COMP0090 2020/21 Assignment 1: "Bag, not bag"

Sahil Shah
20194624
sahil.shah.20@ucl.ac.uk

Akshaya Natarajan
20069959
akshaya.natarajan.20@ucl.ac.uk

Kamiylah Charles
20092484
kamiylah.charles.20@.ucl.ac.uk

Akshay Parmar
20153279
akshay.parmar.20@ucl.ac.uk

Chanel Sadrettin-Brown
16050121
chanel.sadrettin-brown.20@ucl.ac.uk

1 Contributions

Two smaller groups were formed (Akshay and Sahil in one, Akshaya, Chanel and Kamiylah in another) to complete two problems each. The first group focused on the first and second task, while the second group focused on the third and fourth. From time to time the group came together to help each other on questions and check over the work. Finally, we gathered to complete the final task.

2 Project Code

All project code on Google Colab can be found here:

https://colab.research.google.com/drive/1nOvr_4M-Pna54du83l0b_YQvTIaxxJMj?usp=sharing

3 Tasks

3.1 A memory-efficient voted perceptron

3.1.1 & 3.1.2

The voted perceptron is an algorithm which scales the contribution of successful classifications to the weight vectors by the use of coefficients. This algorithm therefore, has a space complexity of $\mathcal{O}(k)$, where k is the number of misclassifications. A variant of the voted perceptron is demonstrated in the code below, with a memory complexity of $\mathcal{O}(1)$. $\mathcal{O}(1)$ represents a constant memory allocated during the algorithm, where the size of data structures is not increased during the algorithm, but the values are simply updated. So each time our weight w is updated the memory allocated to the array is the same.

$$\sum_{i=1}^{k} c_i(\mathbf{w_i} \cdot \mathbf{x}) = \left(\sum c_i \mathbf{w_i}\right)^T \cdot \mathbf{x} \tag{1}$$

with $\mathbf{w_i}$ being the weights, and the c_i being weighting coefficients. The mean weight vector was set as: $\mathbf{w_m} = \sum_{i=1}^{k} c_i \mathbf{w_i}$, then:

$$\text{mean\_w} = 0$$
$$\text{mean\_w} = w \cdot c$$

Data: $\mathcal{D} \coloneqq \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_n}, y_n)\}$
Result: $v \coloneqq \{\mathbf{w_m}\}$
$\mathbf{w} \coloneqq 0^d$;
$\mathbf{w_m} \coloneqq 0^d$;
$c \coloneqq 1$;
while *not converged* **do**
  for $i \coloneqq 1 \ldots, n$ **do**
    $\hat{y} \coloneqq \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x_i} \geq 0 \\ 0 & \text{otherwise} \end{cases}$;
    if $\hat{y} = y$ **then**
      $c \coloneqq c + 1$;
    else
      $\mathbf{w_m} \coloneqq \mathbf{w_m} + \mathbf{w} \cdot c$;
      $\mathbf{w} \coloneqq \mathbf{w} + (y - \hat{y})\mathbf{x_i}$;
      $c \coloneqq 1$;
    end
  end
end

Algorithm 1: The voted-perceptron training variant algorithm with bias terms omitted.

The coefficients represent the number of iterations for which each weight vector correctly classifies the data points. Therefore, the weight vectors with the most successes have the heaviest contribution to the mean weight vector. As a result, this mean weight vector in the efficient variant will be functionally equivalent to the weight matrix, as the updated weights will be summed in a weighted fashion to the mean weight vector instead of being appended to the weight matrix in the voted perceptron.

3.1.3 & 3.1.4 Implementation

Our data sets were reshaped as each image consists of 28x28 pixels, meaning each image then occupied a feature space of length 784.

```
dev_images = np.load("fashion-dev-imgs.npz").reshape(784, 1000)
dev_labels = np.load("fashion-dev-labels.npz")
test_images = np.load("fashion-test-imgs.npz").reshape(784,
    1000)
test_labels = np.load("fashion-test-labels.npz")
train_images = np.load("fashion-train-imgs.npz").reshape(784,
    12000)
train_labels = np.load("fashion-train-labels.npz")
```

Then the weight, mean weight and coefficients were initialised.

```
w = np.zeros(784)
mean_w = np.zeros(784)
c = 1

no_epochs = 100
accuracies = []
dev_accuracies = []

for l in range(no_epochs):
    accuracy = 0
```

```
for i in range(len(train_labels)):
    x = train_images.T[i].flatten()
    if w@x >= 0:
        pred_labels = 1
    else:
        pred_labels = 0
    if pred_labels == train_labels[i]:
        c += 1
    else:
        w = w + np.multiply((train_labels[i] - pred_labels)
            * x, c)
        c = 1

pred_train_labels=np.zeros(len(train_labels))
for i in range(len(train_labels)):
    if w @ train_images[:,i] >= 0:
        pred_train_labels[i] = 1
    else:
        pred_train_labels[i] = 0
train_count = sum(map(lambda x : x==0.0, abs(train_labels -
    pred_train_labels)))
accuracies.append(train_count/len(train_labels))

pred_dev_labels=np.zeros(len(dev_labels))
for i in range(len(dev_labels)):
    if w @ dev_images[:,i] >= 0:
        pred_dev_labels[i] = 1
    else:
        pred_dev_labels[i] = 0
dev_count = sum(map(lambda x : x==0.0, abs(dev_labels -
   pred_dev_labels)))
dev_accuracies.append(dev_count/len(dev_labels))
```

3.1.5 Accuracy Plot on Training Set

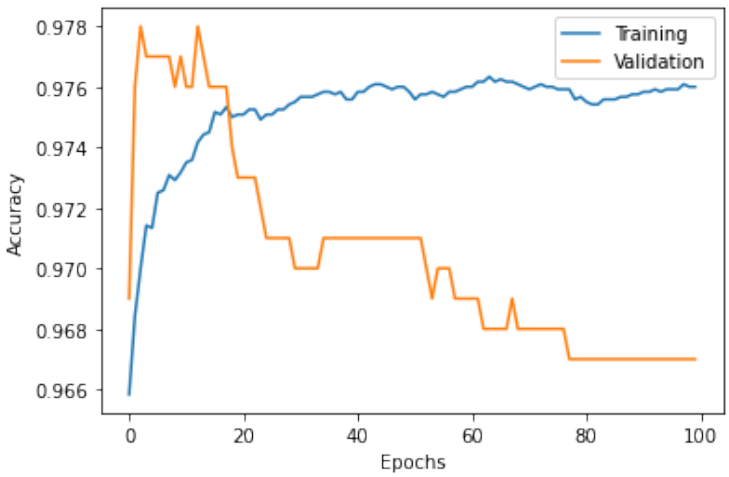

Figure 1: Accuracy against Epochs for Training and Validation sets

From Figure 1, it can be seen that the training accuracy increases with the number of epochs and converges to an accuracy of approximately 97.6%, while the validation accuracy decreases slightly and converges to an accuracy of approximately 96.8%.

3.1.6 Provide the accuracy on the training and validation set, for the epoch on which you obtain your highest accuracy on the validation set.

The highest validation accuracy was found at epoch number 2 and this was 97.8%. The corresponding training accuracy at epoch number 2 was 97%.

This code provides the calculation of the epoch number for the highest accuracy in the validation set and therefore the corresponding training and validation accuracies.

```
best_epoch = np.argmax(dev_accuracies)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    dev_accuracies[best_epoch])
print('Training accuracy at epoch', best_epoch,' =', accuracies
    [best_epoch])
```

3.2 Mean squared-loss logistic regression

3.2.1 - Deriving the analytical gradients for the model parameters given the square loss

Given the mean squared-loss function shown in Equation 2, the analytical gradients of the loss function with respect to the weights and biases are shown in Equations 3 and 4.

$$L = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \hat{y}_i)^2}{2} \tag{2}$$

$$\nabla_w L = -\frac{1}{n}\sum_{i=1}^{n}(y - \sigma(Xw+b)) \cdot \sigma'(Xw+b)) \cdot X \tag{3}$$

$$\nabla_b L = -\frac{1}{n}\sum_{i=1}^{n}(y - \sigma(Xw+b)) \cdot \sigma'(Xw+b)) \tag{4}$$

Where $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\sigma'(x) = \sigma(x)(1-\sigma(x))$.

The code for the logistic function, the logistic derivative, analytical gradients with respect to the weights and bias and loss functions were written as shown below.

```
def logistic(x):
    "Produce the logistic function"
    return (1/(1+np.exp(-x)))

def logistic_derivative(x):
    "Produce the derivative of the logistic function"
    return np.multiply(logistic(x),(1-logistic(x)))

def grad_w(y, w, X, b):
    "Differentiate with respect to w"
    return (-1/y.shape[1])*np.multiply((y-logistic(w.T@X+ b)),
       logistic_derivative(w.T@X+b))@X.T

def summed_grad_w(y, w, X, b):
    "Differentiate with respect to w"
    return (-1/y.shape[1])*np.sum(np.multiply((y-logistic(w.T@X
       + b)),logistic_derivative(w.T@X+b))@X.T)

def grad_b(y, w, X, b):
    "Differentiate with respect to b"
    return (-1/y.shape[1])*np.sum(np.multiply((y-logistic(w.T@X
        + b)),logistic_derivative(w.T@X+b)))
```

```
def loss(y, w, X, b):
    "Provide mean square loss function"
    return (1/(2*y.shape[1]))*np.sum((y-logistic(w.T@X + b))
        **2)
```

3.2.2 Verifying that our analytical gradients are correct using finite difference

We have provided four examples; two for gradients with respect to the weight and two with respect to the bias:

```
def finite_difference_w(y, w, X, b, e):
    "Finite differences gradient with respect to w"
    return ((loss(y, w + 0.5*e, X, b)) - (loss(y, w - 0.5*e, X,
        b)))/e

def finite_difference_b(y, w, X, b, e):
    "Finite differences gradient with respect to b"
    return ((loss(y, w, X, b + 0.5*e)) - (loss(y, w, X, b -
        0.5*e)))/e


w = np.random.randn(784).reshape(-1, 1)
b = 0

grad_b(train_labels, w, train_images, b)


# gradient checking using finite differences for first sample
X1 = train_images[:,1000].reshape(784,1)
Y1 = train_labels[:,1000].reshape(1,1)
grad_b(Y1, w, X1, b)

print(finite_difference_b(Y1, w, X1, b, 0.01))

if abs(grad_b(Y1, w, X1, b) - finite_difference_b(Y1, w, X1, b,
    0.01)) <= 1e-7:
    print('Good bias approximation')

summed_grad_w(Y1, w, X1, b)

finite_difference_w(Y1, w, X1, b, 0.000001)

if abs(summed_grad_w(Y1, w, X1, b) - finite_difference_w(Y1, w,
    X1, b, 0.000001)) <= 1e-7:
    print('Good weight approximation')

# gradient checking using finite differences for second sample
X2 = train_images[:,2].reshape(784,1)
Y2 = train_labels[:,2].reshape(1,1)
grad_b(Y2, w, X2, b)

finite_difference_b(Y2, w, X2, b, 0.01)

if abs(grad_b(Y2, w, X2, b) - finite_difference_b(Y2, w, X2, b,
    0.01)) <= 1e-7:
    print('Good bias approximation')

summed_grad_w(Y2, w, X2, b)

finite_difference_w(Y2, w, X2, b, 0.000001)
```

```
if abs(summed_grad_w(Y2, w, X2, b) - finite_difference_w(Y2, w,
    X2, b, 0.000001)) <= 1e-7:
    print('Good weight approximation')
```

Using the defined finite difference functions, the gradient of the loss functions, with respect to the weights and bias were verified as being correct. The first sample consisted of the thousand and first data point in the training set of 12,000 points. The gradient with respect to the weights (summed across all 784 pixels) was found to be 0.1144949 to seven decimal points and the finite difference function produced a gradient of 0.1144949 also to seven decimal points. The gradient with respect to the bias was found to be 0.0009378 to seven decimal points and the corresponding finite difference function produced a gradient of 0.0009378 to seven decimal points.

The second sample consisted of the third data point in the training set. The gradient with respect to the weights (again summed across all 784 pixels) was found to be -0.1940856 to seven decimal points and the finite difference function produced a gradient of -0.1940856 also to seven decimal points. The gradient with respect to the bias was found to be 0.0008555 to seven decimal points and the corresponding finite difference function produced a gradient of 0.0008555 to seven decimal points.

3.2.3 & 3.2.4 - Algorithm Implementation & Training using Full Batch Gradient Descent

```
epoch_no = 1000
learning_rate = 0.1
training_losses = []
training_accuracies = []
validation_losses = []
validation_accuracies = []
w = np.zeros(784).reshape(-1, 1)
b = 0

for l in range(epoch_no):
    accuracy = 0
    train_loss = 0

    prev_b = b
    b = b - learning_rate*grad_b(train_labels, w, train_images,
        b)

    prev_w = w
    w = w - learning_rate*grad_w(train_labels, w, train_images,
        prev_b).T


    t_count = np.sum((train_labels == ((logistic(w.
       T@train_images + b) >= 0.5)*1.0))*1.0)
    v_count = np.sum((dev_labels == ((logistic(w.T@dev_images +
        b) >= 0.5)*1.0))*1.0)

    epoch_loss = loss(train_labels, w, train_images, b)
    training_losses.append(epoch_loss)
    training_accuracies.append(t_count/train_labels.shape[1])
    validation_losses.append(loss(dev_labels, w, dev_images, b)
       )
    validation_accuracies.append(v_count/dev_labels.shape[1])
```

3.2.5 - Providing a plot of the square loss on the training set for each epoch

Figure 2 shows how the mean squared loss changes with each epoch, showing the loss minimisation for both training and validation sets.

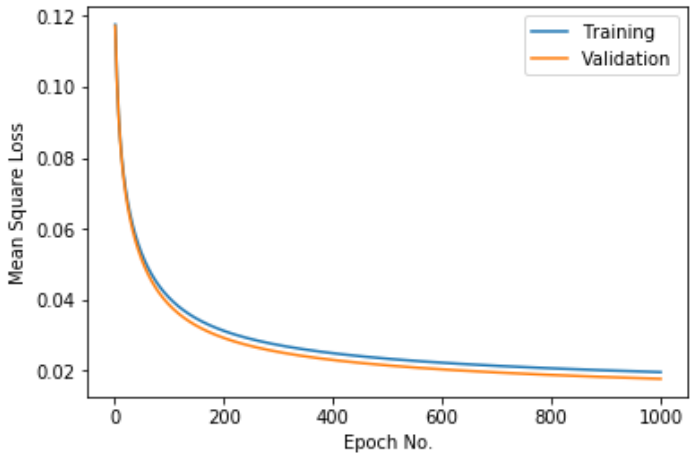

Figure 2: Plot of Mean Square Loss against Epoch

It can be seen that both the training and validation sets provide losses which are very similar for each epoch, with the validation set producing slightly lower mean squared loss with an increasing number of epochs.

3.2.6 - Providing a plot of the accuracy on the training set for each epoch

Figure 3 shows how the accuracy of the model changes with each epoch, showing the convergence to a high accuracy for both the training and validation sets.

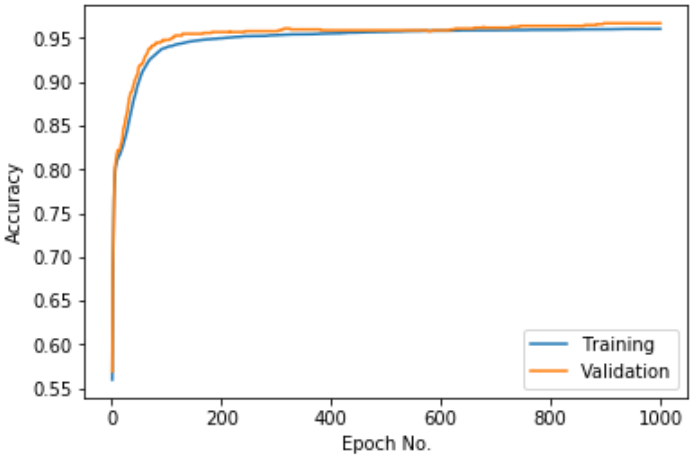

Figure 3: Plot of Accuracy against Epochs

It can be seen that both the training and validation sets produce accuracies which are again very similar for each epoch, showing a convergence to a high accuracy as the number of epochs increases. Furthermore, as the number of epochs increases, the validation set produces a slightly higher accuracy than the training set.

3.2.7 - Providing the accuracy on the training and validation set, for the epoch on which we obtained the highest accuracy on the validation set.

The highest validation accuracy was found at epoch number 896 and this was 96.7%. The corresponding training accuracy at epoch number 896 was 96.0%.

This code provides the calculation of the epoch number for the highest accuracy in the validation set and therefore the corresponding training and validation accuracies.

```
best_epoch = np.argmax(validation_accuracies)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,'=',
    validation_accuracies[best_epoch])
print('Training accuracy at epoch', best_epoch,'=', round(
    training_accuracies[best_epoch],5))
```

3.3 Three-layer multi-layer perceptron

In order to code to 3-layer MLP, the backward propagation gradients had to be derived, Equations 5 to 29 show how this was done. A diagram to represent the stages in forward and backward propagation is also given as Figure 4.

$$\begin{aligned} z &= W^T X + b \\ A &= \sigma(z) \end{aligned} \tag{5}$$

where W are weights, X is the input, b are biases.

Square brackets (as in $z^{[l]}$) indicate the layer number. Whereas round brackets (as in $z^{(i)}$) indicate the training example number. m is the number of samples in the training set (12,000) and n is the number of features in the input (784). L is the maximum number of layers, with l being the l'th layer.

3.3.1 Forward Propagation

$$\begin{aligned} z^{[1]} &= W^{[1]} X + b^{[1]} \\ A^{[1]} &= \sigma(z^{[1]}) \end{aligned} \tag{6}$$

$$\begin{aligned} z^{[2]} &= W^{[2]} A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(z^{[2]}) \end{aligned} \tag{7}$$

$$\begin{aligned} z^{[3]} &= W^{[3]} A^{[2]} + b^{[3]} \\ A^{[3]} &= \sigma(z^{[3]}) \end{aligned} \tag{8}$$

Where $A^{[3]}$ is the output.

3.1.1 - Assuming the log-likelihood loss, derive the analytical gradients for the model parameters:

The cost function to minimise is: (where y represents the training data , $\hat{y}$ represents the predictions and i denotes current example)

$$C = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(\hat{y}^{(i)}) + (1 - \hat{y}^{(i)}) \log(1 - \hat{y}^{(i)})] \tag{9}$$

To minimise find the partial derivative of the cost function with respect to each weight and bias in each layer of the neural network

$\frac{\partial C}{\partial w_{[1]}}, \frac{\partial C}{\partial w_{[2]}}, \frac{\partial C}{\partial w_{[3]}}$ $\frac{\partial C}{\partial b[1]}, \frac{\partial C}{\partial b[2]}, \frac{\partial C}{\partial b[3]}$

General formula for the l'th layer using the chain rule:

$$\begin{aligned} \frac{\partial C}{\partial w^{[l]}} &= \frac{\partial C}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial w^{[l]}} \\ \frac{\partial C}{\partial b^{[l]}} &= \frac{\partial C}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial b^{[l]}} \end{aligned} \tag{10}$$

The general formula can be split at each layer, starting back propagation at layer 3 (final layer):

$$\begin{aligned}\frac{\partial C}{\partial w^{[3]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial w^{[3]}} \\ \frac{\partial C}{\partial b^{[3]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial b^{[3]}}\end{aligned} \tag{11}$$

Layer 2:

$$\begin{aligned}\frac{\partial C}{\partial w^{[2]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial z^{[2]}}{\partial w^{[2]}} \\ \frac{\partial C}{\partial b^{[2]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial z^{[2]}}{\partial b^{[2]}}\end{aligned} \tag{12}$$

Layer 1:

$$\begin{aligned}\frac{\partial C}{\partial w^{[1]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial w^{[1]}} \\ \frac{\partial C}{\partial b^{[1]}} &= \frac{\partial C}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial b^{[1]}}\end{aligned} \tag{13}$$

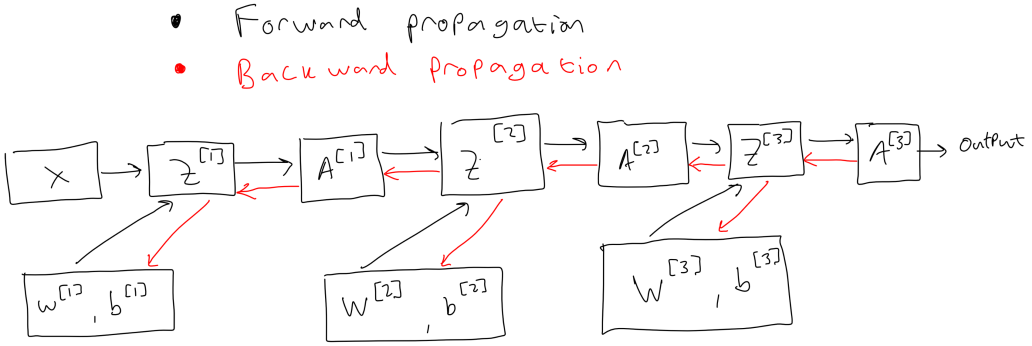

Figure 4: Process of Forward and Backward Propagation

For the activation function $\sigma(z) = \frac{1}{1+e^{-z}}$ to find its derivative with respect to z, use the chain rule and let $u = 1 + e^{-z}$, then $\frac{du}{dz} = -e^{-z}$.

Then $\sigma(z) = \frac{1}{u}$, and $\frac{d\sigma(z)}{du} = -\frac{1}{u^2} = -\frac{1}{(1+e^{-z})^2}$, and therefore $\frac{d\sigma(z)}{du}\frac{du}{dz} = \frac{d\sigma(z)}{dz}$.

$$\begin{aligned}\frac{d\sigma(z)}{dz} &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}}\left(\frac{e^{-z}+1-1}{1+e^{-z}}\right) \\ &= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) \\ &= \sigma(z)(1-\sigma(z))\end{aligned} \tag{14}$$

so $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

(10). The derivations for the individual parts are derived and explained by (Budhwant, 2018).

$$\frac{\partial C}{\partial z^{[L]}} = a^{[L]} - y \tag{15}$$

$$\frac{\partial C}{\partial z^l} = \left(W^{(l+1)T} \cdot \frac{\partial C}{\partial z^{[l+1]}} \right) \sigma(z^{(l)}) \tag{16}$$

$$\frac{\partial z^{[l]}}{\partial w^{[l]}} = a^{(l-1)} \tag{17}$$

$$\frac{\partial C}{\partial w^{[L]}} = \frac{\partial C}{\partial z^{[L]}} \cdot {a^{[l-1]}}^T \tag{18}$$

Terminology in code:

1. $y =$`Y`
2. $a^{[0]} =$`X`
3. $a^{[l]} =$`Al`
4. $z^{[l]} =$`Zl`
5. $\sigma(z) =$`sigmoid`
6. $\sigma'(z) =$`sigmoid gradient`

From the summaries provided above, the gradients at each layer of back propagation were calculated:
Layer 3 gradients:

$$\frac{\partial C}{\partial z^{[3]}} = a^{[3]} - y \tag{19}$$

$$\frac{\partial z^{[3]}}{\partial w^{[3]}} = a^{[2]} \tag{20}$$

$$\begin{aligned} \frac{\partial C}{\partial w^{[3]}} &= \frac{\partial C}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}} \\ &= (a^{[3]} - y)a^{[2]} \end{aligned} \tag{21}$$

Layer 2 gradients:

$$\begin{aligned} \frac{\partial C}{\partial z^{[2]}} &= \left(W^{[3]T} \frac{\partial C}{\partial z^{[3]}} \right) \cdot \sigma'(z^{[2]}) \\ &= \left(W^{[3]T}(a^{[3]} - y) \right) \cdot \sigma'(z^{[2]}) \end{aligned} \tag{22}$$

$$\frac{\partial z^{[2]}}{\partial w^{[2]}} = a^{[1]} \tag{23}$$

$$\begin{aligned} \frac{\partial C}{\partial w^{[2]}} &= \frac{\partial C}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}} \\ &= \left[(W^{[3]T}(a^{[3]} - y))\sigma'(z^{[2]}) \right] a^{[1]} \end{aligned} \tag{24}$$

Layer 1 gradients:

$$\begin{aligned} \frac{\partial C}{\partial z^{[1]}} &= \left(W^{[2]T} \cdot \frac{\partial C}{\partial z^{[2]}} \right) \sigma'(z^{[1]}) \\ &= \left(W^{[2]t} \cdot (W^{[3]T} \cdot (a^{[3]} - y)) \right) \sigma'(z^{[2]})\sigma'(z^{[1]}) \end{aligned} \tag{25}$$

$$\frac{\partial z^{[1]}}{\partial w^{[1]}} = a^{[0]} = X \tag{26}$$

$$\begin{gathered} \frac{\partial C}{\partial w^{[1]}} = \frac{\partial C}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}} \\ \left(W^{[2]t} \cdot (W^{[3]T} \cdot (a^{[3]} - y))\right) \sigma'(z^{[2]}) \sigma'(z^{[1]} X) \end{gathered} \tag{27}$$

Summary: gradients coded

$$\begin{aligned} \frac{\partial C}{\partial w^{[3]}} &= (a^{[3]} - y) a^{[2]} \\ \frac{\partial C}{\partial w^{[2]}} &= \left(W^{[3]T} \frac{\partial C}{\partial z^{[3]}} \cdot \sigma'(z^{[2]})\right) a^{[1]} \\ \frac{\partial C}{\partial w^{[1]}} &= \left(W^{[2]T} \frac{\partial C}{\partial z^{[2]}} \cdot \sigma'(z^{[1]})\right) X \end{aligned} \tag{28}$$

The gradients with respect to biases can be generalised as: $\frac{\partial C}{\partial b^{(l)}} = \frac{\partial C}{\partial z^{(l)}}$, as $\frac{\partial z^{(l)}}{\partial b^{(l)}} = 1$.

$$\begin{aligned} \frac{\partial C}{\partial b^{[3]}} &= a^{[3]} - y \\ \frac{\partial C}{\partial b^{[2]}} &= \left(\left(W^{[3]T} \cdot \frac{\partial C}{\partial z^{[3]}}\right) \sigma'(z^{[2]})\right) a^{[1]} \\ \frac{\partial C}{\partial b^{[1]}} &= \left(\left(W^{[2]T} \cdot \frac{\partial C}{\partial z^{[2]}}\right) \sigma'(z^{[1]})\right) X \end{aligned} \tag{29}$$

Initialisation functions:

```
# initialising parameters before optimization on training set
def initial_parameters(input_size, h1, h2, output_size):

    W3 = np.random.randn(output_size,h2)
    W2 = np.random.randn(h2,h1)
    W1 = np.random.randn(h1,input_size)

    b3 = np.zeros((output_size,1))
    b2 = np.zeros((h2,1))
    b1 = np.zeros((h1,1))

    parameters = (W1, b1 , W2 , b2 , W3 ,b3)

    return parameters

# loss function, cost will be calculated during our training
   for loop
def log_like(y_true, y_pred):
    f = -np.log(y_pred),y_true) +
        np.multiply(-np.log(1 - y_pred), 1 - y_true
    log_l = np.multiply(f)
    return log_l

# defining the sigmoid activation function and its derivative
def sigmoid_gradient(z):
    sigp = sigmoid(z) * (1-sigmoid(z))
    return sigp
```

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

3.3.2 - Verifying that the analytical gradients are correct using finite differences. Three examples provided.

(Ng)

```
def gradient_check_n(parameters, gradients, X, Y, epsilon = 1e
    -7):

    # Set-up variables
    parameters_values, _ = dictionary_to_vector(parameters)
    grad = gradients_to_vector(gradients)
    num_parameters = parameters_values.shape[0]
    J_plus = np.zeros((num_parameters, 1))
    J_minus = np.zeros((num_parameters, 1))
    gradapprox = np.zeros((num_parameters, 1))

    # Compute gradapprox
    for i in range(num_parameters):
        thetaplus = np.copy(parameters_values)
        thetaplus[i][0] =   thetaplus[i][0] +  epsilon
        y_pred_p , _ =  forward_prop(X, vector_to_dictionary(
            thetaplus))
        log_l = log_like(Y, y_pred_p)
        J_plus[i] = 1./1 * np.sum(log_l)

        thetaminus = np.copy(parameters_values)
        thetaminus[i][0] = thetaminus[i][0] -  epsilon
        y_pred_m, _ = forward_prop(X, vector_to_dictionary(
            thetaminus))
        log_l = log_like(Y, y_pred_m)
        J_minus[i] = 1./1 * np.sum(log_l)

        gradapprox[i] = (J_plus[i] - J_minus[i])/(2* epsilon)



    numerator = np.linalg.norm(gradapprox - grad)
    denominator = np.linalg.norm(gradapprox ) + np.linalg.norm(
        grad)
    difference =  numerator/denominator

    if difference >= 1e-7:
        print ("Backward propagation doesn't work = " + str(
            difference))
    else:
        print ("Backward propagation works perfectly = " + str(
            difference))

    return difference
```

Example 1:

```
X1 = X_train[:,2]
X1 = X1.reshape(784,1)
Y1 = Y_train[:,2]
```

```
Y1 = Y1.reshape(1,1)

parameters = initial_parameters(X1.shape[0],10,10,Y1.shape[0])

_, forward_pass = forward_prop(X1, parameters)
gradients = back_prop(X1, Y1, forward_pass)
difference = gradient_check_n(parameters, gradients, X1, Y1)
```

Output of example 1 was:

Backward propagation works perfectly = 5.141918151479845e-08

Example 2:

```
X2 = X_train[:,1]
X2 = X2.reshape(784,1)
Y2 = Y_train[:,1]
Y2 = Y2.reshape(1,1)

_, forward_pass = forward_prop(X2, parameters)
gradients = back_prop(X2, Y2, forward_pass)
difference = gradient_check_n(parameters, gradients, X2, Y2)
```

Output of example 2:

Backward propagation works perfectly = 5.519067012377234e-08

Example 3:

```
X3 = X_train[:,10000]
X3 = X3.reshape(784,1)
Y3= Y_train[:,10000]
Y3 = Y3.reshape(1,1)

_, forward_pass = forward_prop(X3, parameters)
gradients = back_prop(X3, Y3, forward_pass)
difference = gradient_check_n(parameters, gradients, X3, Y3)
```

Output of example 3:

Your Backward propagation works perfectly = 3.1854708610947 08e-08

3.3.3 and 3.3.4 - Implementing the algorithm and training the model to convergence using full-batch gradient descent on the training set.

Set-up for training algorithm (and gradient checking):

```
# calculating accuracy
def accuracy(y_true, y_pred):
    sim = 0
    y_pred = np.array(y_pred).reshape(1,len(y_pred))
    for k in range(y_true.shape[1]):
        if y_true[0][k] == y_pred[0][k]:
            sim += 1

    return (sim/y_true.shape[1])

# mapping y_pred < 0.5 to 0 and else 1
def threshold_func(y_pred):
    y_pred_mapped =[]
    for l in range(y_pred.shape[1]):
        if y_pred[0][l] >= 0.5:
```

```
            y_pred_mapped.append(1)
        else:
            y_pred_mapped.append(0)
    return y_pred_mapped


# function to calculate forward propogation
def forward_prop(X, parameters):

    m = X.shape[1]
    W1, b1 , W2 , b2 , W3 , b3 = parameters

    Z1 = W1 @ X + b1
    A1 = sigmoid(Z1)
    Z2 = W2 @ A1 + b2
    A2 = sigmoid(Z2)
    Z3 = W3 @ A2 + b3
    A3 = sigmoid(Z3)

    forward_pass = (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3,
        b3)

    return A3, forward_pass



#function to calculate back propogation
def back_prop(X, Y, forward_pass):

    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) =
       forward_pass

    dZ3 = (A3 - Y)
    dW3 = ((dZ3@A2.T))/m
    db3 = (np.sum(dZ3,axis=1,keepdims=True))/m

    dA2 = W3.T@dZ3

    dZ2=  ((dA2) * sigmoid_gradient(Z2))
    dW2 = ((dZ2@A1.T))/m
    db2 = (np.sum(dZ2,axis=1,keepdims=True))/m

    dA1 = W2.T@dZ2

    dZ1 = ((dA1) * sigmoid_gradient(Z1))
    dW1 = ((dZ1@X.T))/m
    db1 = (np.sum(dZ1,axis=1,keepdims=True))/m

    gradients = (dZ3, dW3, db3, dA2, dZ2, dW2, db2, dA1, dZ1,
       dW1, db1)

    return gradients

# function performing full batch gradient descent
def gradient_descent(parameters, gradients, learning_rate):

    # information from forward and backprop
```

```
    dZ3, dW3, db3, dA2, dZ2, dW2, db2, dA1, dZ1, dW1, db1 =
        gradients
    W1, b1 , W2 , b2 , W3 ,b3 = parameters

    # updating weights and biases
    W3 -= learning_rate * dW3
    W2 -= learning_rate * dW2
    W1 -= learning_rate * dW1

    b3 -= learning_rate * db3
    b1 -= learning_rate * db1
    b2 -= learning_rate * db2

    parameters = ( W1, b1 , W2 , b2 , W3 ,b3 )


    return parameters
```

Training algorithm:

```
# main function for training the model
def training_model(X, Y, h1, h2, epochs,learning_rate):
    m = X.shape[1]
    cost_stored = []
    accuracies = []
    best_val_accuracy = np.zeros((epochs,2))

    # initialization of weights, biases
    parameters = initial_parameters(X.shape[0], h1, h2, Y.shape
        [0])

    for i in range(epochs):

        # forward propogation
        y_pred, forward_pass = forward_prop(X, parameters)

        # calculating loss using log likelihood loss function
        log_l = log_like(Y, y_pred)
        cost = 1./m * np.sum(log_l)
        cost_stored.append(cost)

        # training accuracy per epoch
        accuracies.append(accuracy(Y, threshold_func(y_pred)))
        best_val_accuracy[i][1] = accuracy(Y, threshold_func(
            y_pred))

        # back propogation
        gradients = back_prop(X, Y, forward_pass)

        # gradient descent
        parameters = gradient_descent(parameters, gradients,
            learning_rate)

        # validation accuracy per epoch
        y_pred_val, forward_pass = forward_prop(X_val,
            parameters)
        y_pred_mapped_val = threshold_func(y_pred_val)
        best_val_accuracy[i][0] = accuracy(Y_val,
            y_pred_mapped_val)
```

```
        if i % 10 == 0:
            print('Epoch ',i,' - Cost: ',round(cost,5),' -
                Accuracy: ',
                    round(accuracies[i],5))


    # final accuracy
    y_hat = threshold_func(y_pred)
    print('Final Accuracy is : ', accuracy(Y,y_hat))

    return y_hat, accuracies, cost_stored , best_val_accuracy


y_hat, accuracies, cost_stored, best_val_accuracy =
    training_model(X_train, Y_train, 300, 300, 1000, 0.1)
```

3.3.5 - Providing a plot of the training set for each epoch:

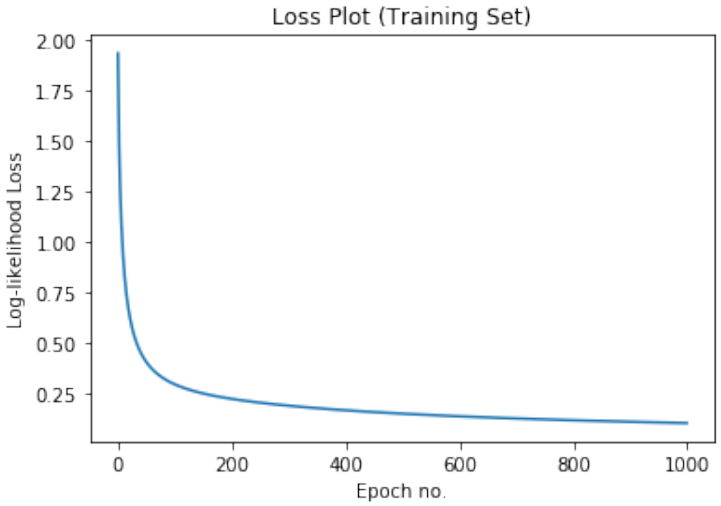

Figure 5: Plot of loss on the training set for each epoch.

3.3.6 - Providing a plot of the accuracy set for each epoch:

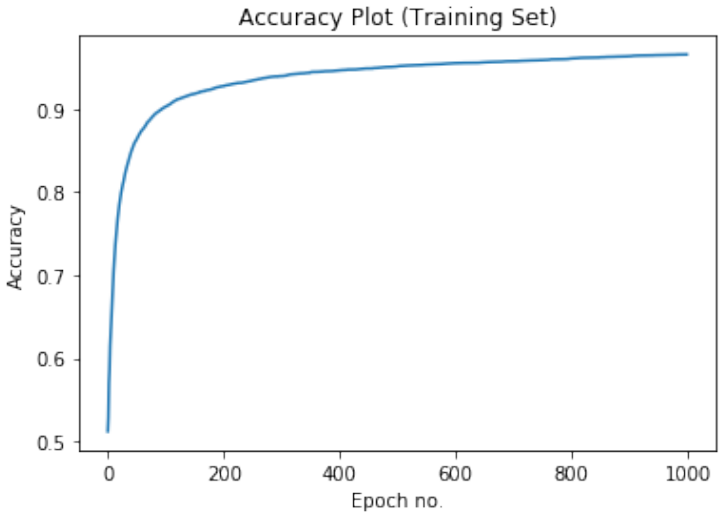

Figure 6: Plot of accuracy on the training set for each epoch.

3.3.7 - Providing the accuracy on the training and validation set, for the epoch on which we obtained the highest accuracy on the validation set:

The epoch with the highest accuracy on the validation set was epoch number 965, resulting in a validation accuracy of 96.6% and a corresponding training accuracy of 95.7%

```
#best epoch
best_idx, _ = np.argmax(best_val_accuracy, axis=0)
print('Epoch with best validation accuracy: ', best_idx)
print('Training accuracy: ',best_val_accuracy[best_idx][1])
print('Validation accuracy: ',best_val_accuracy[best_idx][0])
```

3.4 Hyperparameter tuning

```
# logistic sigmoid function

def sigmoid(z):
    sig = 1/(1+np.exp(-z))

    return sig

# derivative of the logistic sigmoid function

def sig_prime(z):
    sig_dev = sigmoid(z) * (1-sigmoid(z))

    return sig_dev

# set up random weights and zero biases to initialize before
   optimization on training set

def initial_parameters(input_size, h1, output_size):


    W2 = np.random.randn(output_size,h1)
    W1 = np.random.randn(h1,input_size)
```

```
    b2 = np.zeros((output_size,1))
    b1 = np.zeros((h1,1))

    parameters = (W2, b2, W1, b1)
    return parameters


def forward_prop(X, parameters):

    W2, b2, W1, b1 = parameters
    # forward propagation steps to calculate output, the values
        of y that the model predicts (A3)

    Z1 = W1@X + b1
    A1 = sigmoid(Z1)
    Z2 = W2@A1 + b2
    A2 = sigmoid(Z2)

    y_pred = A2
    forward_pass = (Z2, Z1, A2, A1)

    return y_pred, forward_pass


# define cost function that needs to be minimized, log
   likelihood

def cost_loss(y_pred, y_true, parameters):

    m_samples = y_true.shape[1]  # total number of samples in
       set being tested
    loss_func = (y_true * np.log(y_pred)) + ((1-y_true) * np.
       log(1 - y_pred))
    cost_func = -(1/m_samples) * np.sum(loss_func)

    return cost

# implementation of backpropagation

def back_prop(parameters, forward_pass, X, Y):

    m = X.shape[1]
    Z2, Z1, A2, A1 = forward_pass
    W2, b2, W1, b1 = parameters
    y_pred = A2

    dZ2 = (y_pred - Y)
    dW2 = ((dZ2@A1.T))/m
    db2 = (np.sum(dZ2,axis=1,keepdims=True))/m

    dZ1 = ((W2.T@dZ2) * sig_prime(Z1))
    dW1 = ((dZ1@X.T))/m
    db1 = (np.sum(dZ1,axis=1,keepdims=True))/m

    gradients = (dW2, db2, dW1, db1)

    return gradients
```

```
def gradient_descent(parameters, gradients, learning_rate):

    # information from forward and backprop
    dW2, db2, dW1, db1 = gradients
    W2, b2, W1, b1 = parameters

    # updating weights and biases
    W2 -= learning_rate * dW2
    W1 -= learning_rate * dW1

    b1 -= learning_rate * db1
    b2 -= learning_rate * db2

    parameters = (W2, b2, W1, b1)

    return parameters

def accuracy(y_true, y_pred):
    sim = 0

    for k in range(y_true.shape[1]):
        if y_true[0][k] == y_pred[0][k]:
            sim += 1

    return (sim/y_true.shape[1])

def threshold_func(y_pred):
    y_pred_mapped = []
    for l in range(y_pred.shape[1]):
        if y_pred[0][l] >= 0.5:
            y_pred_mapped.append(1)
        else:
            y_pred_mapped.append(0)
    y_pred_mapped = np.array(y_pred_mapped).reshape(1, len(
       y_pred_mapped))
    return y_pred_mapped


def gradient_descent_momentum(parameters, gradients,
   learning_rate, momentum, velocity):

    # information from forward and backprop
    W2, b2, W1, b1 = parameters
    vdw1, vdb1, vdw2, vdb2 = velocity
    dW2, db2, dW1, db1 = gradients


    # updating weights and biases

    vdw1 = momentum * vdw1 - learning_rate * dW1
    vdw2 = momentum * vdw2 - learning_rate * dW2
    vdb1 = momentum * vdb1 - learning_rate * db1
    vdb2 = momentum * vdb2 - learning_rate * db2

    W1 += vdw1
    W2 += vdw2
    b1 += vdb1
    b2 += vdb2
```

```
    parameters = (W2, b2, W1, b1)
    velocity = (vdw1, vdb1, vdw2, vdb2)
    return parameters, velocity


def initial_parameters_mom(input_size, h1, output_size):


    W2 = np.random.randn(output_size,h1)
    W1 = np.random.randn(h1,input_size)

    b2 = np.zeros((output_size,1))
    b1 = np.zeros((h1,1))

    parameters = (W2, b2, W1, b1)

    vdw1 = np.zeros((h1,input_size))
    vdw2 = np.zeros((output_size,h1))
    vdb1 = np.zeros((h1,1))
    vdb2 = np.zeros((output_size,1))

    velocity = (vdw1, vdb1, vdw2, vdb2)

    return parameters, velocity
```

Question 3.4.1.a Full Batch Gradient Descent without momentum:

```
# main function for training the model
def full_batch(X, Y, h1, epochs,learning_rate):
    m = X.shape[1]
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))

    # initialization of weights, biases
    parameters = initial_parameters(X.shape[0], h1, Y.shape[0])

    for i in range(epochs):

        # forward propogation
        y_pred, forward_pass = forward_prop(X, parameters)

        # calculating loss using log likelihood loss function
        cost = cost_loss(y_pred, Y, parameters)

        #training loss per epoch
        losses[i][1] = cost

        # training accuracy per epoch
        # accuracies.append(accuracy(Y, threshold_func(y_pred))
            )
        accuracies[i][1] = accuracy(Y, threshold_func(y_pred))

        # back propogation
        gradients = back_prop(parameters, forward_pass, X, Y)

        # gradient descent
        parameters = gradient_descent(parameters, gradients,
            learning_rate)
```

```
        # validation loss & accuracy per epoch
        y_pred_val, forward_pass = forward_prop(X_val,
            parameters)
        losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
        y_pred_mapped_val = threshold_func(y_pred_val)
        accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val)


        if i % 10 == 0:
            print('Epoch ',i,'==> Training loss: ', round(
                losses[i][1],4), '- Train accuracy: ', round(
                accuracies[i][1],4) , '- Validation loss: ',
                round(losses[i][0],4), '- Validation accuracy:
                ', round(accuracies[i][0],4))


    # final accuracy
    y_hat = threshold_func(y_pred)
    print('Final Accuracy is : ', accuracy(Y,y_hat))

    return y_hat, losses , accuracies, parameters

    y_hat, fb_losses , fb_accuracies, _ = full_batch(X_train,
        Y_train, 300, 200, 0.1)



best_epoch = np.argmax(fb_accuracies[:,0],axis=0)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    fb_accuracies[best_epoch][0])
print('Training accuracy at epoch', best_epoch,' =',
    fb_accuracies[best_epoch][1])
```

Plotting accuracy over epochs for the train and validation sets:

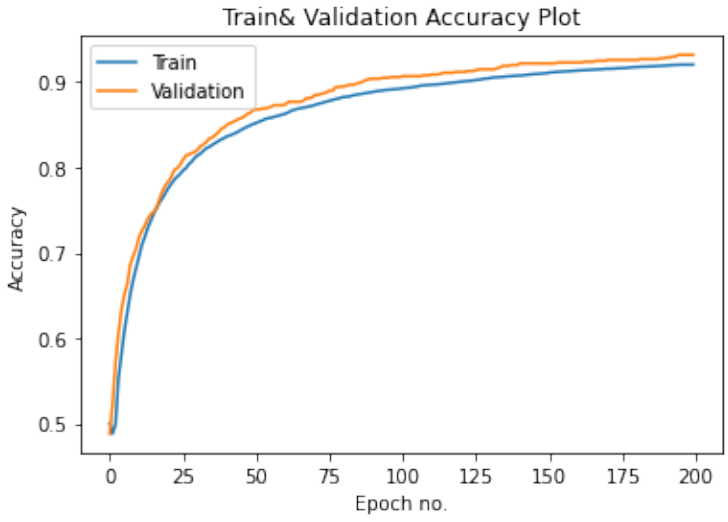

Figure 7: Full Batch Gradient Descent Accuracy Plot (without Momentum)

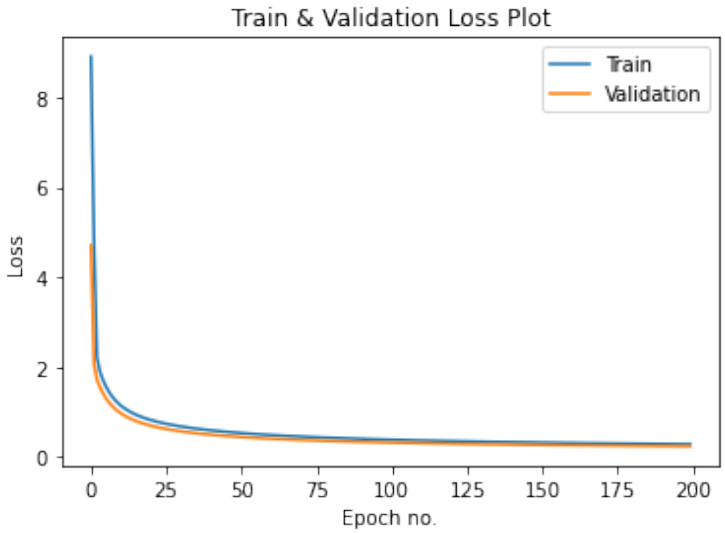

Figure 8: Full Batch Gradient Descent Loss Plot (without Momentum)

3.4.1.d Full Batch Gradient Descent with Momentum

```
# main function for training the model
def full_batch_momentum(X, Y, h1, epochs, learning_rate,
    momentum):

    m = X.shape[1]
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))

    # initialization of weights, biases
    parameters, velocity = initial_parameters_mom(X.shape[0],
        h1, Y.shape[0])

    for i in range(epochs):
        # forward propogation
        y_pred, forward_pass = forward_prop(X, parameters)

        # calculating loss using log likelihood loss function
        cost = cost_loss(y_pred, Y, parameters)
        losses[i][1] = cost


        # training accuracy per epoch
        accuracies[i][1] = accuracy(Y, threshold_func(y_pred))

        # back propogation
        gradients = back_prop(parameters, forward_pass, X, Y)

        # gradient descent
        parameters, velocity = gradient_descent_momentum(
            parameters, gradients, learning_rate, momentum,
            velocity)

        # validation accuracy per epoch
```

```
            y_pred_val, forward_pass = forward_prop(X_val,
                parameters)
            losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
            y_pred_mapped_val = threshold_func(y_pred_val)
            accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val)

            if i % 10 == 0:
                print('Epoch ',i,'==> Training loss: ', round(
                    losses[i][1],4), '- Train accuracy: ', round(
                    accuracies[i][1],4) , '- Validation loss: ',
                    round(losses[i][0],4), '- Validation accuracy:
                    ', round(accuracies[i][0],4))


        # final accuracy
        y_hat = threshold_func(y_pred)
        print('Final Accuracy is : ', accuracy(Y,y_hat))

        return y_hat, losses , accuracies, parameters

y_hat, fbm_losses , fbm_accuracies, _ = full_batch_momentum(
    X_train, Y_train, 300, 200, 0.1, 0.9)
```

```
best_epoch = np.argmax(fbm_accuracies[:,0],axis=0)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    fbm_accuracies[best_epoch][0])
print('Training accuracy at epoch', best_epoch,' =',
    fbm_accuracies[best_epoch][1])
```

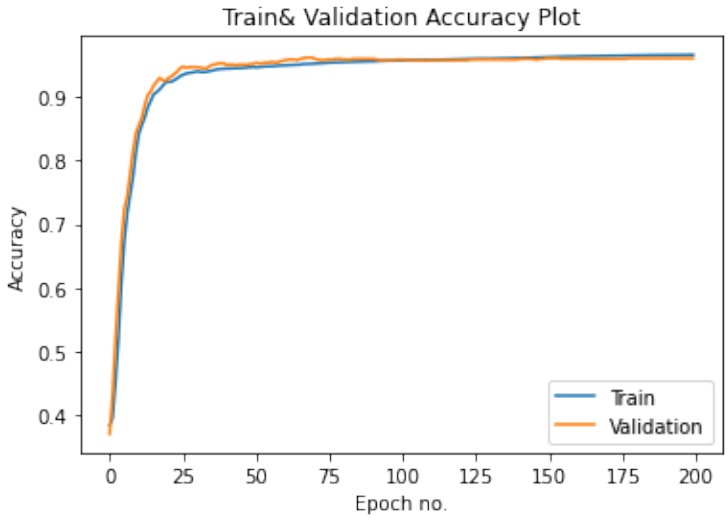

Figure 9: Full Batch Gradient Descent Accuracy Plot (with Momentum)

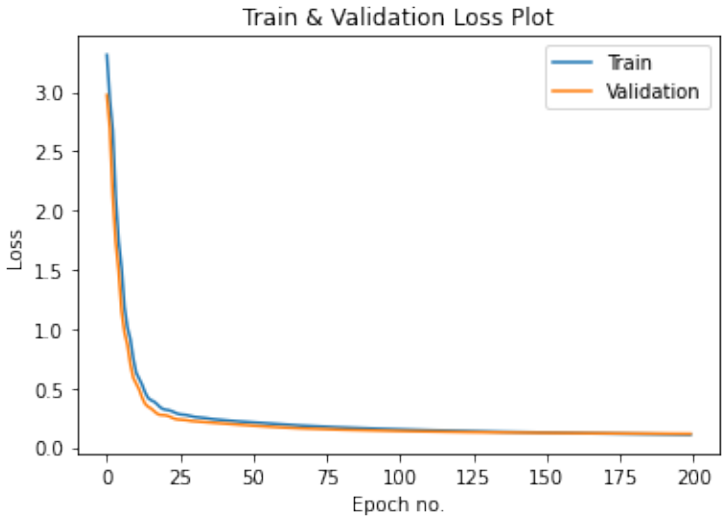

Figure 10: Full Batch Gradient Descent Loss Plot(with Momentum)

Question 3.4.1.b Stochastic Gradient Descent without Momentum

```
def sgd(X, Y, h1, epochs, learning_rate):
    m = X.shape[1]
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))
    parameters = initial_parameters(X.shape[0], h1, Y.shape[0])
    indexes = np.array([i for i in range(12000)])

    for i in range(epochs):
        np.random.shuffle(indexes)
        X_epoch = X_train[:, indexes]
        Y_epoch = Y_train[:, indexes]

        loss = 0
        acc = 0
        for k in range(X.shape[1]):
            X1 = X_epoch[:,k].reshape(784,1)
            Y1 = Y_epoch[:,k].reshape(1,1)

            y_pred, forward_pass = forward_prop(X1, parameters)

            loss += (1/m) * cost_loss(y_pred, Y1, parameters)

            acc += (1/m) * accuracy(threshold_func(y_pred), Y1)

            gradients = back_prop(parameters, forward_pass, X1,
                Y1)
            parameters = gradient_descent(parameters, gradients
                , learning_rate)

        losses[i][1] = loss
        accuracies[i][1] = acc

        # validation accuracy per epoch
        y_pred_val, forward_pass = forward_prop(X_val,
            parameters)
```

```
            losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
            y_pred_mapped_val = threshold_func(y_pred_val)
            accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val) #
                0 for validation accuracy


            if i % 10 == 0:
                print('Epoch ',i,'==> Training loss: ', round(
                    losses[i][1],4), '- Train accuracy: ', round(
                    accuracies[i][1],4) , '- Validation loss: ',
                    round(losses[i][0],4), '- Validation accuracy:
                    ', round(accuracies[i][0],4))

        return losses, accuracies, parameters

sgd_losses, sgd_accuracies, _ = sgd(X_train, Y_train, 100, 50,
    0.001)

best_epoch = np.argmax(sgd_accuracies[:,0],axis = 0)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    sgd_accuracies[best_epoch][0])
print('Training accuracy at epoch', best_epoch,' =',
    sgd_accuracies[best_epoch][1])
```

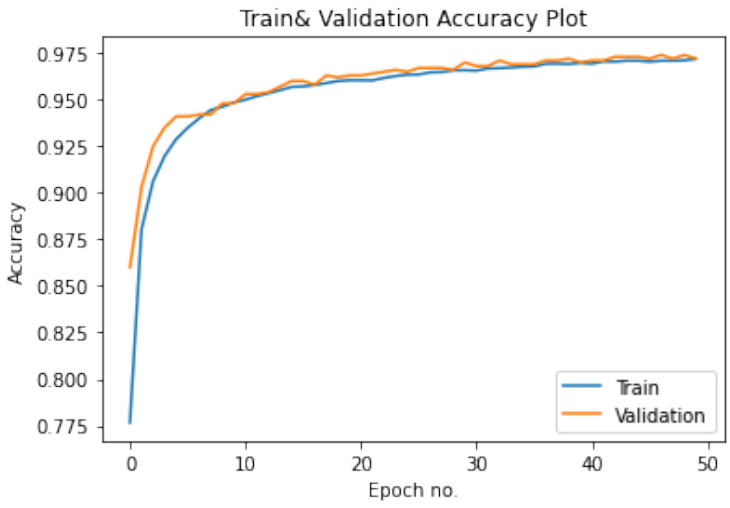

Figure 11: Stochastic Gradient Descent Accuracy Plot (without Momentum)

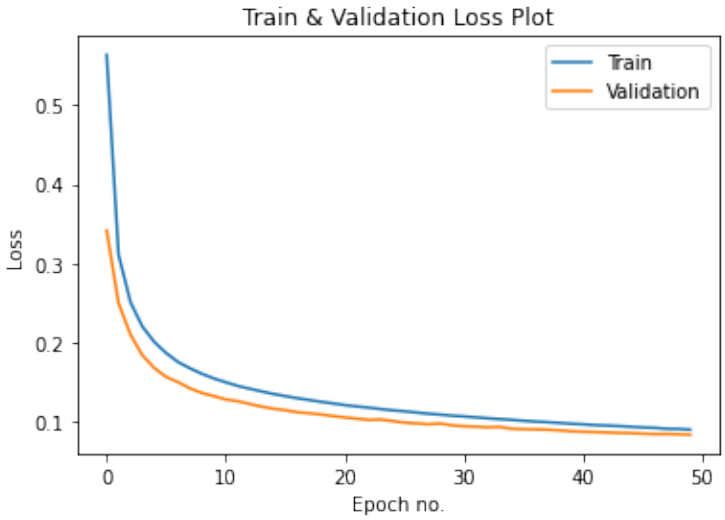

Figure 12: Stochastic Gradient Descent Loss Plot (without Momentum)

Question 3.4.1.e Stochastic Gradient Descent with momentum

```
# SGD with momentum
def sgd_momentum(X, Y, h1, epochs, learning_rate,momentum):
    m = X.shape[1]
    parameters, velocity = initial_parameters_mom(X.shape[0],
       h1, Y.shape[0])
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))
    indexes = np.array([i for i in range(12000)])

    for i in range(epochs):
        np.random.shuffle(indexes)
        X_epoch = X_train[:, indexes]
        Y_epoch = Y_train[:, indexes]

        loss = 0
        acc = 0
        for k in range(X.shape[1]):
            X1 = X_epoch[:,k].reshape(784,1)
            Y1 = Y_epoch[:,k].reshape(1,1)

            y_pred, forward_pass = forward_prop(X1, parameters)

            loss += (1/m) * cost_loss(y_pred, Y1, parameters)

            acc += (1/m) * accuracy(threshold_func(y_pred), Y1)

            gradients = back_prop(parameters, forward_pass, X1,
                Y1)
            parameters, velocity = gradient_descent_momentum(
               parameters, gradients, learning_rate, momentum,
                velocity)

        losses[i][1] = loss
        accuracies[i][1] = acc
```

```
            # validation accuracy per epoch
            y_pred_val, forward_pass = forward_prop(X_val,
                parameters)
            losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
            y_pred_mapped_val = threshold_func(y_pred_val)
            accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val) #
                0 for validation accuracy

            if i % 10 == 0:
                print('Epoch ',i,'==> Training loss: ', round(
                    losses[i][1],4), '- Train accuracy: ', round(
                    accuracies[i][1],4) , '- Validation loss: ',
                    round(losses[i][0],4), '- Validation accuracy:
                    ', round(accuracies[i][0],4))

        return losses, accuracies, parameters

sgdm_losses, sgdm_accuracies, _ = sgd_momentum(X_train, Y_train
    , 100, 50, 0.001, 0.7)

best_epoch = np.argmax(sgdm_accuracies[:,0],axis=0)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    sgdm_accuracies[best_epoch][0])
print('Training accuracy at epoch', best_epoch,' =',
    sgdm_accuracies[best_epoch][1])
```

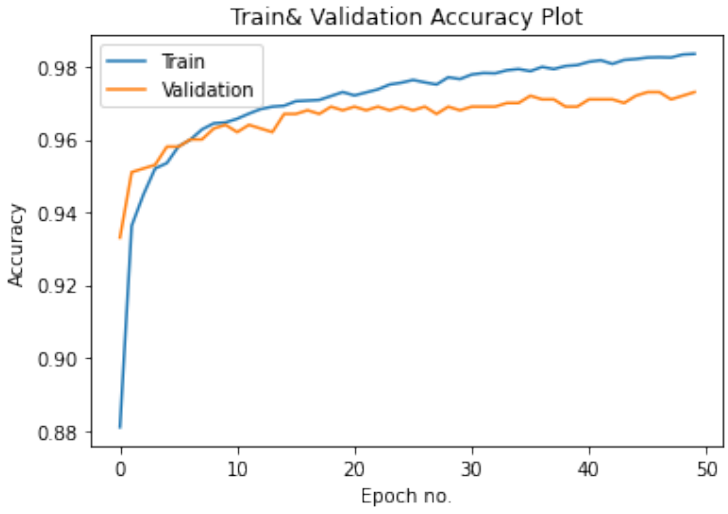

Figure 13: Stochastic Gradient Descent Accuracy Plot (with Momentum)

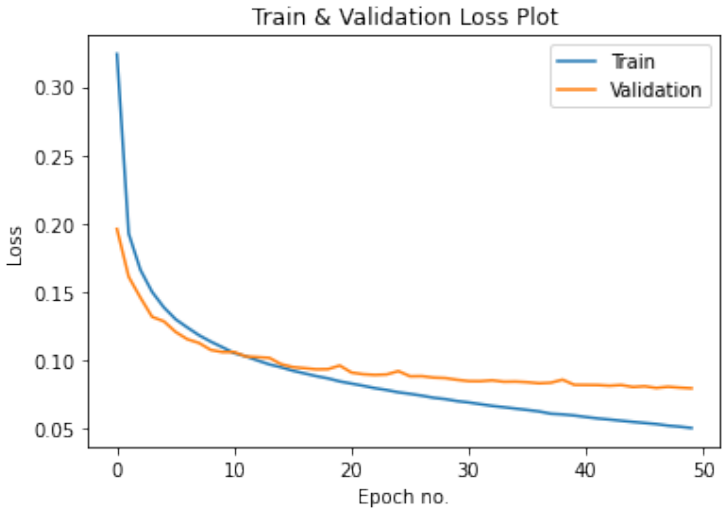

Figure 14: Stochastic Gradient Descent Loss Plot (with Momentum)

Question 3.4.1.c Mini-Batch Gradient without Momentum

```
#helper functions for mini batch
def mini_batches_split(X, Y, batch_size):
    minibatches = []

    indexes = np.array([i for i in range(12000)])
    np.random.shuffle(indexes)
    X_epoch = X[indexes, :]
    Y_epoch = Y[indexes, :]

    for i in range(0, X_epoch.shape[0], batch_size):
        X_mini = X_epoch[i:i + batch_size]
        Y_mini = Y_epoch[i:i + batch_size]

        minibatches.append((X_mini, Y_mini))

    return minibatches


def mini_batch(X, Y, h1, epochs, learning_rate, batch_size):

    parameters = initial_parameters(X.shape[0], h1, Y.shape[0])
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))
    for i in range(epochs):
        loss = 0
        acc = 0
        mini_batches = mini_batches_split(X.T, Y.T, batch_size)
        no_of_batches = len(mini_batches)
        for m in mini_batches:
            X_mini, Y_mini = m
            X_mini = X_mini.T
            Y_mini = Y_mini.T
            m_sample = X_mini.shape[1]

            y_pred, forward_pass = forward_prop(X_mini,
                parameters)
```

```
            loss += (1/no_of_batches) * cost_loss(y_pred,
                Y_mini, parameters)
            acc += (1/no_of_batches) * accuracy(threshold_func(
                y_pred), Y_mini)

            gradients = back_prop(parameters, forward_pass,
                X_mini, Y_mini)
            parameters = gradient_descent(parameters, gradients
                , learning_rate)


        losses[i][1] = loss
        accuracies[i][1] = acc

        # validation accuracy per epoch
        y_pred_val, forward_pass = forward_prop(X_val,
            parameters)
        losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
        y_pred_mapped_val = threshold_func(y_pred_val)
        accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val) #
            0 for validation accuracy

        if i % 10 == 0:
            print('Epoch ',i,'==> Training loss: ', round(
                losses[i][1],4), '- Train accuracy: ', round(
                accuracies[i][1],4) , '- Validation loss: ',
                round(losses[i][0],4), '- Validation accuracy:
                ', round(accuracies[i][0],4))

    return losses, accuracies, parameters

mb_losses, mb_accuracies, _ = mini_batch(X_train, Y_train, 100,
    50, 0.05, 64)

best_epoch = np.argmax(mb_accuracies[:,0],axis=0)
print('Epoch with max validation accuracy: ', best_epoch)
print('Validation accuracy at epoch', best_epoch,' =',
    mb_accuracies[best_epoch][0])
print('Training accuracy at epoch', best_epoch,' =',
    mb_accuracies[best_epoch][1])
```

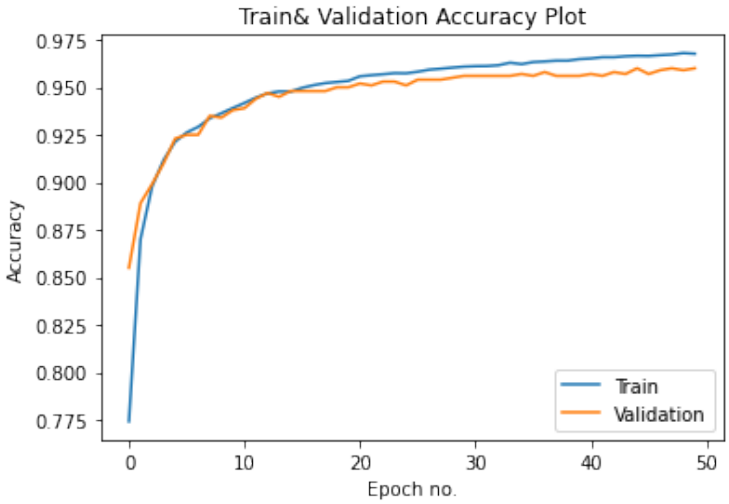

Figure 15: Mini-Batch Gradient Descent Accuracy Plot (without Momentum)

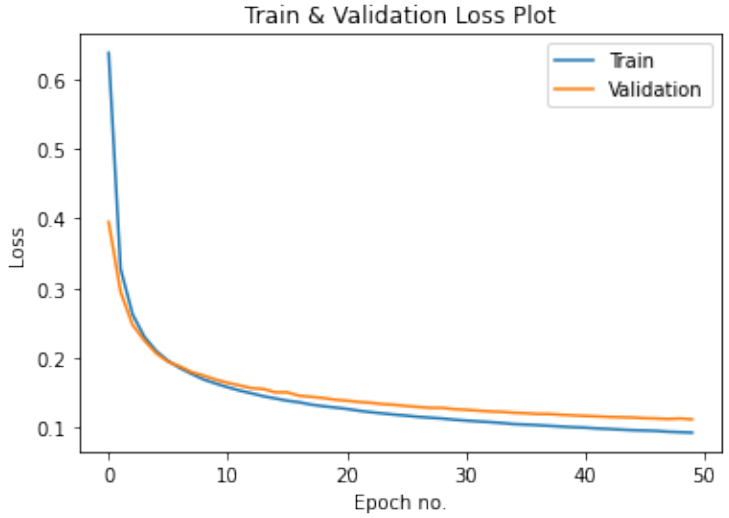

Figure 16: Mini-Batch Gradient Descent Loss Plot (without Momentum)

Question 3.4.1.f Mini-Batch Gradient Descent with Momentum

```
def mini_batch_momentum(X, Y, h1, epochs, learning_rate,
    momentum, batch_size):

    parameters, velocity = initial_parameters_mom(X.shape[0],
        h1, Y.shape[0])
    accuracies = np.zeros((epochs,2))
    losses = np.zeros((epochs,2))

    for i in range(epochs):
        loss = 0
        acc = 0
        mini_batches = mini_batches_split(X.T, Y.T, 64)
        no_of_batches = len(mini_batches)
```

```
        for m in mini_batches:
            X_mini, Y_mini = m
            X_mini = X_mini.T
            Y_mini = Y_mini.T
            m_sample = X_mini.shape[1]

            y_pred, forward_pass = forward_prop(X_mini,
               parameters)
            loss += (1/no_of_batches) * cost_loss(y_pred,
               Y_mini, parameters)
            acc += (1/no_of_batches) * accuracy(threshold_func(
               y_pred), Y_mini)

            gradients = back_prop(parameters, forward_pass,
               X_mini, Y_mini)
            parameters, velocity = gradient_descent_momentum(
               parameters, gradients, learning_rate, momentum,
                velocity)

        losses[i][1] = loss
        accuracies[i][1] = acc

        # validation accuracy per epoch
        y_pred_val, forward_pass = forward_prop(X_val,
           parameters)
        losses[i][0] = cost_loss(y_pred_val, Y_val, parameters)
        y_pred_mapped_val = threshold_func(y_pred_val)
        accuracies[i][0] = accuracy(Y_val, y_pred_mapped_val) #
            0 for validation accuracy

        if i % 10 == 0:
            print('Epoch ',i,'==> Training loss: ', round(
               losses[i][1],4), '- Train accuracy: ', round(
               accuracies[i][1],4) , '- Validation loss: ',
               round(losses[i][0],4), '- Validation accuracy:
               ', round(accuracies[i][0],4))


    return losses, accuracies , parameters

mbm_losses, mbm_accuracies, _ = mini_batch_momentum(X_train,
   Y_train, 100, 50, 0.05, 0.8, 64)
```

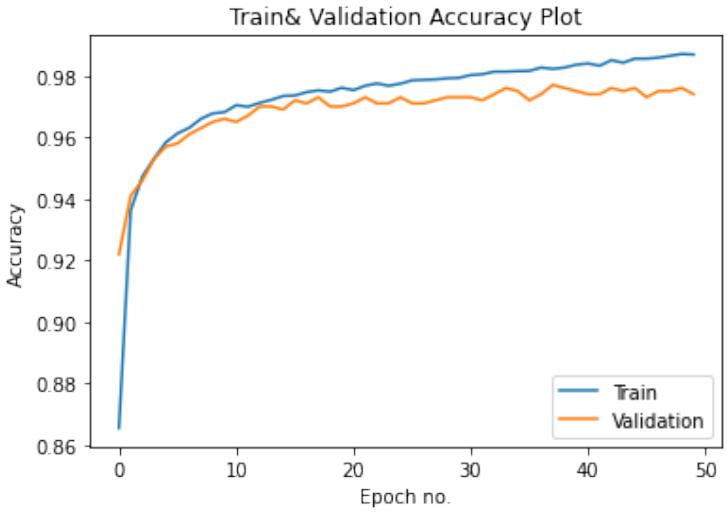

Figure 17: Mini-Batch Gradient Descent Accuracy Plot (with Momentum)

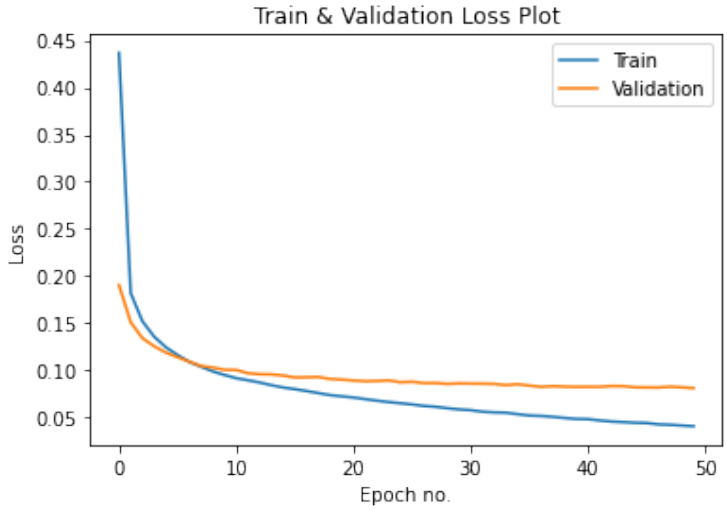

Figure 18: Mini-Batch Gradient Descent Accuracy Plot (with Momentum)

3.4.2 $(\mathrm{a}, \mathrm{b}, \mathrm{c})$ *Tuning hyperparameters*

Question 3.4.3 -

Optimal hyperparameter values for Full Batch Gradient Descent:

Learning rate = 0.1, Momentum = 0.9

Optimal hyperparameter values for Stochastic Gradient Descent:

Learning rate = 0.001, Momentum = 0.7

Optimal hyperparameter values for Mini-Batch Gradient Descent:

Learning rate: 0.05, Momentum: 0.8, Batch size: 64

In order to find the optimal parameters we created a for loop that loops over all three hyperparameters (two in the case of Full Batch and Stochastic Gradient Descents. We then searched for the combination that minimised the loss and took those to be the 'good' hyper parameters. See the code below this

process was applied to our models for both Stochastic and Mini-Batch Gradient Descent with Momentum.

Question 3.4.4 and 3.4.5 -

Please see the plots for the good parameters above below their respective models.

Question 3.4.6 -

The Mini-Batch Gradient Descent with Momentum method gave the highest accuracy on the validation set. This was achieved at epoch number 37, with a training accuracy of 98.2% and a validation accuracy of 97.7%.

```
# Hyperparameter search code for minibatch gradient descent

batch_sizes = [64,128,256]
learning_rate_range = [0.001, 0.005, 0.01, 0.05, 0.1]
momentum_range = [0.7,0.8,0.9]
losses = np.zeros((len(learning_rate_range),len(batch_sizes),
    len(momentum_range)))

for lr in range(len(learning_rate_range)):
    for b in range(len(batch_sizes)):
        for m in range(len(momentum_range)):
            _ , _ , parameters = mini_batch_momentum(X_train,
                Y_train, 100, 50, learning_rate_range[lr],
                momentum_range[m], batch_sizes[b])
            y_pred_val, _ = forward_prop(X_val, parameters)
            losses[lr][b][m] = cost_loss(y_pred_val, Y_val,
                parameters)


# Hyperparameter search code for stochastic gradient descent

# batch_sizes = [64,128,256]
learning_rate_range = [0.001, 0.005, 0.01, 0.05]
momentum_range = [0.7,0.8,0.9]
losses = np.zeros((len(learning_rate_range),len(momentum_range)
    ))

for lr in range(len(learning_rate_range)):
    for m in range(len(momentum_range)):
        _ , _ , parameters = training_model_sgd_momentum(
            X_train, Y_train, 100, 50, learning_rate_range[lr],
             momentum_range[m])
        y_pred_val, _ = forward_prop(X_val, parameters)
        losses[lr][m] = cost_loss(y_pred_val, Y_val, parameters
            )

losses

cv_min = np.min(l)
ijk_min = np.where(l == cv_min)
ijk_min = tuple([i.item() for i in ijk_min])
opt_learning_rate = learning_rate_range[ijk_min[0]]
opt_batch_size = batch_sizes[ijk_min[1]]
opt_momentum = momentum_range[ijk_min[1]]
print(ijk_min)

print(opt_learning_rate, opt_batch_size, opt_momentum)
```

3.5 Model shootout

3.5.1 - Implement a "vanilla" perceptron (our baseline model) and train on the training set, using the validation set to determine when to stop training:

(Chandra, 2018)

Reshaping the data:

```
X = X_train.reshape(784,12000)
Y = Y_train.reshape(1,12000)
```

Defining functions we will need during training:

```
def accuracy(y_true, y_pred):
    sim = 0
    for k in range(y_true.shape[1]):
        if y_true[0][k] == y_pred[0][k]:
            sim += 1

    return (sim/y_true.shape[1])
```

Training

```
epochs = 100
weights = np.zeros(X.shape[0]).reshape(1,-1)
biases = 0
accuracies = np.zeros((epochs,2))
losses = np.zeros((epochs,2))
training_accuracy = []
validation_accuracy = []

for i in range(epochs):
    for j in range(X.shape[1]):
        X1 = X[:,j].reshape(784,1)
        y_hat = weights @ X1 + biases
        if y_hat < 0 and Y[:,j] == 1:
            weights = weights + X1.T
            biases = biases
        elif y_hat >= 0 and Y[:,j] == 0:
            weights = weights - X1.T
            biases = biases

    pred_train = []
    for k in range(Y_train.shape[1]):
        y_pred = weights @ X_train[:,k] + biases
        if y_pred >= 0:
            pred_train.append(1)
        else:
            pred_train.append(0)

    preds = np.array(pred_train).reshape(1,12000)
    training_accuracy.append(accuracy(Y_train, preds))


    pred_val = []
    c = 0
    for l in range(Y_val.shape[1]):
        y_pred = weights @ X_val[:,l] + biases
        if y_pred >= 0:
            pred_val.append(1)
```

```
        else:
            pred_val.append(0)

    preds = np.array(pred_val).reshape(1,1000)
    validation_accuracy.append(accuracy(Y_val, preds))


best_epoch = len(validation_accuracy) - np.argmax(
   validation_accuracy[::-1]) - 1

print('Training accuracy => ', training_accuracy[best_epoch])
print('Validation accuracy =>',validation_accuracy[best_epoch])

#vanilla perceptron prediction - validation accuracy
pred_val = []
for i in range(Y_val.shape[1]):
    y_pred = weights @ X_val[:,i] + biases
    if y_pred >= 0:
        pred_val.append(1)
    else:
        pred_val.append(0)

preds = np.array(pred_val).reshape(1,1000)

print('Validation Accuracy: ', accuracy(Y_val, preds) * 100, '%
   ')

#vanilla perceptron prediction - train accuracy
pred_train = []
for i in range(Y_train.shape[1]):
    y_pred = weights @ X_train[:,i] + biases
    if y_pred >= 0:
        pred_train.append(1)
    else:
        pred_train.append(0)

preds = np.array(pred_train).reshape(1,12000)
accuracy(Y_train, preds)

print('Train Accuracy: ', accuracy(Y_train, preds) * 100, '%')
```

(after 6 epochs) Validation Accuracy: 97.6% Training Accuracy: 96.9%

Question 3.5.2 - From Figure 21 it is clear that validation peaks at around epoch 6, where the training accuracy is 96.9% and the validation accuracy is 97.6%. The code above found the maximum validation accuracy and returned the epoch for which it occurred, it was observed that the accuracy did not improve after epoch 6. The model was trained for 100 epochs but it was clear that the validation accuracy did not increase beyond epoch 6, for this reason it is appropriate to stop training at around 50 epochs.

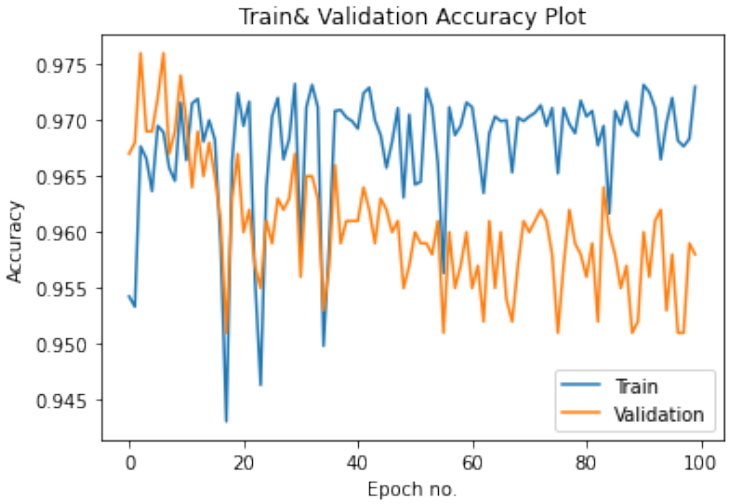

Figure 19: Vanilla Perceptron Accuracy Plot

Question 3.5.3 and 3.5.4 - Previously the best model was the Mini-Batch Gradient Descent with Momentum on the 2-layer MLP. In order to outperform the baseline model the Mini-Batch Gradient Descent with Momentum model was applied again with the optimal hyperparameters. The hyperparameters for the best model were: learning rate = 0.05, Momentum = 0.8 and Batch Size = 64. Questions 3.5.5 and 3.5.6 demonstrate the plots for this.

To approach this task the vanilla perceptron model was ran, it was trained and the best validation accuracy from training was 97.6% was kept as the baseline. In order to improve upon this the baseline was compared with the validation accuracies of the 6 existing models (from question 3.4) that had the 'good' hyperparemters applied. It was discovered that the Stochastic and Mini-Batch Gradient Descent models with Momentum had comparable acuracy with the vanilla perceptron validation accuracy. The Mini-Batch Gradient Descent model with momentum had the highest validation accuracy, so this model was used to try to outperform the baseline.

Question 3.5.5 and 3.5.6 -

The best model was the Mini-Batch Gradient Descent with Momentum model, this was able to achieve the highest validation accuracy and the lowest validation loss.

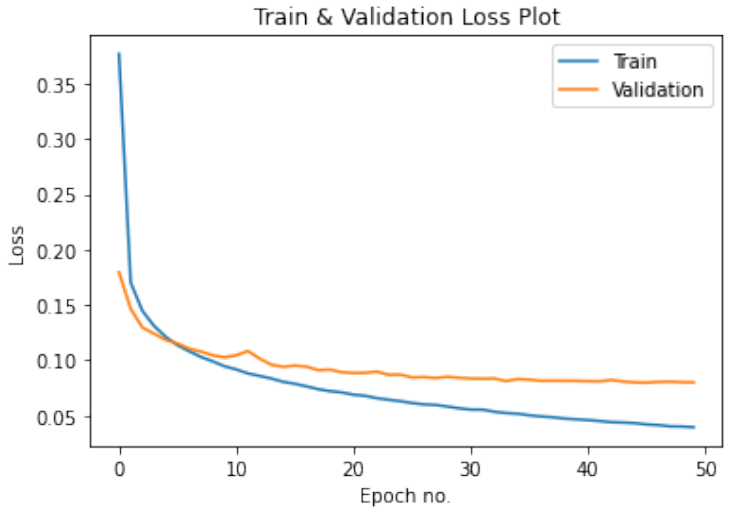

Figure 20: Mini-Batch Gradient Descent Loss Plot (with Momentum)

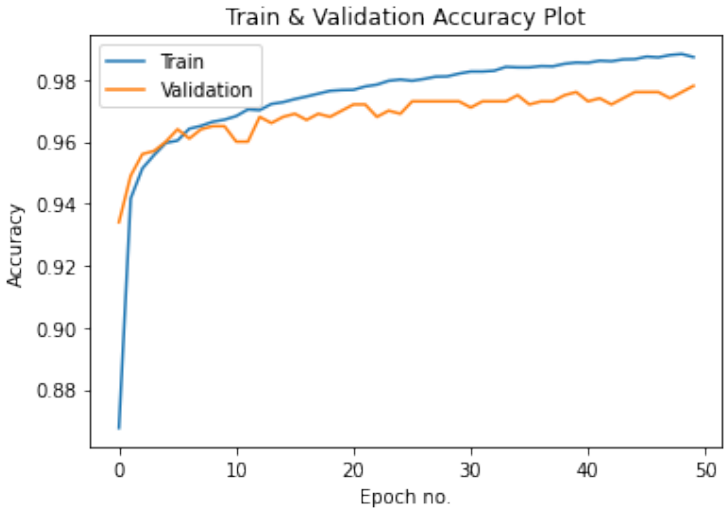

Figure 21: Mini-Batch Gradient Descent Accuracy Plot (with Momentum)

Question 3.5.7 -

The highest accuracy, for the best model, on the validation set was obtained at epoch 49. At this epoch the training accuracy was 98.7% and the validation accuracy was 97.8%.

References

Pranav Budhwant. A beginner's guide to deriving and implementing backpropagation. 2018. URL https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536\.

Akshay L Chandra. Perceptron learning algorithm: A graphical explanation of why it works. 2018. URL https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975.

Andrew Ng. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. URL https://www.coursera.org/learn/deep-neural-network#syllabus\.