Training error: 0.89558 Test error: 0.89720
[  16]  Training error: 0.78302 Test error: 0.78710
[  17]  Training error: 0.07975 Test error: 0.07950
[  18]  Training error: 0.04952 Test error: 0.05190
[  19]  Training error: 0.04667 Test error: 0.05300
[  20]  Training error: 0.03177 Test error: 0.03640
[  21]  Training error: 0.02875 Test error: 0.03700
[  22]  Training error: 0.02545 Test error: 0.03670
[  23]  Training error: 0.02288 Test error: 0.03480
[  24]  Training error: 0.02258 Test error: 0.03590
[  25]  Training error: 0.01877 Test error: 0.03400
[  26]  Training error: 0.01767 Test error: 0.03400
[  27]  Training error: 0.01547 Test error: 0.03330
[  28]  Training error: 0.01965 Test error: 0.03660
[  29]  Training error: 0.01462 Test error: 0.03180
[  30]  Training error: 0.01472 Test error: 0.03320
[  31]  Training error: 0.01065 Test error: 0.02900
[  32]  Training error: 0.01708 Test error: 0.03490
[  33]  Training error: 0.01170 Test error: 0.03020
[  34]  Training error: 0.01397 Test error: 0.03230
[  35]  Training error: 0.01000 Test error: 0.02940
[  36]  Training error: 0.01208 Test error: 0.03240
[  37]  Training error: 0.00727 Test error: 0.02860
[  38]  Training error: 0.00577 Test error: 0.02680
[  39]  Training error: 0.01133 Test error: 0.03050
[  40]  Training error: 0.00383 Test error: 0.02480
[  41]  Training error: 0.00950 Test error: 0.03020
[  42]  Training error: 0.00480 Test error: 0.02780
[  43]  Training error: 0.00968 Test error: 0.02960
[  44]  Training error: 0.00520 Test error: 0.02650
[  45]  Training error: 0.00583 Test error: 0.02810
[  46]  Training error: 0.00848 Test error: 0.02950
[  47]  Training error: 0.00458 Test error: 0.02740
[  48]  Training error: 0.00402 Test error: 0.02540
[  49]  Training error: 0.00503 Test error: 0.02880
[  50]  Training error: 0.00773 Test error: 0.03140
[  51]  Training error: 0.00172 Test error: 0.02390
[  52]  Training error: 0.00350 Test error: 0.02550
[  53]  Training error: 0.00245 Test error: 0.02520
[  54]  Training error: 0.00143 Test error: 0.02460
[  55]  Training error: 0.00068 Test error: 0.02380
[  56]  Training error: 0.00048 Test error: 0.02370
[  57]  Training error: 0.00020 Test error: 0.02320
[  58]  Training error: 0.00015 Test error: 0.02250
[  59]  Training error: 0.00008 Test error: 0.02240
[  60]  Training error: 0.00000 Test error: 0.02220
[  61]  Training error: 0.00005 Test error: 0.02220
[  62]  Training error: 0.00003 Test error: 0.02200
[  63]  Training error: 0.00000 Test error: 0.02180
[  64]  Training error: 0.00000 Test error: 0.02160
[  65]  Training error: 0.00000 Test error: 0.02160
[  66]  Training error: 0.00000 Test error: 0.02150
[  67]  Training error: 0.00000 Test error: 0.02140
[  68]  Training error: 0.00000 Test error: 0.02160
[  69]  Training error: 0.00000 Test error: 0.02150

ReLU FINAL ACCURACIES:
Training accuracy:   100.00%
Validation accuracy: 97.85%
Done:)
```