Deep Learning: Assignment 1

Sebastian Aguerre (2695712)

November 2024

1 Local Derivatives

1.1 Question 1

To train a neural network we are required to use the gradient in order to propagate the error and update its weights. Therefore, here we will derive the local derivatives of the loss function of a network as well as the derivate of the output activation function. The activation function used is the**softmax function** which is given by:

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}.} \tag{1}$$

The loss used for a classification task is the **cross-entropy loss**, which for a given instance x is given by:

$$l = -\sum \log y_i \tag{2}$$

where y_i is the probability of class i, and we let c be the correct class such that y_c is the probability attributed to the correct class.

Derivative of cross-entropy Loss with y_i (2). For this we simply take the derivative of the term within the summation, that is of $\log y_i$:

$$\frac{\partial}{\partial y_i} = \frac{1}{y_i}.$$

Since we only care about the derivative with respect to the correct class, that is $i = c$, we can easily get the derivatives for all terms within the summation:

$$\frac{\partial}{\partial y_i}(-\log y_i) = \begin{cases} -\frac{1}{y_i} & \text{if } i = c \\ 0 & \text{if } i \neq c \end{cases}$$

Where the derivative for $i \neq c$ is simply zero because the term is considered a constant. We now add up all the zeros and distribute the minus sign to get that the derivative of the Loss with respect to the y_i is:

$$\frac{\partial l}{\partial y_i} = -\frac{1}{y_c}$$

Derivative of softmax output y_i with respect to linear combination O_i For this derivation, we first acknowledge that $\frac{d}{dx}e^x = e^x$, following from that we can we can use the quotient rule to derive the final expression for the derivative of (1) which has two cases $i = j$ and $i \neq j$. Which represent the probability of the correct class (true target) and the probability for all other classes For $i = j$,

$$\frac{\partial y_i}{\partial O_i} = \frac{\left(\sum_k e^{O_i} \cdot \frac{\partial e^{O_i}}{\partial O_i}\right) - \left(e^{O_i} \cdot \frac{\partial}{\partial O_i} \sum_k e^{O_k}\right)}{(\sum_k e^{O_k})^2} \tag{3}$$

Which simplifies to

$$\frac{e^{O_i} \cdot (\sum_k e^{O_k} - e^{O_i})}{(\sum_k e^{O_k})^2} = y_i \cdot \frac{(\sum_k e^{O_k} - e^{O_i})}{\sum_k e^{O_k})}.$$

Hence we get the final expression

$$y_i(1 - y_i).$$

Now we derive the expression for $i \neq j$. we note first that:

$$\sum_k e^{O_k} = e^{O_j} \text{ and } \frac{\partial e^{o_i}}{\partial O_j} = 0$$

now using the information we plug it into (3) and we get:

$$= \frac{0 - (e^{O_i} \cdot e^{O_j})}{(\sum_k e^{O_k})^2}$$

which simplifies to:

$$= -y_i \cdot y_j.$$

Therefore, we get that the derivative of the softmax output with respect to the linear combination is:

$$\frac{\partial y_i}{\partial O_i} = \begin{cases} y_i(1 - y_i) & \text{if } i = j \\ -y_i y_j & \text{if } i \neq j \end{cases}$$

Question 2 The derivative of cross-entropy loss with respect to O_i can be found using the **chain rule**. Since we have already computed $\frac{\partial L}{\partial y_i}$ (1.1) and $\frac{\partial y_i}{\partial O_i}$ (1.1) we simply plug in the expression by cases into:

$$\frac{\partial l}{\partial o_i} = \frac{\partial l}{\partial y_i} \cdot \frac{\partial y_i}{\partial o_i}.$$

For the case of $i = c$ we get: For the true class c:

$$\frac{\partial l}{\partial o_c} = \frac{\partial l}{\partial y_c} \cdot \frac{\partial y_c}{\partial o_c} = -\frac{1}{y_c} \cdot y_c(1 - y_c) = y_c - 1$$

and for the case of $i \neq c$, which by extension means $i \neq j$:

$$\frac{\partial l}{\partial o_j} = \frac{\partial l}{\partial y_j} \cdot \frac{\partial y_j}{\partial o_j} = 0 \cdot (-y_j y_c) = y_j$$

Therefore, we put it all together to get that the gradient of the loss concerning each logit o_i is:

$$\frac{\partial l}{\partial o_i} = \begin{cases} y_i - 1 & \text{if } i = c \\ y_i & \text{if } i \neq c \end{cases}.$$

We do not need this expression for our backpropagation algorithm since the backpropagation algorithm works with local derivatives and not with global derivatives. Therefore we can simply use the derivatives of $\frac{\partial L}{\partial y_i}$ and $\frac{\partial y_i}{\partial O_i}$ to get to $\frac{\partial L}{\partial O_i}$.

2 Scalar backpropagation

Question 3 The above code contain the implementation of the desired neural network using pure python:

```python
import math as m

# support functions
def sigmoid(values: list):
    return [1/(1 + m.exp(-x)) for x in values]

def softmax(values: list):
    exp_values = [m.exp(value) for value in values]
    exp_sum = sum(exp_values)

    return [(x/exp_sum) for x in exp_values]

def weighted_sum(input: list, weights:list, bias: list):
    # initialize empty output list
    output = [0] * len(weights[0])

    # loop over number of weights per input (number of output)
    for i in range(len(weights[0])):
        # loop over number of input
        for j in range(len(input)):
            output[i] += input[j]*weights[j][i]
        output[i] += bias[i]

    return output


def predict(smax_output, target):
    return target[smax_output.index(max(smax_output))]

def forwardpass(input, w1, b1, w2, b2):
    # feedforward pass
    o1 = weighted_sum(input, w1, b1) # first linear combination
    h = sigmoid(o1) # sigmoid activation function
    o2 = weighted_sum(h, w2, b2) # second linear combination
    y = softmax(o2) # sigmoid activation function

    return o1, h, o2, y

def elemwise_mult(values1: list, values2: list):
    return [x * y for x, y in zip(values1, values2)]
```

```
def deriv_loss(values: list, target_idx):
    return [-1/x if values.index(x) == target_idx else 0 for x in values]

def deriv_sigmoid(values: list):
    sigmoid_vals = sigmoid(values)
    return [x*(1 - x) for x in sigmoid_vals]

def deriv_softmax(values: list, target_idx):
    deriv = [0] * len(values)
    for idx, value in enumerate(values):
        if idx == target_idx:
            deriv[idx] = value*(1 - value)
        else:
            deriv[idx] = (-1)*values[target_idx]*value

    return deriv

def get_loss(prediction, target):
    return - m.log(prediction[0]) if target[0] == 1 else -m.log(prediction[1])

def backprop(input, w1, b1, w2, b2, target, o1, h, o2, y):
    # calculate the loss
    loss = get_loss(y, target)
    target_idx = target.index(1)

    # backwards pass to compute derivatives
        # dL_dy = deriv_loss(y, target_idx) # (2, 0)
        # dL_do2 = elemwise_mult(dL_dy, deriv_softmax(y, target_idx)) # (2, )
    dL_do2 = [y[i] - target[i] for i in range(len(y))]
    dL_dw2 = [[x*y for x in dL_do2] for y in h] # (3, 2)
    dL_db2 = dL_do2
    dL_dh = [dL_do2[i]*w2[0][i] for i in range(len(dL_do2))]
    for i in range(len(dL_do2)):
        dL_dh[i] += dL_do2[i]*w2[1][i]
    dL_do1 = [dL_dh[i]* d_sig for d_sig in deriv_sigmoid(o1)]
    dL_dw1 = [[dL_do1[i]*input[j] for i in range(len(dL_do1))] for j in range(len(input))]
    dL_db1 = dL_do1

    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def forward_and_backward(input, w1, b1, w2, b2, target):
    # feedforward pass
    o1, h, o2, y = forwardpass(input, w1, b1, w2, b2)

    # backwards pass and get derivatives
    dL_dw1, dL_db1, dL_dw2, dL_db2, loss = backprop(input, w1, b1, w2, b2, target, o1, h, o
    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def gradient_desent(w1, b1, w2, b2, dL_dw1, dL_db1, dL_dw2, dL_db2, alpha):
    # update weights, biases in first layer
    for i in range(len(w1)):
        for idx, w in enumerate(w1[i]):
            w1[i][idx] = w - alpha*dL_dw1[i][idx]

    for i in range(len(b1)):
        b1[i] =  b1[i] - alpha*dL_db1[i]

    # update weights, biases in second layer
    for i in range(len(w2)):
        for idx, w in enumerate(w2[i]):
            w2[i][idx] = w - alpha*dL_dw2[i][idx]

    for i in range(len(b2)):
        b2[i] = b2[i] - alpha*dL_db2[i]

    return w1, b1, w2, b2
```

We we initialize the network and test for the first derivatives and the loss:

```
    input = [1, -1]
# weights for layer and bias
w1 = [[1.0, 1.0, 1.0], [-1.0, -1.0, -1.0]]
b1 = [0.0, 0.0, 0.0]
w2 = [[1.0, 1.0], [-1.0, -1.0], [-1.0, -1.0]]
b2 = [0.0, 0.0]

# testing
target = [1,0]
derivs  = forward_and_backward(input, w1, b1, w2, b2, target)
text = ["dL_dw1", "dL_db1", "dL_dw2", "dL_db2", "Loss"]
for idx, d in enumerate(derivs):
    print(f"{text[idx]} = {d}")
```

The output for Loss and Gradients after one forward and backward pass are:

```
dL_dw1 = [[0.0, 0.0, 0.0], [-0.0, -0.0, -0.0]]
dL_db1 = [0.0, 0.0, 0.0]
dL_dw2 = [[-0.44039853898894116, 0.44039853898894116], [-0.44039853898894116, 0.440398538988
dL_db2 = [-0.5, 0.5]
Loss = 0.6931471805599453
```

Question 4 Bellow we can see the training function used to train the network as well as the plotting function used to visualize the loss as the network is trained:

```
    import matplotlib.pyplot as pl

# Training and plotting functions
def plot_loss(avg_loss):
    """ Plot avg and epoch Loss """
    plt.figure(figsize=(8,5))
    plt.plot(avg_loss, label = "Avg Loss", linestyle = "-", color = "green")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training Loss Over Epochs")
    plt.legend()
    plt.show()

def train(xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch):
    avg_loss = []
    sample_size = xtrain.shape[0]

    for epoch in range(max_epoch):
    # start training in current epoch
        train_loss = 0
        # training during epoch
        for i in range(sample_size):
            train_instance = np.random.randint(0, sample_size)
            # forward and backwards pass, returning the derivatives
            target = [1, 0] if ytrain[train_instance] == 1 else [0, 1]
            dL_dw1, dL_db1, dL_dw2, dL_db2, loss = forward_and_backward(xtrain[train_instanc
            train_loss += loss
            # update weights
            w1, b1, w2, b2 = gradient_desent(w1, b1, w2, b2, dL_dw1, dL_db1, dL_dw2, dL_db2

        # calculate the avg loss of training
        avg_loss.append(train_loss/sample_size)

        # randomize dataset
        indices = np.arange(sample_size)
        xtrain = np.array([xtrain[i] for i in indices])
        ytrain = np.array([ytrain[i] for i in indices])

    return avg_loss, w1, b2, w2, b2
```

Using the training implementation above, the following initialization, normalization and hyperparameters:

```
    # initialization of weights
w1 = np.random.normal(loc = 0.0, scale = 1.0, size=(2, 3))
b1 = np.zeros(3)
w2 = np.random.normal(loc = 0.0, scale = 1.0, size = (3,2))
b2 = np.zeros(2)

#normalization of data
mean_xtrain = np.mean(xtrain, axis = 0)
sd_xtrain = np.std(xtrain, axis = 0)
norm_xtrain = (xtrain - mean_xtrain)/sd_xtrain
# train model
alpha = 0.01
max_epoch = 25
avg_loss, w1, b1, w2, b2 = train(norm_xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch)
```

we are able to train our neural network. Once the network has been trained we plot the lost against epoch to get the plot bellow:

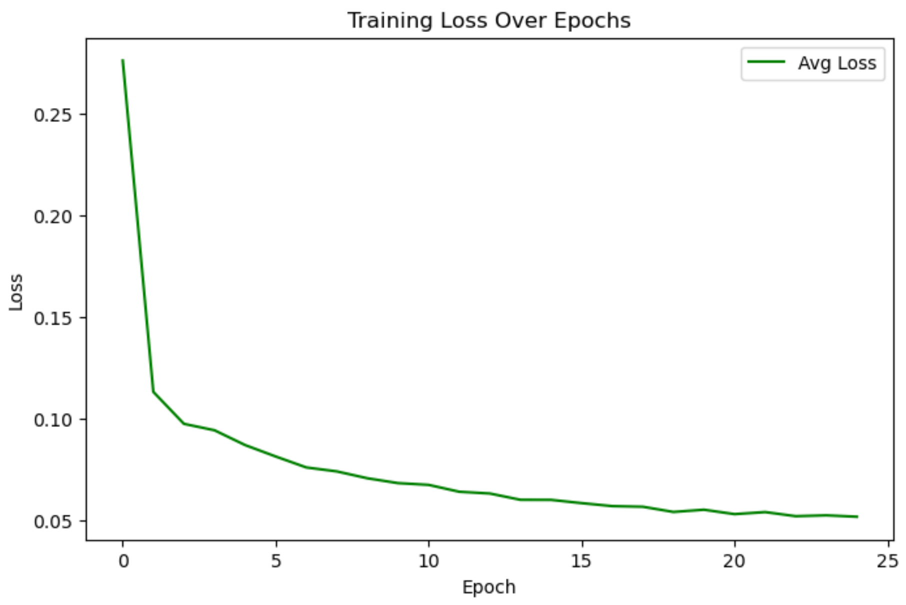

Figure 1: Training Loss over Epoch

The plot showcases how the neural network loss drops steadily until it plateaus close to 0.05.

3 Tensor backpropagation

3.1 Question 5

The code bellow implements a neural network of one hidden layer using vectorization. The network uses the sigmoid function as an activation function for the hidden layer and softmax at the output layer. The network is trained on the MNIST dataset, and utilizes min-max normalization. During the training process we use stochastic gradient descent, but perform the updates only at specific intervals to mimic that of a mini-batch training.

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

def softmax(values: list):
    exp_values = np.exp(values - np.max(values)) # substracting max for numerical stability

    return exp_values/np.sum(exp_values)

def forwardpass(input, w1, b1, w2, b2):

    o1 = np.dot(input, w1) + b1 # first line of linear combination
    h = sigmoid(o1) # sigmoid activation function
    o2 = np.dot(h, w2) + b2 # second linear combination
    y = softmax(o2) # sigmoid activation function

    return o1, h, o2, y

def get_loss(prediction, target):
    """ Calculate loss of current prediction """
    return -np.log(prediction[target])


def deriv_sigmoid(values: list):
    """ Derivative of sigmoid for vectors """
    sigmoid_vals = sigmoid(values)
    return sigmoid_vals*(1 - sigmoid_vals)


def deriv_cross_softmax(pred, target):
    grad = np.copy(pred)
    grad[target] -= 1 # adjust gradient for target
    return grad

def backprop(input, w1, b1, w2, b2, target, o1, h, o2, y):
    #compute loss
    loss = get_loss(y, target)

    # backwards pass to compute derivatives
    dL_do2 = deriv_cross_softmax(y, target) # deriv. of loss with rep. to the secon linear o
    dL_dw2 = np.outer(dL_do2, h).T # deriv. of the loss with regards to second set of weigh
    dL_db2 = dL_do2 # deriv. of the loss with respect to the second set of biases
    dL_dh = np.dot(w2, dL_do2) # deriv. of the loss with respects to the output of the sigmo
    dL_do1 = dL_dh * deriv_sigmoid(o1) # deriv. of the loss with respects to the first linea
    dL_dw1 = np.outer(dL_do1, input).T # deriv. of the loss with respects to the first set o
    dL_db1 = dL_do1 # deriv. of the loss with respects to the first set of biases

    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def forward_and_backward(input, w1, b1, w2, b2, target):
    # feedforward pass
    o1, h, o2, y = forwardpass(input, w1, b1, w2, b2)

    # backwards pass and get derivatives
    dL_dw1, dL_db1, dL_dw2, dL_db2, loss = backprop(input, w1, b1, w2, b2, target, o1, h, o2

    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def gradient_desent(w1, b1, w2, b2, dL_dw1, dL_db1, dL_dw2, dL_db2, alpha):
    # update weights, biases in first layer
    w1 = w1 - alpha*dL_dw1
    b1 = b1 - alpha*dL_db1
    w2 = w2 - alpha*dL_dw2
    b2 = b2 - alpha*dL_db2

    return w1, b1, w2, b2

    def train(xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch, batch_size):
    avg_loss = []
    sample_size = xtrain.shape[0]

    for epoch in range(max_epoch):
    # start training in current epoch
        train_loss = 0
        # create gradient vector for average gradients
        grad_vec = [
                    np.zeros((784, 300)),
                    np.zeros((300, )),
                    np.zeros((300, 10)),
                    np.zeros((10,)),
                    0.0
               ]
        # training during epoch
        for iter in range(len(xtrain)):

            # compute forwards and backwards pass to get derivatives and loss
            (dL_dw1, dL_db1, dL_dw2, dL_db2,  loss) = forward_and_backward(xtrain[iter], w1,

            # add up gradients to work with averages
            for i, grad in enumerate((dL_dw1, dL_db1, dL_dw2, dL_db2, loss)):
                grad_vec[i] += grad

            # update weights when we iterated over a "batch"
            if iter != 0 and iter % batch_size == 0:

                # get gradient average
                grad_vec = [grad/batch_size for grad in grad_vec]
                w1, b1, w2, b2 = gradient_desent(w1, b1, w2, b2, grad_vec[0], grad_vec[1], g

                # update training loss
                train_loss += grad_vec[4]

                # set grad_vec to zero
                for i in range(len(grad_vec)):
                    grad_vec[i] *= 0

            if  iter % (len(xtrain)* 0.1) == 0:
                print(f"Epoch {epoch + 1} at {round(iter/len(xtrain), 1)*100}%")

        # calculate the avg loss of training
        avg_loss.append(train_loss / round(sample_size/batch_size))

        # randomize dataset
        indices = np.arange(sample_size)
        xtrain = np.array([xtrain[i] for i in indices])
        ytrain = np.array([ytrain[i] for i in indices])


    print(f"w1 weights found = {w1}")
    print(f"b1 biases found = {b1}")
    print(f"w2 weights found = {w2}")
    print(f"b2 biases found = {b2}")

    return avg_loss, w1, b1, w2, b2
```

Below we showcase the code used for initializing and training our neural network.

```
    # setting for network
```

```
hidden_layer = 300
input_size = 784
output_size = 10

# initialization of weights
w1 = np.random.normal(loc = 0.0, scale = 1.0, size=(input_size, hidden_layer))
b1 = np.zeros(hidden_layer)
w2 = np.random.normal(loc = 0.0, scale = 1.0, size = (hidden_layer, output_size))
b2 = np.zeros(output_size)

#normalization of data
norm_xtrain = (xtrain/255.0)
# train model
alpha = 0.01
max_epoch = 15
batch_size = 64
avg_loss, w1, b1, w2, b2 = train(norm_xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch, batc
```

From the training of this network, we got the following plot which shows that the network was able to learn the underlying pattern of the data as observed by the reduction in the loss as a function of epochs. It is important to note that the loss curve has not fully plated, therefore hinting that the network would need more epochs in order to finish converging.

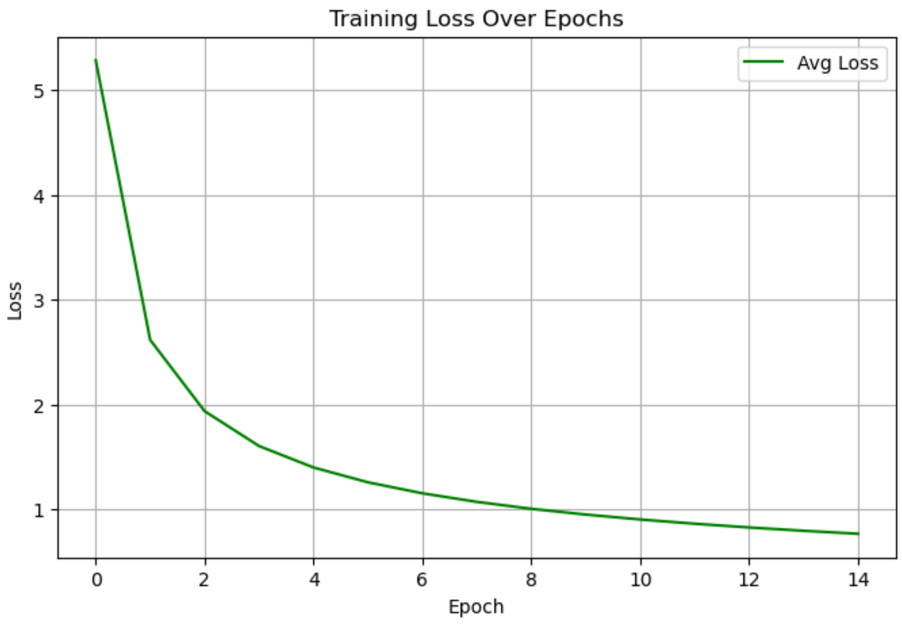

Figure 2: Training Loss over Epoch

3.2

Question 6 For our third implementation, we modified our neural network in order to process **batch of images** all ot once, using tensor operations for efficiency. This means that we needed to modify all operations so that they would work with tensors (aka matrices). The forward and backward passes should handle a whole batch simultaneously, performing matrix multiplications and gradient computations in a single step. This means that we are able to take the gradients of multiple instances all at once and perform gradient descent on the aggregate to speed up the training process. This method reduces the computational cost and speeds up training to a significant degree.

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

def softmax(values: list):
    exp_values = np.exp(values - np.max(values, axis=1, keepdims=True)) # substracting max
    return exp_values/np.sum(exp_values, axis = -1, keepdims = True)

def get_loss(predictions, targets):
    batch_size = predictions.shape[0]
    # epsilon = 1e-12 # buffer to prevent overflow
    loss = np.sum(-np.log(predictions[np.arange(batch_size), targets]))
    return  loss / batch_size


def deriv_sigmoid(values: list):
    sigmoid_vals = sigmoid(values)
    return sigmoid_vals*(1 - sigmoid_vals)


def deriv_cross_softmax(pred, target):
    grad = np.copy(pred)
    batch_size = pred.shape[0]
    grad[np.arange(batch_size), target] -= 1 # adjust gradient for target
    return grad/batch_size

def forwardpass(input, w1, b1, w2, b2):
    # feedforward pass

        # (batch_size, input)@(input, n_hidden) -> (batch_size, n_hidden)
    O1 = np.matmul(input, w1) + b1 # first line of linear combination
        # sigmoid( (batch_size, n_hidden) ) -> (batch_size, n_hidden)
    H = sigmoid(O1) # sigmoid activation function
        # (batch_size, n_hidden)@(n_hidden, output) -> (batch_size, output)
    O2 = np.matmul(H, w2) + b2 # second linear combination
```

```
        # softmax( (batch_size, output) ) -> (batch_size, output)
    Y = softmax(O2) # sigmoid activation function

    return O1, H, O2, Y

def backprop(input, w1, b1, w2, b2, target, O1, H, O2, Y):
    #compute loss
    loss = get_loss(Y, target)

    # backwards pass to compute derivatives
        # (batch_size, output)
    dL_do2 = deriv_cross_softmax(Y, target) # deriv. of loss with rep. to the softmax outpu

        # (n_hidden, output)
    dL_dw2 = np.matmul(H.T, dL_do2)# deriv. of the loss with respect to w2

        # (output, )
    dL_db2 = np.sum(dL_do2, axis = 0) # deriv. of the loss with respect to b2

        # (batch_size, n_hidden)
    dL_dh = np.matmul(dL_do2, w2.T) # deriv. of the loss with respect to the sigmoid output

        #(batch_size, n_hidden)
    dL_do1 = dL_dh * deriv_sigmoid(O1) # deriv. of the loss with respect to the first linea

        # (input, n_hidden)
    dL_dw1 = np.matmul(input.T, dL_do1) # deriv. of the loss with respects to w1

        # (n_hidden, )
    dL_db1 = np.sum(dL_do1, axis = 0) # deriv. of the loss with respects to b1

    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def forward_and_backward(input, w1, b1, w2, b2, target):
    # feedforward pass
    O1, H, O2, Y = forwardpass(input, w1, b1, w2, b2)

    # backwards pass and get derivatives
    dL_dw1, dL_db1, dL_dw2, dL_db2, loss = backprop(input, w1, b1, w2, b2, target, O1, H, O2

    return dL_dw1, dL_db1, dL_dw2, dL_db2, loss

def gradient_desent(w1, b1, w2, b2, dL_dw1, dL_db1, dL_dw2, dL_db2, alpha):
    # update weights and biases
    w1 -= alpha * dL_dw1
    b1 -= alpha * dL_db1
    w2 -= alpha * dL_dw2
    b2 -= alpha * dL_db2

    return w1, b1, w2, b2
def batch_train(xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch, batch_size):
    avg_loss = []

    for epoch in range(max_epoch):
    # start training in current epoch
        train_loss = 0

        # training during epoch
        for iter in range(0, len(xtrain), batch_size):

            # get batch for current iteration
            xbatch = xtrain[iter : batch_size + iter]
            ybatch = ytrain[iter : batch_size + iter]

            # compute forwards and backwards pass to get derivatives and loss
            dL_dw1, dL_db1, dL_dw2, dL_db2,  loss = forward_and_backward(xbatch, w1, b1, w2,

            # aggregate loss
            train_loss += loss

            # update weights through gradient descent
            w1, b1, w2, b2 = gradient_desent(w1, b1, w2, b2, dL_dw1, dL_db1, dL_dw2, dL_db2,

        # calculate the avg loss of training
        avg_loss.append(train_loss / (len(xtrain) / batch_size))

        # randomize dataset
        indices = np.arange(len(xtrain))
        xtrain = np.array([xtrain[i] for i in indices])
        ytrain = np.array([ytrain[i] for i in indices])

        print(f"Epoch {epoch + 1} finished")

    return avg_loss, w1, b1, w2, b2

# setting for network
hidden_layer = 300
input_size = xtrain.shape[1]
output_size = 10

# initialization of weights
w1 = np.random.randn(input_size, hidden_layer)
```

```
b1 = np.zeros(hidden_layer)
w2 = np.random.randn(hidden_layer, output_size)
b2 = np.zeros(output_size)

# normalizing data
xtrain_norm = xtrain/255.0

# train model
alpha = 0.02
max_epoch = 50
batch_size = 64
avg_loss, w1, b1, w2, b2 = batch_train(xtrain, ytrain, w1, b1, w2, b2, alpha, max_epoch, bat
```

The Loss was computed and plotted over the epochs.

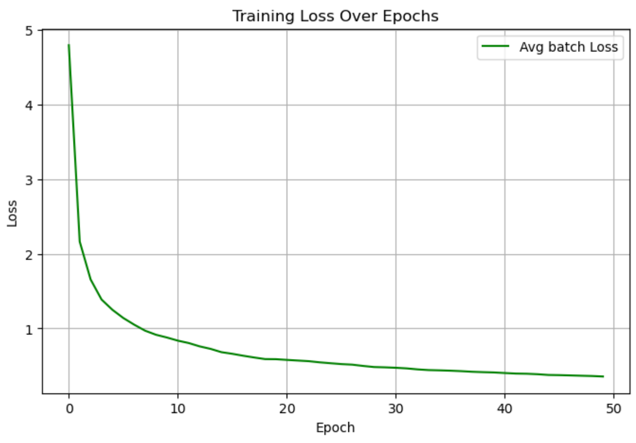

Figure 3: Training Loss over Epoch for mini-batch gradient descent

4 Analysis

4.1

Question 7

Experiment1 For this experiment, we look at the difference between training loss and validation loss over 5 epochs.

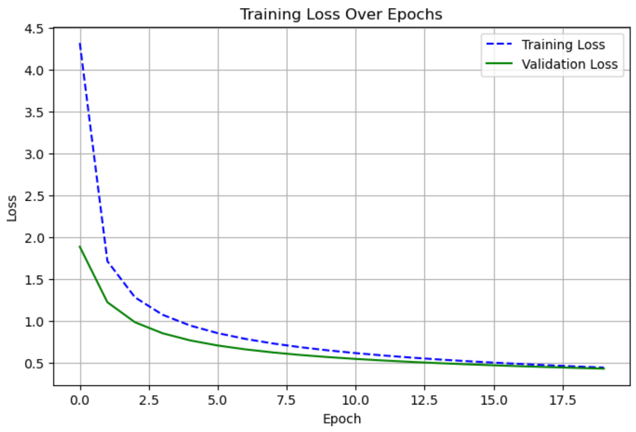

Figure 4: Training Loss vs Validation Loss

From the above plot it is evident that the validation loss is lower than the training set across the first epochs. This is due to the fact that we are calculating the validation set after an entire epoch of training therefore the validation set utilizes weights that have been already exposed to the data and therefore have undergone learning. In contrast, we see that the training loss is higher than the validation loss but this is simply because the training loss is the average of the losses across an entire epoch therefore there is greater fluctuation and the average is just a point estimate, that is a measure of tendency. It is clear that if we would invert the computation of the loss; that is compute first the validation loss and then the training loss, we would see the reverse pattern. Where the validation set would start with a higher loss and gradually get closer to the training set. Yet again the validation loss is composed of a smaller sample size and therefore is less representative as an ”average” yet it holds the property that the network is not learning from it.

Experiment 2 For this experiment, we looked at the network performance given different weight initialization.

As shown by the above grave there is some variability in the network performance but the difference in performance becomes less relevant as the network goes through more epochs. From further experimentation, it was observed that

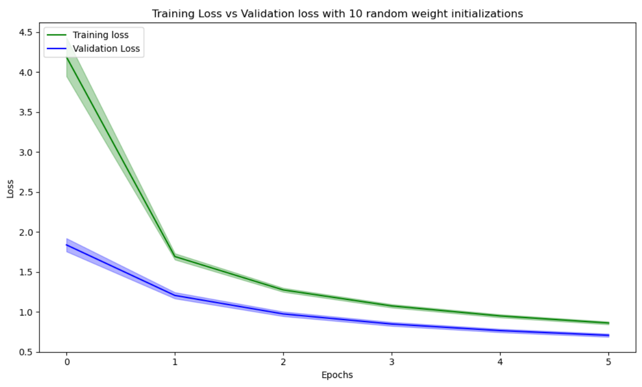

Figure 5: Variation in model performance given different initializations

the more iterations we ran the experiment the larger this variation became. Therefore we can conclude that given the model is indeed sensitive to weight initialization, a good initialization is needed to benefit from this property. It is important to note that for this experiment we did not use any sophisticated initialization techniques and that all weight initialization was sampled from a standard normal Gaussian distribution. That being said using more sophisticated techniques could have a more significant impact on model performance than what we observed.

Experiment 3 For this experiment, we tested the network with four different learning rates to see how this would impact the model's performance and learning.

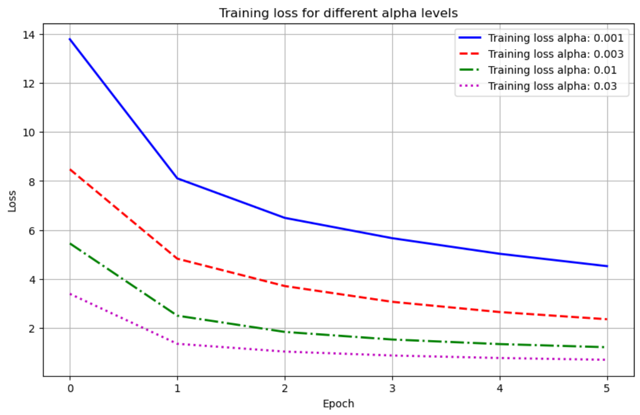

Figure 6: Effects of different learning rates on performance

As seen from the above plot the most effective learning rate by far is $\alpha =$

0.03, followed by $\alpha = 0.01$. The difference between a learning rate of 0.0001 and 0.003 is humongous; $\alpha = 0.0001$ is so small that the network learns ridiculously slow, so much so that by the end of the last epoch the loss for $\alpha = 0.0001$ has not even reached the loss value for epoch 0 for $\alpha = 0.03$. This goes to show how important it is to have an appropriate learning rate as it will enable the network to converge significantly faster and make the learnign/training process much easier.

Experiment 4 Now we train the network on the entire data set with the following hyperparameters: $\alpha = 0.03$ and batch size of 64 for 10 epochs. After the network underwent training we used the validation set to check for accuracy and we got a 91$\%$ accuracy. This goes to show that for a simple Neural Network with only one hidden layer, it has a great capacity to learn, and if we would train it over more epochs the performance would get better. Comparing this to the state of the CNNs, the performance is not at all bad considering the simplicity of the model.