COMP0090 Intro to Deep Learning Assignment 1 - Team RealDeep

Toby Drane
ucabtdr@ucl.ac.uk

Ravi Patel
ucabrp1@ucl.ac.uk

Udi Ibgui
zcemyib@ucl.ac.uk

Louis Prosser
ucabljp@ucl.ac.uk

Daniel May
ucabdd3@ucl.ac.uk

November 4, 2020

- Toby Drane took primary responsibility for question 4. He also contributed to helping with the adoption of the momentum and learning methods for a multi-layer perceptron.
- Ravi Patel took primary responsibility for question 5. He also wrote code for data preparation, and generic functions for randomising data, early stopping, and plotting. He helped with aspects of questions 1 and 4.
- Udi Ibgui took primary responsibility for question 3. The code he wrote for the 3-layer MLP in question 3 was adapted by Toby and Ravi for MLPs in questions 4 and 5.
- Louis Prosser took primary responsibility for question 1. He also contributed to the implementation of the 'vanilla' perceptron algorithm and helped with general formatting issues.
- Daniel May took primary responsibility for question 2. He also contributed by helping to adapt his code for finite differences to question 3.

Contents

0 Data Preparation

In order to complete the assignment, it is first necessary to perform some data preparation. This includes importing libraries, flattening the data and initialising some frequently used variables. All code is written in Python.

0.1 Importing libraries

```python
# import libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
```

0.2 Loading the dataset

```python
# load dataset

trainxs = np.load(f"{path}/fashion-train-imgs.npz")
trainys = np.load(f"{path}/fashion-train-labels.npz")
devxs   = np.load(f"{path}/fashion-dev-imgs.npz")
devys   = np.load(f"{path}/fashion-dev-labels.npz")
testxs  = np.load(f"{path}/fashion-test-imgs.npz")
testys = np.load(f"{path}/fashion-test-labels.npz")
```

0.3 Reshaping the data

```python
# reshape data (flatten images to 1D)

def flatten_2D_images(nparray):
  return np.reshape(nparray, (nparray.shape[0]*nparray.shape[1], nparray.shape[2]))

trainxs_flat = flatten_2D_images(trainxs)
devxs_flat = flatten_2D_images(devxs)
testxs_flat = flatten_2D_images(testxs)

print(f"Original train data shape: {trainxs.shape}")
print(f"Flattened train data shape: {trainxs_flat.shape}")
print(f"Original dev data shape: {devxs.shape}")
print(f"Flattened dev data shape: {devxs_flat.shape}")
print(f"Original test data shape: {testxs.shape}")
print(f"Flattened test data shape: {testxs_flat.shape}")
```

```
Original train data shape: (28, 28, 12000)
Flattened train data shape: (784, 12000)
Original dev data shape: (28, 28, 1000)
Flattened dev data shape: (784, 1000)
Original test data shape: (28, 28, 1000)
Flattened test data shape: (784, 1000)
```

0.4 Useful values

```python
# number of examples in each dataset

train_n = trainxs_flat.shape[1]
dev_n = devxs_flat.shape[1]
test_n = testxs_flat.shape[1]
```

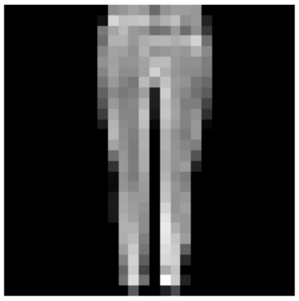
```
# number of dimesions in each image

dims = trainxs_flat.shape[0]

# pixel value range

pixel_value_max = np.amax(trainxs_flat)
pixel_value_min = np.amin(trainxs_flat)
print(f"Max pixel value: {pixel_value_max}")
print(f"Min pixel value: {pixel_value_min}")
```

```
Max pixel value: 1.0
Min pixel value: 0.0
```

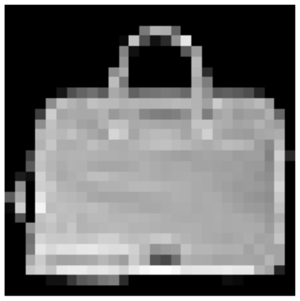
0.5 Data exploration

It is also very helpful to examine some examples from the dataset in order to get a better idea of what the data actually looks like and what the models are meant to achieve.

```
# display an example image with label

examples_to_show = list(range(1000,1002))

for example in examples_to_show:
    plt.imshow(trainxs[:, :, example].T, cmap='gray')
    plt.axis('off')
    plt.show()
    print(f"Y Label: {trainys[example]}")
```



Y label: 0



Y label: 1

1 Question 1: Voted Perceptron Variant

1.1 Memory-efficient voted perceptron variant

The algorithm below (Algorithm 1) is functionally equivalent to the algorithm stated in the question but with memory complexity $\mathcal{O}(1)$ as opposed to $\mathcal{O}(k)$:

Algorithm 1: Averaged Perceptron

```
Data: $\mathcal{D} \coloneqq \{(\boldsymbol{x_1}, y_1), ..., (\boldsymbol{x_n}, y_n)\}$
Result: $\boldsymbol{T} \coloneqq \sum_{i:=1}^{k} c_i \boldsymbol{w_i}$, $B \coloneqq \sum_{i:=1}^{k} c_i b_i$
$\boldsymbol{w} \coloneqq \boldsymbol{0^d}$
$b \coloneqq 0$
$\boldsymbol{T} \coloneqq \boldsymbol{0^d}$
$B \coloneqq 0$
while not converged do
  shuffle dataset
  for $i \coloneqq 1, .., n$ do
    $\hat{y} \coloneqq \begin{cases} 1 & \text{if } \boldsymbol{x_i} \cdot \boldsymbol{w} + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$
    if $\hat{y} = y$ then
      $\boldsymbol{T} \coloneqq \boldsymbol{T} + \boldsymbol{w}$
      $B \coloneqq B + b$
    else
      $\boldsymbol{w} \coloneqq \boldsymbol{w} + (y - \hat{y})\, \boldsymbol{x_i}$
      $b \coloneqq b + (y - \hat{y})$
      $\boldsymbol{T} \coloneqq \boldsymbol{T} + \boldsymbol{w}$
      $B \coloneqq B + b$
    end if
  end for
end while
```

In this improved algorithm, $\boldsymbol{T}$ represents the overall sum of the weight vectors applied during training, with each contribution to the sum scaled by the number of correct classifications (the c_i's) achieved by that weight vector. Similarly, B represents the sum of the biases, scaled by the same factors. $\boldsymbol{w}$ and b are used to temporarily store each weight vector and bias respectively which are added to $\boldsymbol{T}$ and B upon each correct classification and then updated as soon as an incorrect classification is made. As in the algorithm in the question, we denote k to be the number of times a sample is misclassified during training and hence is equal to the number of weight vectors included in the final sum $\boldsymbol{T}$. We shuffle the dataset before each epoch to avoid issues related to ordering in the original dataset, that might affect generalisation.

1.2 Proof of functional equivalence

The algorithm in the question stores each weight vector along with its associated number of correct classifications during training. However, this is memory-inefficient as it is possible to instead only record a cumulative sum of each weight vector multiplied by the number of correct classifications it has taken part in during training as well as a cumulative sum of the bias terms multiplied by the same factors and hence the memory required to compute this new algorithm does not scale with the number of misclassifications.

Once trained, this algorithm performs predictions as follows:

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \left[\left(\sum_{i:=1}^{k} c_i \boldsymbol{w_i}\right) \cdot \boldsymbol{x} + \left(\sum_{i:=1}^{k} c_i b_i\right)\right] \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We can prove that this is functionally equivalent to the algorithm stated in the question by using properties of the dot product. First, the distributive property of the dot product states that, for any real vectors $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$,

$$\boldsymbol{x} \cdot (\boldsymbol{y} + \boldsymbol{z}) = \boldsymbol{x} \cdot \boldsymbol{y} + \boldsymbol{x} \cdot \boldsymbol{z} \tag{2}$$

And the scalar multiplication property of the dot product states that for real vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ and real scalars c_1 and c_2,

$$(c_1\boldsymbol{x}) \cdot (c_2\boldsymbol{y}) = c_1 c_2 \left(\boldsymbol{x} \cdot \boldsymbol{y}\right) \tag{3}$$

Therefore, using both these properties and ignoring bias terms, the output of our algorithm above is equivalent to the output of the algorithm in the question:

$$\boldsymbol{T} \cdot \boldsymbol{x} = \left(\sum_{i:=1}^{k} c_i \boldsymbol{w_i}\right) \cdot \boldsymbol{x} = \sum_{i:=1}^{k} c_i \left(\boldsymbol{w_i} \cdot \boldsymbol{x}\right) \tag{4}$$

1.3 Algorithm implementation

We now implement this algorithm on the pre-prepared subset of the “Fashion-MNIST” dataset.

```
MAX_NUMBER_OF_EPOCHS = 100
train_accuracy_per_epoch = []
dev_accuracy_per_epoch = []

# initialise variables

w = np.zeros(dims)
b = 0
T = np.zeros(dims)
B = 0
best_dev_accuracy = 0

for j in range(MAX_NUMBER_OF_EPOCHS):

    # run an epoch

    train_errors = 0

    # shuffle the data

    shuffled_indices = [*range(0,train_n,1)]
    np.random.shuffle(shuffled_indices)

    # train the data

    for i in shuffled_indices:

        y_hat = 1 if (np.dot(trainxs_flat[:,i],w) + b) >= 0 else 0

        if y_hat == trainys[i]:
            T += w
            B += b
        else:
            w += (trainys[i] - y_hat) * trainxs_flat[:,i]
            b += (trainys[i] - y_hat)
            T += w
            B += b
            train_errors += 1

    train_accuracy_per_epoch.append((train_n-train_errors)/train_n)

    # evaluate the weighted sums of the weights and biases on the validation set
    dev_errors = 0
    for i in range(dev_n):

        dev_pred = 1 if (np.dot(devxs_flat[:,i],T) + B) >= 0 else 0

        if dev_pred != devys[i]:
            dev_errors += 1

    dev_accuracy = (dev_n-dev_errors)/dev_n
    dev_accuracy_per_epoch.append(dev_accuracy)

    # store accuracy, weights, and bias from epoch with best validation accuracy

    if dev_accuracy > best_dev_accuracy:
        best_dev_accuracy = dev_accuracy
        best_dev_T = T
        best_dev_B = B
        best_weights_epoch = j

    # early stopping criteria

    if len(train_accuracy_per_epoch) >= 30:
        if (np.mean(train_accuracy_per_epoch[-15:]) -
        ↪  np.mean(train_accuracy_per_epoch[-30:-15])) < 0 :
            print("break")
            break

# accuracy on the training set for the epoch on which we obtain the highest accuracy on the
↪  validation set

best_train_accuracy = train_accuracy_per_epoch[best_weights_epoch]
```

1.4 Train model to convergence on training set

In order to train the model to convergence, we implemented a maximum number of epochs as well as an early stopping criteria which was triggered if the mean accuracy of the last 15 epochs on the training set was less than the mean accuracy of the preceding 15 epochs on the training set. Averaging over multiple epochs allows the algorithm to run for a suitable minimum length of time and accounts for the frenetic nature of the accuracy graphs for the perceptron.

We set the maximum number of epochs to be 100 and our model ran for 91 epochs when the early stopping criteria was triggered. The training accuracy of the model was 0.968 at this point and the validation accuracy was 0.972.

1.5 Voted perceptron variant: Accuracy plot

A plot of the accuracy of this model on the training and validation sets can be found in figure 1.

```
# Training and validation accuracy plot for averaged perceptron

plt.plot(train_accuracy_per_epoch, color='k', linestyle='-')
plt.plot(dev_accuracy_per_epoch, color='r', linestyle='-')
plt.title(f'Averaged Perceptron: Model Accuracy',  color='k')
plt.ylabel('Accuracy',  color='k')
plt.xlabel('Epoch',  color='k')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.tick_params(colors='k')
plt.xlim(0)
plt.savefig("averaged_perceptron_accuracy_plot.png", dpi=300)
plt.show()
```

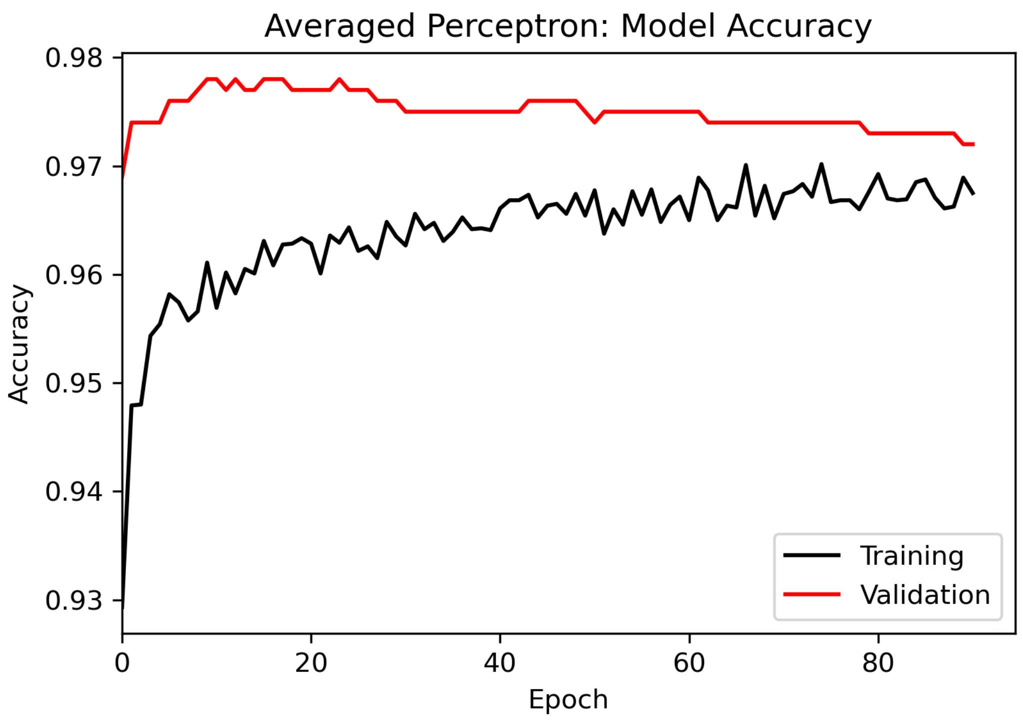


Figure 1: Training and validation accuracy by epoch for the averaged perceptron algorithm

1.6 Voted perceptron variant: Epoch with highest validation set accuracy

The highest accuracy on the validation set was achieved at epoch 9. The values for the accuracy on the training and validation sets for this epoch are shown in table 1.

Set	Accuracy
Training	0.961
Validation	0.978

Table 1: The training and validation accuracy at epoch 9.

2 Question 2: Logistic Regression

2.1 Analytical gradients

Our goal is to minimize the mean-squared loss function for logistic regression, given by

$$\begin{aligned} L &= \frac{1}{n}\frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{2} \\ &= \frac{1}{n}\frac{\sum_{i=1}^{n}(y_i - \sigma(w'x_i + b))^2}{2} \end{aligned} \tag{5}$$

where $\sigma(x) = \dfrac{1}{1+e^{-x}}$.

The gradient with respect to the weight parameter is given by

$$\begin{aligned} \frac{\partial L}{\partial w} &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(w'x_i + b)) \cdot \frac{\partial \sigma(w'x_i + b)}{\partial w} \\ &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(w'x_i + b)) \cdot \sigma(w'x_i + b)(1 - \sigma(w'x_i + b))x_i \\ &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i}) \cdot \hat{y_i}(1 - \hat{y_i})x_i \end{aligned} \tag{6}$$

The gradient with respect to the bias parameter is given by

$$\begin{aligned} \frac{\partial L}{\partial b} &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(w'x_i + b)) \cdot \frac{\partial \sigma(w'x_i + b)}{\partial b} \\ &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \sigma(w'x_i + b)) \cdot \sigma(w'x_i + b)(1 - \sigma(w'x_i + b)) \\ &= -\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i}) \cdot \hat{y_i}(1 - \hat{y_i}) \end{aligned} \tag{7}$$

2.2 Checking gradients with finite differences

It is possible to verify these analytical gradients for the square loss using finite differences. We do so below for three samples of the training data, choosing $epsilon = 10^{-5}$ in the finite difference equation.

We generate weights at random and set the initial bias to zero, then calculate the partial gradients for each of these parameters, analytically and using finite differences. We print the values and confirm by eye that the methods concur in their outputs. More formally, we then take the absolute difference between the resulting vectors, and confirm that they are individually each less than a tiny value — we choose e^{-10}.

A sample of the printed output is shown below, along with the code.

```
Training set index 38, y is 0, y_hat is 0.49975358489866584
* Bias difference: 5.577718842353363e-12
* Bias difference less than e^(-10)? True
* Sample weight differences: [7.499923533778319e-13, 3.4056073933141917e-12,
↪  1.0628234403675663e-12, 3.0158688046899584e-12, 5.759052956744171e-12,
↪  2.2441215552504445e-12, 1.862121568052544e-12, 3.48172879416353e-13,
↪  4.729272529147011e-13, 1.2954429196021522e-12]
* Weight differences all less than e^(-10)? True

Training set index 39, y is 1, y_hat is 0.5076590989007915
* Bias difference: 4.633793349029247e-14
* Bias difference less than e^(-10)? True
* Sample weight differences: [1.9941687190438984e-12, 1.9941687190438984e-12,
↪  1.9941687190438984e-12, 1.9941687190438984e-12, 1.5322413476903662e-12,
↪  5.287485727034635e-12, 1.8119464262333906e-12, 1.895691936759647e-12,
↪  5.2215176626901894e-14, 1.8490625697253904e-12]
* Weight differences all less than e^(-10)? True

Training set index 40, y is 0, y_hat is 0.4434366519407762
* Bias difference: 4.844902257161721e-12
* Bias difference less than e^(-10)? True
* Sample weight differences: [9.330210215541257e-14, 9.330210215541257e-14,
↪  9.330210215541257e-14, 9.330210215541257e-14, 9.330210215541257e-14,
↪  9.330210215541257e-14, 9.330210215541257e-14, 9.330210215541257e-14,
↪  9.330210215541257e-14, 9.330210215541257e-14]
* Weight differences all less than e^(-10)? True
```

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

def f(w, b, x):
  return sigmoid(np.dot(w, x) + b)

def loss(w, b, x, y):
  """Compute the mean-squared loss"""
  return -(1/2)*(y-f(w, b, x))**2

def df_dw_fd(w, b, x, y, epsilon = 10**(-5)):
  """Compute the gradient with respect to the weights using finite differences"""
  gradients = []
  for i in range(0, w.shape[0]):
    w_i = w[i]
    w[i] += epsilon / 2.0
    lhs = loss(w, b, x, y)
    w[i] = w_i
    w[i] -= epsilon / 2.0
    rhs = loss(w, b, x, y)
    w[i] = w_i
    df_dw_i = (lhs - rhs) / epsilon
    gradients.append(df_dw_i)
  return gradients

def df_db_fd(w, b, x, y, epsilon = 10**(-5)):
  """Compute the gradient with respect to the bias using finite differences"""
  lhs = loss(w, b + epsilon / 2.0, x, y)
  rhs = loss(w, b - epsilon / 2.0, x, y)
  return (lhs - rhs) / epsilon

def df_dw_analytical(w, b, x, y):
  """Compute the gradient with respect to the weights analytically"""
  y_hat = f(w, b, x)
  return -(y - y_hat)*y_hat*(1-y_hat)*x

def df_db_analytical(w, b, x, y):
  """Compute the gradient with respect to the weights analytically"""
  y_hat = f(w, b, x)
  return -(y - y_hat)*y_hat*(1-y_hat)

training_samples = list(range(38,41))
```

```
for i in training_samples:
  x, y = trainxs_flat[:,i], trainys[i]

  w = np.random.normal(0, 1, dims)/100
  b = 0.0

  fd_bias = df_db_fd(w, b, x, y)
  fd_weights = df_dw_fd(w, b, x, y)

  analytical_bias = df_db_analytical(w, b, x, y)
  analytical_weights = df_dw_analytical(w, b, x, y)

  absolute_bias_difference = np.absolute(fd_bias - analytical_bias)
  absolute_weight_differences = np.absolute(fd_weights - analytical_weights)

  print(f"Training set index {i}, y is {y}, y_hat is {f(w, b, x)}")
  print('Finite differences estimates:')
  print(f"* Bias: {fd_bias}")
  print(f"* Weights: {fd_weights}")
  print('Analytical gradient estimates:')
  print(f"* Bias: {analytical_bias}")
  print(f"* Weights: {analytical_weights.tolist()}")
  print('Differences')
  print(f"* Bias difference: {absolute_bias_difference}")
  print(f"* Bias difference less than e^(-10)? {absolute_bias_difference < np.e**(-10)}")
  print(f"* Sample weight differences: {np.extract(x > 0,
  ↪  absolute_weight_differences)[0:10].tolist()}")
  print(f"* Weight differences all less than e^(-10)? {(absolute_weight_differences <
  ↪  np.e**(-10)).all()}")
```

2.3 Algorithm implementation

```
def predict(w, b, x):
  """Predict the class of a feature vector x given some weights and a bias"""
  return 1 if f(w, b, x) >= 0.5 else 0

def accuracy(xs, ys, w, b):
  """Compute the accuracy for a set of feature vectors and their targets given
  some weights and a bias"""
  n = xs.shape[1]

  correct = 0
  for i in range(n):
    x = xs[:,i]
    y = ys[i]
    if predict(w, b, x) == y:
      correct += 1
  return correct/n

train_loss_per_epoch = []
train_accuracy_per_epoch = []
dev_accuracy_per_epoch = []

def train_logistic_regression(training_xs, training_ys, max_epochs=50, learning_rate=0.01):
  """Train a logistic regression model for a set of feature vectors and their
  targets, for a given learning rate, and a maximum number of epochs unless
  it meets the convergence criterion before this epoch"""
  assert(training_xs.shape[1] == training_ys.shape[0])

  train_n = training_xs.shape[1]
  dims = training_xs.shape[0]
  w = np.random.normal(0, 1, dims)/100
  b = 0.0

  for epoch in range(max_epochs):
    grad_w = np.zeros(dims)
    grad_b = 0

    mean_squared_loss = 0

    shuffled_indices = [*range(0,train_n,1)]
    np.random.shuffle(shuffled_indices)

    for i in shuffled_indices:
      x, y = training_xs[:,i], training_ys[i]
      y_hat = f(w, b, x)

      grad_w -= (y - y_hat) * y_hat * (1 - y_hat) * x
      grad_b -= (y - y_hat) * y_hat * (1 - y_hat)

      mean_squared_loss += (y - y_hat)**2/(2*train_n)

    grad_w /= train_n
    grad_b /= train_n

    w += learning_rate * grad_w
    b += learning_rate * grad_b

    train_accuracy = accuracy(training_xs, training_ys, w, b)
    dev_accuracy = accuracy(devxs_flat, devys, w, b)

    train_loss_per_epoch.append(mean_squared_loss)
    train_accuracy_per_epoch.append(train_accuracy)
    dev_accuracy_per_epoch.append(dev_accuracy)

    # Early stopping (convergence criterion)
    if len(train_accuracy_per_epoch) >= 30:
      if np.mean(train_accuracy_per_epoch[-15:]) - np.mean(train_accuracy_per_epoch[-30:-15])
      ↪  < 0.001:
          print(f"Break on epoch {epoch}")
          break

  return w, b
```

2.4 Training model to convergence

We trained the logistic regression model on the training set with learning rates of 1, 0.1, and 0.01, to compare the number of epochs their loss and accuracy would take to flatten out, and the values of these when they do.

We implemented early stopping, where training stopped if the mean accuracy from the most recent 15 epochs was less than 0.1% greater than the mean accuracy for the preceding 15 epochs. The training had to proceed for a minimum of 30 epochs before early stopping could be triggered.

With a learning rate of 1, the larger weight updates cause the loss and accuracy to jump around at the beginning of training, but the curve flattens out in significantly fewer epochs and with a greater training set accuracy (96.0%) when it reaches the early stopping point than with learning rates 0.1 (94.7%) and 0.01 (92.8%).

The learning rate of 0.01 took 753 epochs to stop, at which point it had the lowest accuracy — versus 111 and 183 respectively for rates 1 and 0.01 — suggesting it may be lower than necessary in our case. It had its highest validation set accuracy on its final epoch (753), suggesting — along with its graphs — that it was still improving rather than fully flattening out.

The loss and accuracy curves for a rate of 0.1 were smoother than for rate 1, and it reached the early stopping point in only 1.5 times the number of epochs, and around 4 times fewer than with a learning rate of 0.01, suggesting it may be a reasonable middle ground.

```
MAX_EPOCHS = 1000
LEARNING_RATE = 0.1
w_hat, b_hat = train_logistic_regression(trainxs_flat, trainys, learning_rate=LEARNING_RATE,
↪  max_epochs=MAX_EPOCHS)
```

2.5 Logistic regression: Loss plot

Plots of the loss of this model on the training set can be found in figure 2, for learning rates 1, 0.1, and 0.01. See Subsection 2.4 for a discussion of these plots.

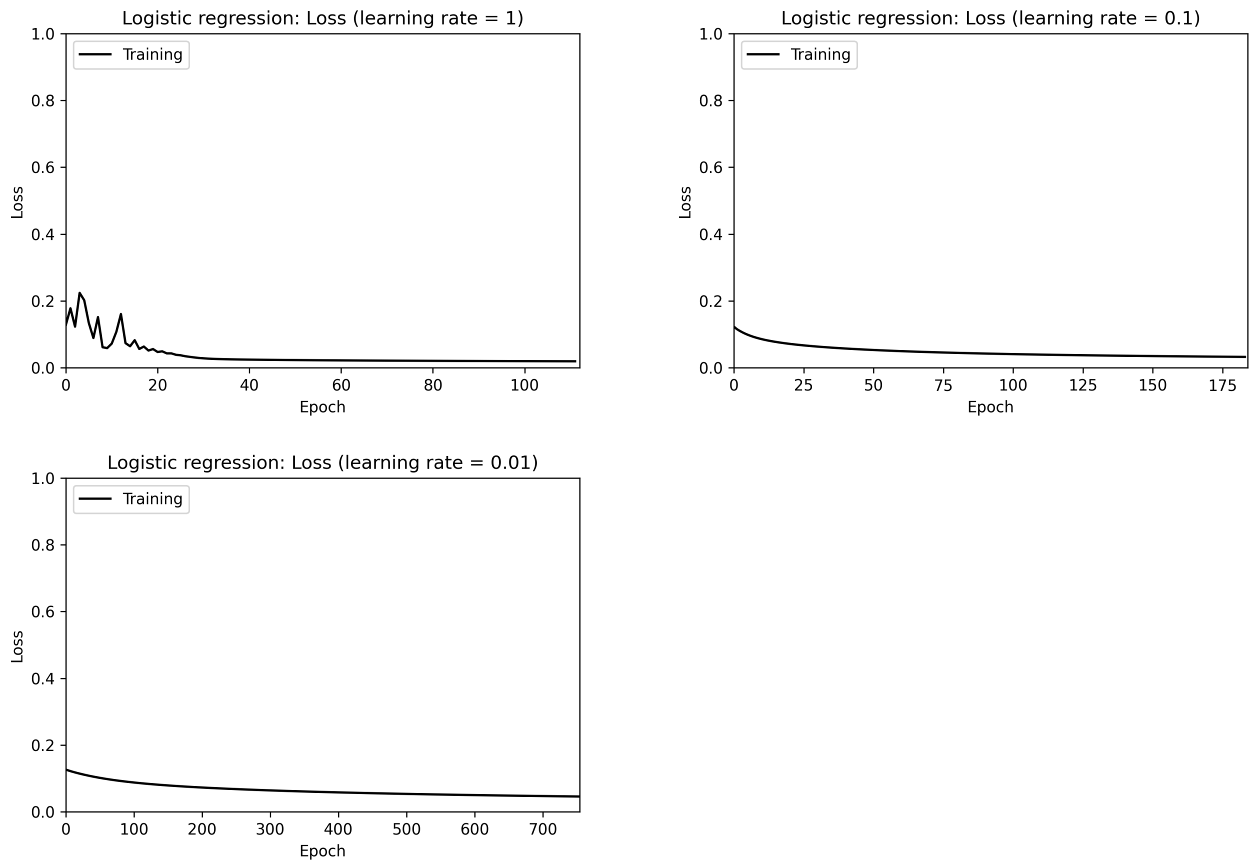


Figure 2: Training loss by epoch for the mean squared-loss logistic regression algorithm for learning rates = 1, 0.1, and 0.01

```
plt.plot(train_loss_per_epoch, color='k', linestyle='-')
plt.title(f'Logistic regression: Loss (learning rate = {LEARNING_RATE})',  color='k')
plt.ylabel('Loss',  color='k')
plt.xlabel('Epoch',  color='k')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.tick_params(colors='k')
plt.xlim(0, len(train_loss_per_epoch))
plt.ylim(0, 1)
plt.savefig(f"q2_loss_{LEARNING_RATE}.png", dpi=300)
plt.show()
```

2.6 Logistic regression: Accuracy plot

Plots of the accuracy of this model on the training and validation sets can be found in figure 3, for learning rates 1, 0.1, and 0.01. We can see that each model performs similarly — even slightly better — on the validation sets as on the training set, which is a promising sign if they are to be used to classify production data. See Subsection 2.4 for a further discussion of these plots.

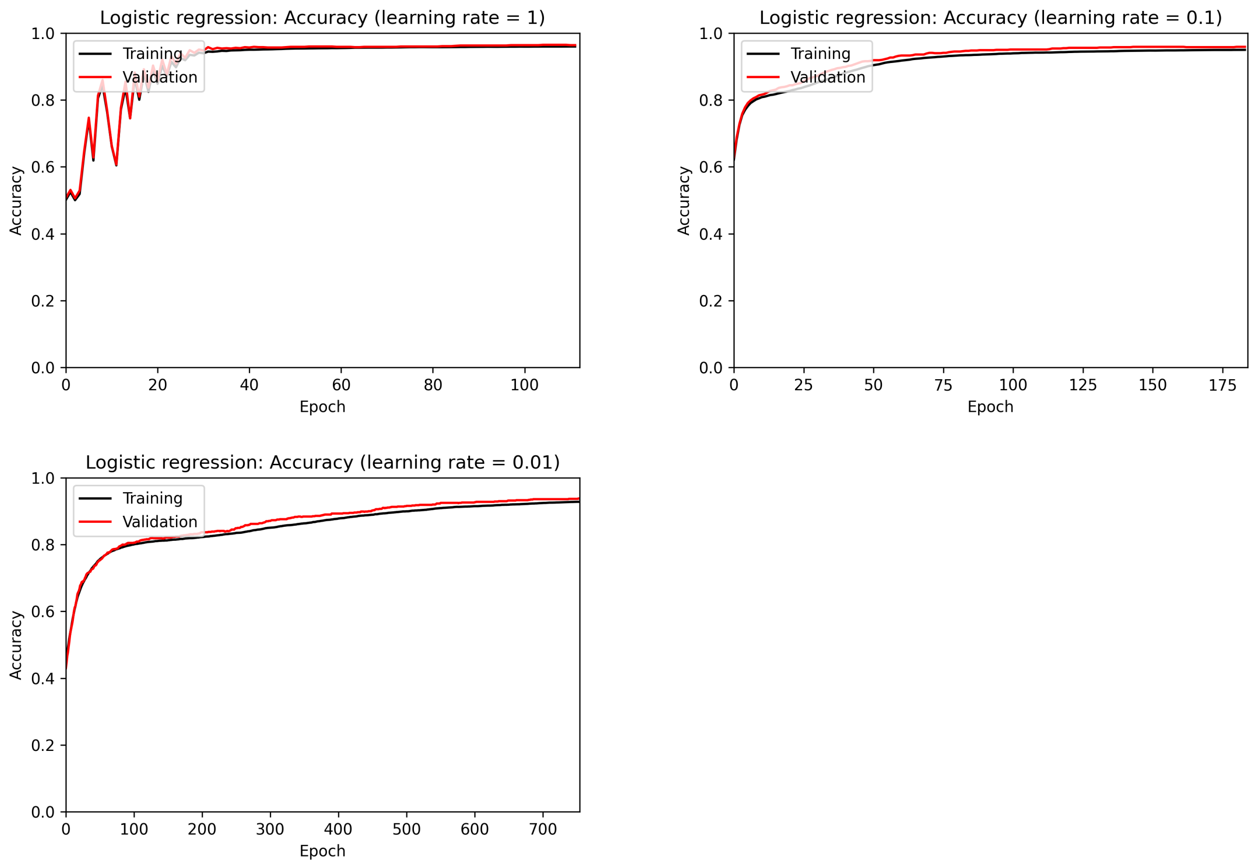


Figure 3: Training and validation accuracy by epoch for the mean squared-loss logistic regression algorithm for learning rates = 1, 0.1, and 0.01

```
plt.plot(train_accuracy_per_epoch, color='k', linestyle='-')
plt.plot(dev_accuracy_per_epoch, color='r', linestyle='-')
plt.title(f'Logistic regression: Accuracy (learning rate = {LEARNING_RATE})',  color='k')
plt.ylabel('Accuracy',  color='k')
plt.xlabel('Epoch',  color='k')
plt.legend(['Training', 'Validation'], loc='upper left')
plt.tick_params(colors='k')
plt.xlim(0, len(train_accuracy_per_epoch))
plt.ylim(0, 1)
plt.savefig(f"q2_accuracy_{LEARNING_RATE}.png", dpi=300)
plt.show()
```

2.7 Logistic regression: Epoch with highest validation set accuracy

Choosing a learning rate of 0.1, the algorithm reached its early stopping point at epoch 183, and reached its highest validation set accuracy on epoch 144. The training and validation accuracy at this epoch are shown in Table 2.

Set	Accuracy
Training	0.947
Validation	0.959

Table 2: The training and validation accuracy at epoch 144.

For learning rates 1 and 0.01 respectively, the highest validation accuracy was reached on epoch 105 and 753, with validation accuracy of 0.965 and 0.938, and training accuracy 0.960 and 0.928.

```
index_of_maximum = np.argmax(np.array(dev_accuracy_per_epoch))
epoch = index_of_maximum + 1
print(f"Epoch: {epoch}")
print(f"Training accuracy on epoch {epoch}: {train_accuracy_per_epoch[index_of_maximum]}")
print(f"Validation accuracy on epoch {epoch}: {dev_accuracy_per_epoch[index_of_maximum]}")
```

3 Question 3: 3-layer Multilayer Perceptron

3.1 Analytical gradients

Given the function;

$$f(x) = \sigma(W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3) \tag{1}$$

We are asked to calculate the derivatives of the loss function in terms of W_1, W_2, W_3, b_1, b_2, and b_3. To do this, the chain rule will be used, with respect to partial derivatives. The loss function calculates the difference between the algorithm's output, $\hat{y}$, and the desired output, y. Here, the negative log-likelihood loss Function will be used. This is given as follows;

$$L(\hat{y}, y) = -(y \log[\hat{y}] + (1 - y) \log[1 - \hat{y}]) \tag{2}$$

Function (1) is a combination of linear functions which are then input into activation functions. In this particular case, the sigmoid activation function is used. For simplicity, we rewrite the different components;

$$\begin{aligned}
z^{[1]} &= W_1x + b_1 \\
a^{[1]} &= \sigma(z^{[1]}) \\
z^{[2]} &= W_2a^{[1]} + b_2 \\
a^{[2]} &= \sigma(z^{[2]}) \\
z^{[3]} &= W_3a^{[2]} + b_3 \\
a^{[3]} &= \hat{y} = \sigma(z^{[3]})
\end{aligned} \tag{3}$$

The partial derivative of each component is then calculated:

$$\begin{aligned}
\frac{\partial L}{\partial a^{[3]}} &= \frac{a^{[3]} - y}{a^{[3]}(1 - a^{[3]})} \\
\frac{\partial a^{[3]}}{\partial z^{[3]}} &= a^{[3]}(1 - a^{[3]}) \\
\frac{\partial z^{[3]}}{\partial a^{[2]}} &= W_3 \\
\frac{\partial a^{[2]}}{\partial z^{[2]}} &= a^{[2]}(1 - a^{[2]}) \\
\frac{\partial z^{[2]}}{\partial a^{[1]}} &= W_2 \\
\frac{\partial a^{[1]}}{\partial z^{[1]}} &= a^{[1]}(1 - a^{[1]}) \\
\frac{\partial z^{[l]}}{\partial W_l} &= a^l \\
\frac{\partial z^{[l]}}{\partial b_l} &= 1
\end{aligned} \tag{4}$$

The gradients of the loss with respect to each parameter can now be computed using the the chain rule:

$$
\begin{aligned}
\frac{\partial L}{\partial W_3} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial W_3} = a^{[2]}(a^{[3]}-y)\\
\frac{\partial L}{\partial b_3} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial b_3} = (a^{[3]}-y)\\
\frac{\partial L}{\partial W_2} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial W_2} = W_3(a^{[3]}-y)a^{[2]}(1-a^{[2]})a^{[1]}\\
\frac{\partial L}{\partial b_2} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial b_2} = W_3(a^{[3]}-y)a^{[2]}(1-a^{[2]})\\
\frac{\partial L}{\partial W_1} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial W_1} = W_3W_2(a^{[3]}-y)a^{[2]}(1-a^{[2]})a^{[1]}(1-a^{[1]})x\\
\frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z^{[2]}}\frac{\partial z^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial z^{[1]}}\frac{\partial z^{[1]}}{\partial b_1} = W_3W_2(a^{[3]}-y)a^{[2]}(1-a^{[2]})a^{[1]}(1-a^{[1]})
\end{aligned}
\tag{5}
$$

Taking our equations from (4) and (5), we can form a general derivation for the parameters, such as:

$$\frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}}a^{[l]}(1-a^{[l]}) \tag{6}$$

$$\frac{\partial L}{\partial W_l} = \frac{\partial L}{\partial z^{[l]}}a^{[l-1]} \tag{7}$$

$$\frac{\partial L}{\partial b_l} = \frac{\partial L}{\partial z^{[l]}} \tag{8}$$

3.2 Checking gradients with finite differences

In this part, the gradients for a randomly initialized 3-layer MLP are first calculated. The gradients are obtained by following the derivations obtained in Section 3.1. The code below first initializes weights and biases for an L-layered MLP. It then produces an output, $\hat{y}$, which is compared to the desired output, y using the negative log-likelihood loss function to give a Loss. The Loss is then differentiated with respect to each parameter to give the gradients.

We can use the finite difference method to approximate the correct, exact analytical gradients for each parameter. By checking that the difference between the approximate gradients and those given by the analytical equations above is sufficiently tiny, we gain confidence that the analytical gradients are correct. The code outputs a warning if one of the differences is too large. Below, is the code calling all the functions that were used to complete the gradient checking. The full code of each function is available in Appendix A (for generating the analytical gradients, it is part of A.1, whilst for the finite difference and gradient checking, the code is in A.2).

```
#Initialize parameters, pass them forward, calculate loss and obtain gradients via
↪  backpropagation
parameters = parameters_initialization(layers_dims)
output, both = forward_pass(trainxs_flat, parameters)
loss = loss_fun(output, trainys)
backprop_grads = backprop_loop(output, trainys, both)


#Get finite differences and compare them to gradients
finite_methods = fin_method(parameters, backprop_grads, trainxs_flat, trainys, epsilon =
↪  10**-10)
difs = abs_dif(backprop_grads, finite_methods)
print('The largest difference between the Analytical Gradients and the Finite Differences is:
↪  ', max(difs))
```

```
The largest difference between the Analytical Gradients and the Finite Differences is:
↪  2.9182475596535237e-06
```

Hence the difference between our gradients and the Finite Difference method is very low, indicating our analytical gradients are correct.

3.3 Model implementation

All the functions provided in Appendix A.1 were combined, as can be seen below to create an L-layer MLP (specifically a 3-layer MLP in this question). Ultimately, the function begins by generating weights and biases, corresponding to the size and number of layers. These are passed forward through our given function to calculate an output for each training example. Each of these is compared to the expected output and the loss is calculated using the negative log-likelihood loss. The loss is then used to update the parameters using backpropagation. This process is repeated for a set number of iterations or until a stopping criterion is met. At each iteration, the loss and accuracy for both the training and validation set is computed and recorded. This is then used to generate plots to show how loss and accuracy changes over all the corresponding iterations.

```python
def MLP(X, y, layers_dims, learning_rate, epochs):

    #Keep track of loss and accuracy of each epoch
    loss = []
    accs = []

    dev_loss = []
    dev_accuracies = []

    # Parameters initialisation.
    parameters = parameters_initialization(layers_dims)

    #Run epochs
    for i in range(0, epochs):

        #Forward pass
        output, both = forward_pass(X, parameters)

        #Loss
        los = loss_fun(output, y)

        #Accuracy
        acc = model_accuracy(output, y)

        # Backprop
        grads = backprop_loop(output, y, both)

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        #Record loss and Acuracy
        loss.append(los)
        accs.append(acc)

        # Print the loss every 100 training example
        if i % 100 == 0:
            print ("Loss after Epoch %i: %f" % (i, los))
            print ("Accuracy after Epoch %i: %f" % (i, acc))


        #Calculate Loss and Acuracy on Validation set using current epoch's paramters
        out = forward_pass(devxs_flat, parameters)
        dev_los = loss_fun(out[0], devys)
        dev_accuracy = model_accuracy(out[0], devys)

        #Record Validation Loss and Acuracy
        dev_loss.append(dev_los)
        dev_accuracies.append(dev_accuracy)

        if len(dev_loss) >= 300:
            if (np.mean(dev_loss[-20:]) - np.mean(dev_loss[-40:-20])) > -.001:
                print("Converged or reached max epochs")
                break


    # plot the loss
    plt.plot(np.squeeze(loss), color='k', linestyle='-')
    plt.plot(np.squeeze(dev_loss), color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.title(f"Loss Plot for Training and Validation Set")
    plt.show()

    # plot the accuracy
    plt.plot(np.squeeze(accs), color='k')
    plt.plot(np.squeeze(dev_accuracies), color='r')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.title("Accuracy Plot for Training and Validation Set")
    plt.show()

    return parameters
```

Finally, the model was tested. Initial scoping experiments were performed to identify a suitable number of nodes to include in the hidden layer. It was found that the MLP performed best (in terms of accuracy vs. computation trade-off) with $W_1 = 100 * 28^2$ and $W_2 = 60 * 100$. It then followed that $b_1 = 100, b_2 = 60, W_3 = 1 * 60$ and $b_3 = 1$

```
#Set MLP size and dimensions
layers_dims = [784, 100, 60, 1]

#Calculate optimum paramters given our training set using the MLP. The learning rate was set
↪  to 0.01 and a max iteration of 10000
parameters = MLP(trainxs_flat, trainys, layers_dims, learning_rate = 0.01, epochs=10000)

#Calculate the output of the function on the validation set, given the parameters obtained by
↪  the MLP
out = forward_pass(devxs_flat, parameters)

#Calculate loss and accuracy and print it
loss = loss_fun(out[0], devys)

accuracy = model_accuracy(out[0], devys)

print('Validation Loss:', loss)
print('Validation Accuracy', accuracy)
```

3.4 Training

In order to train the 3 layer MLP (full code in Appendix A.1) to convergence, we implemented a maximum number of epochs as well as an early stopping criteria. The code snippet can be seen below. Stopping was triggered if the mean loss of the last 20 epochs on the validation set did not drop significantly (more than 0.001) compared to the mean validation loss of the preceding 20 epochs.

```
if len(dev_loss) >= 300:
    if (np.mean(dev_loss[-20:]) - np.mean(dev_loss[-40:-20])) > -.001:
        print("Converged or reached max epochs")
        break
```

3.5 3-layer MLP: Loss plot

The plot of loss for the training and validation datasets, for our 3-layer MLP, are shown in Figure 4.

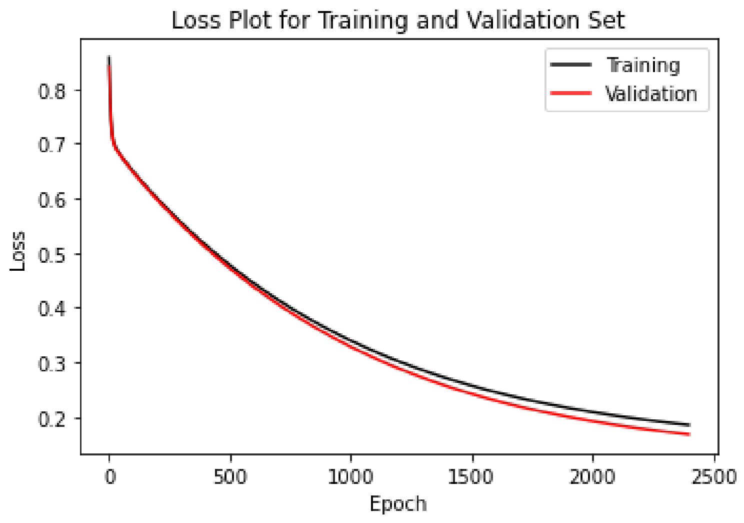


Figure 4: Training and validation loss by epoch for the 3-layer MLP. The steepness of the curve can be seen to have become very shallow.

3.6 3-layer MLP: Accuracy plot

The plot of accuracy for the training and validation datasets, for our 3-layer MLP, are shown in Figure 5.

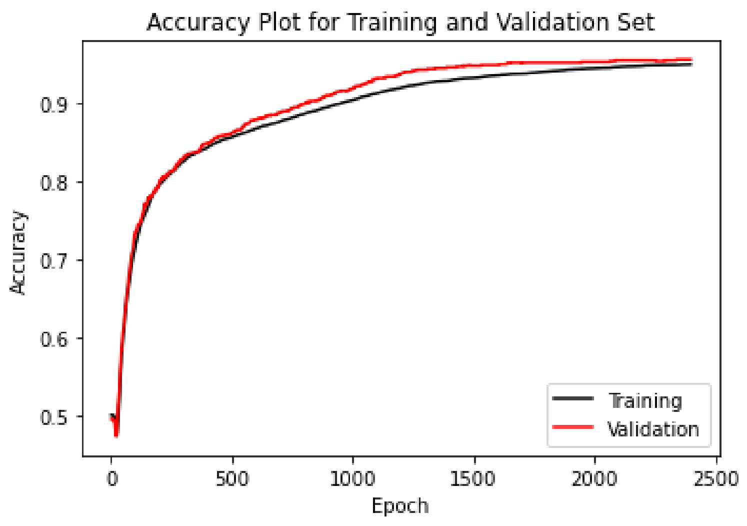


Figure 5: Training and validation accuracy by epoch for the 3-layer MLP. The curve is relatively flat for several iterations.

3.7 3-layer MLP: Accuracy for training and validation set

The 3 layer MLP achieved a highest accuracy of 0.948 on the validation dataset and an accuracy of 0.990 on the training dataset.

Set	Accuracy
Training	0.948
Validation	0.955

Table 3: The training and validation accuracy at the best epoch.

Interestingly, the model's performance on the validation set was better than on the training set, both in terms of accuracy and loss. This was slightly unexpected, as the training data was the data used to adjust the model parameters, whilst the validation set was new and unseen data.

4 Question 4: 2-layer Multilayer Perceptron

4.1 2-layer MLP Network

We are first asked to construct a two-layer neural network, with the structure of the said network defined as:

$$W_1 \in \mathbb{R}^{28^2 \times 3} \tag{8}$$

$$b_1 \in \mathbb{R}^3 \tag{9}$$

$$W_2 \in \mathbb{R}^{1 \times 3} \tag{10}$$

$$b_1 \in \mathbb{R} \tag{11}$$

The full Python implementation for the model can be seen in Appendix B.

4.1.1 Momentum

Momentum is loosely derived from a theory in Physics to accelerate learning. Momentum introduces a 'velocity' term, which is just the previous back propagation value multiplied by some learning coefficient. The algorithm for back-propagation with momentum is seen below:

$$\begin{aligned} V_i &= \lambda V_{i-1} + \alpha \delta f(W_i) \\ W_i &= W_{i-1} - V_i \end{aligned} \tag{12}$$

Where λ is a momentum coefficient and α is the learning rate. Our model implementation for momentum is the following:

```
L = len(parameters) // 2 # number of layers in the neural network. We are dividing by 2
↪  because parameter contains W and b
weight_increment = {}
velocity_new = {}

for l in range(L):
    weight_increment["W"+str(l+1)] = learning_rate * grads["dW" + str(l+1)]
    weight_increment["b"+str(l+1)] = learning_rate * grads["db" + str(l+1)]

    velocity_new["W" + str(l+1)] = momentum_coefficient * velocity_old["W" + str(l+1)] +
    ↪  weight_increment["W" + str(l+1)]
    velocity_new["b" + str(l+1)] = momentum_coefficient * velocity_old["b" + str(l+1)] +
    ↪  weight_increment["b" + str(l+1)]

    # With momentum
    parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - velocity_new["W" + str(l+1)]
    parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - velocity_new["b" + str(l+1)]
```

4.1.2 Stochastic, Mini-Batch, Full-Batch

The implementation of the three learning models is as such.

```
"""
Mini-Batch, Stochastic is the following code but with batch sizes of one, this means it still
↪  get's randomly shuffled

batchesX = batchify(newX, batch_size, 1)
batchesY = batchify(newY, batch_size, 0)
acc_total = 0
loss_total = 0
```

```
for b in range(len(batchesX)):
    bx = batchesX[b]
    by = batchesY[b]

    AL, caches = forward_pass(bx, parameters)
    loss_single_batch = loss_fun(AL, by)
    loss_total += loss_single_batch
    acc_single_batch = model_accuracy(AL, by)
    acc_total += acc_single_batch
    grads = backprop_loop(AL, by, caches)
    parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
↪   momentum_coefficient, velocity)
    velocity = velocity_new

loss_fulldataset = loss_total/(len(batchesX))
accuracy_fulldataset = acc_total/(len(batchesX))
"""


"""
Full-Batch

AL, caches = forward_pass(newX, parameters)
# prev = parameters
loss_fulldataset = loss_fun(AL, newY)
accuracy_fulldataset = model_accuracy(AL, newY)
grads = backprop_loop(AL, newY, caches)
parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
↪   momentum_coefficient, velocity)
velocity = velocity_new
"""
```

4.2 Finding “good” hyper-parameters

The method for finding good hyper-parameters can be seen below.

4.3 Hyper-parameter tuning

ML models, whether supervised or not, have the objective to minimize an expected loss $L(\hat{y}, y)$ over our training data. Our learning model is to infer a function $f_s(x_i) \approx y_i$ for any given unknown future data sets. Machine learning models often have other parameters known as *hyper-parameters*, often denoted by λ, and the learning algorithm is one chosen from a set of these different λ. Our optimal model can be described as (adapted from Bergstra and Bengio [2]):

$$\lambda^* \approx argmin_{\lambda \in H} E[L(\hat{y}, A_\lambda(X^{train}))] \tag{13}$$

The goal is to choose the best combination of parameters such that λ^* yields the best model. Multiple hyper-parameter search strategies exists, with “grid” and “manual” search being the most commonly used.

4.3.1 Grid search

For this model, grid search was used to explore the optimal hyper-parameters. A grid search experiment is created by forming all possible hyper-parameter values into a set of combinations. For our model our values are: learning rate, batch size and the momentum coefficient. The number of trials can then be formalised as a search of $S = \Pi_{k=1}^{K} L^k$ elements. This product however as noted by Bellman [1], suffers from the *curse of dimensionality*, as the number of hyper-parameters grows the product between them grows exponentially. The grid search algorithm testing the different hyper-parameters can be seen below:

```
learning_rates = [0.1,0.01,0.001]
batch_sizes = [32,64,128,256,len(trainxs_flat)]
momentums = [0.95, 0.9, 0.0]

for i in range(len(learning_rates)):
    for j in range(len(momentums)):
        for b in range(len(batch_sizes)):
            rate = learning_rates[i]
            momentum = momentums[j]
            batch = batch_sizes[b]
            parameters, best_model = L_layer_model(trainxs_flat,
                                                   trainys,
                                                   layers_dims,
                                                   learning_rate = rate,
                                                   max_epochs=10000,
                                                   momentum_coefficient=momentum,
                                                   batch_size=batch)
```

Our hypothesis for choosing the best model hyper-parameters can be described as the following:

- For every hyper-parameter combination, train the model to convergence. Once completed, note the accuracy and loss on both the validation and training data set and the epoch required to meet convergence.
- Repeat the above for the three different gradient descent learning models: mini-batch, full-batch and stochastic.
- Compare the validation accuracy for all models and parameters.
- One may conclude that the best choice of hyper-parameters are those that have the highest validation accuracy scores, but also converged in the smallest amount of time.

The best hyper-parameters values for the different types of gradient descent can be seen in Table 4. The resulting loss and accuracy plots for all these models can be seen in Figure 6.

Type	Learning Rate	Batch Size	Momentum Val	Accuracy Val	Accuracy Train
Stochastic w/Momentum	0.01	N/A	0.9	0.986	0.99
Stochastic no/Momentum	0.1	N/A	N/A	0.985	0.99
Mini-Batch w/Momentum	0.1	32	0.9	0.985	0.98
Mini-Batch no/Momentum	0.1	32	N/A	0.975	0.97
Full-Batch w/Momentum	0.1	N/A	0.9	0.972	0.96
Full-Batch no/Momentum	0.1	N/A	N/A	0.959	0.95

Table 4: 2-layer MLP Best Hyper-Parameters Values

From the table, it may be seen that the best training and validation accuracy results are achieved by training with Stochastic Gradient Descent, with Momentum, with a learning rate of 0.01 and a momentum value of 0.9. Higher learning rates have tended to perform better for most models tested. Momentum seems to also have improved performance. Validation accuracy seemed to improve from full batch gradient descent, to mini-batch, and again to stochastic.

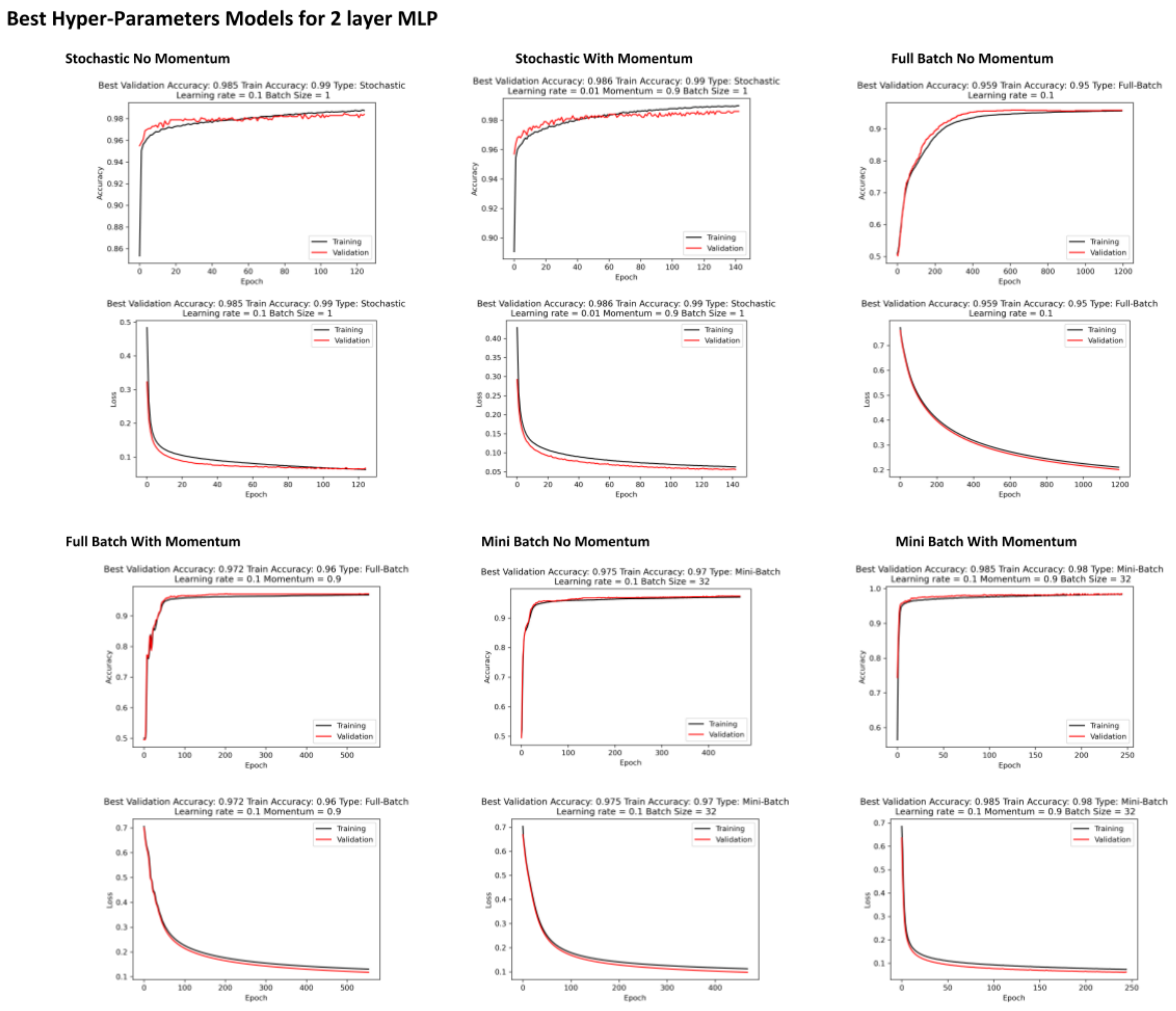


Figure 6: 2-layer MLP Best Hyper-Parameters Loss/Accuracy

4.4 2-layer MLP: Loss plot for "good" hyper-parameters

The plot of loss for the training and validation datasets, for our best performing 2-layer MLP after hyperparameter selection, are shown in Figure 7.

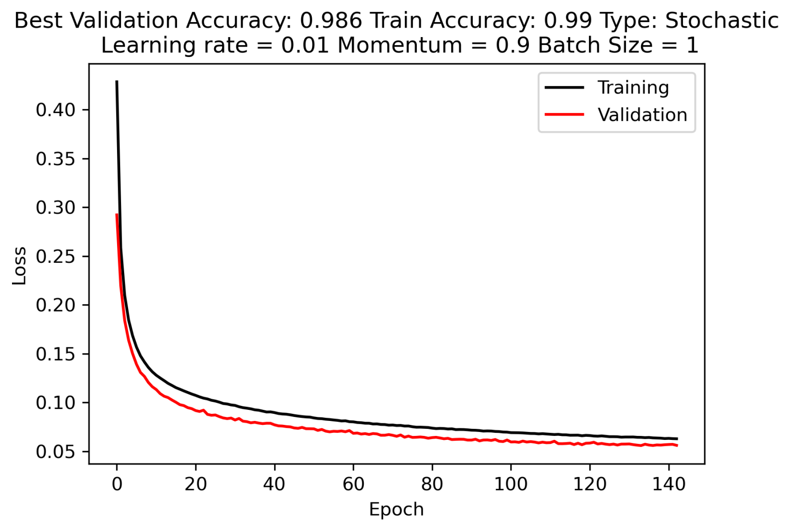


Figure 7: Best 2-layer MLP loss plot for training and validation datasets

4.5 2-layer MLP: Accuracy plot for "good" hyper-parameters

The plot of accuracy for the training and validation datasets, for our best performing model, are shown in Figure 8.

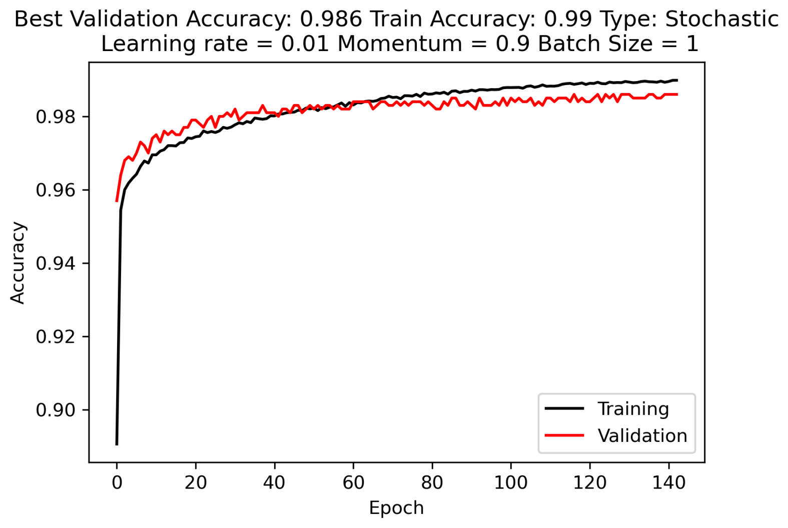


Figure 8: Best 2-layer MLP accuracy plot for training and validation datasets

4.6 2-layer MLP: Train and validation accuracy for best epoch

As shown in Table 5 and at the top of Table 4, our best 2-layer MLP achieved a highest accuracy of 0.986 on the validation dataset after 116 epochs of training. This epoch was associated with an accuracy of 0.990 on the training dataset.

Set	Accuracy
Training	0.990
Validation	0.986

Table 5: The training and validation accuracy at epoch 116

5 Question 5: Best Model vs Baseline

5.1 Vanilla perceptron model implementation

We have implemented a "vanilla" perceptron model to serve as a baseline model, summarised in Algorithm 2. The code implementation is below.

Algorithm 2: Vanilla Perceptron

```
Data: $\mathcal{D} := \{(\boldsymbol{x_1}, y_1), ..., (\boldsymbol{x_n}, y_n)\}$
$\boldsymbol{w} := \boldsymbol{0^d}$
$b := 0$
while not converged do
    shuffle dataset
    for $i := 1, .., n$ do
        $\hat{y} := \begin{cases} 1 & \text{if } \boldsymbol{x_i} \cdot \boldsymbol{w} + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$
        if $\hat{y} = y$ then
            continue
        else if $y = 1$ then
            $\boldsymbol{w} := \boldsymbol{w} + \boldsymbol{x_i}$
            $b := b + 1$
        else if $y = 0$ then
            $\boldsymbol{w} := \boldsymbol{w} - \boldsymbol{x_i}$
            $b := b - 1$
        end if
    end for
end while
```

```python
# Implementation of vanilla perceptron

MAX_NUMBER_OF_EPOCHS = 100
w = np.zeros(dims)
b = 0.0

train_accuracy_per_epoch = []
dev_accuracy_per_epoch = []

best_dev_accuracy = 0
best_dev_bias = 0
best_dev_weights = np.zeros(dims)

for j in range(MAX_NUMBER_OF_EPOCHS):
    train_errors = 0
    dev_errors = 0

    shuffled_indices = [*range(0,train_n,1)]
    np.random.shuffle(shuffled_indices)

    # run a training epoch
    for i in shuffled_indices:
        train_x = trainxs_flat[:,i]
        train_pred = 1 if (np.dot(w,train_x) + b)>=0 else 0
        if train_pred == 0 and trainys[i] ==1:
            w += train_x
            b += 1
```

```
                train_errors += 1
            if train_pred == 1 and trainys[i] ==0:
                w -= train_x
                b -= 1
                train_errors += 1
        train_accuracy_per_epoch.append((train_n-train_errors)/train_n)

        #evaluate the updated weights and bias on the validation set
        for i in range(dev_n):
            dev_x = devxs_flat[:,i]
            dev_pred = 1 if (np.dot(w,dev_x) + b)>=0 else 0
            if dev_pred != devys[i]:
                dev_errors += 1
        dev_accuracy = (dev_n-dev_errors)/dev_n
        dev_accuracy_per_epoch.append(dev_accuracy)

        #store accuracy, weights, and bias from epoch with best validation accuracy
        if dev_accuracy > best_dev_accuracy:
            best_dev_accuracy = dev_accuracy
            best_dev_weights = w
            best_dev_bias = b
            best_weights_epoch = j

        # early stopping criteria
        if len(dev_accuracy_per_epoch) >= 30:
            if (np.mean(dev_accuracy_per_epoch[-15:]) - np.mean(dev_accuracy_per_epoch[-30:-15]))
             ↪  < 0:
                print("break")
                break

# accuracy on the training for the epoch on which we obtain the highest accuracy on the
↪  validation set

best_train_accuracy = train_accuracy_per_epoch[best_weights_epoch]

# Training and validation accuracy plot for vanilla perceptron

plt.plot(train_accuracy_per_epoch, color='k', linestyle='-')
plt.plot(dev_accuracy_per_epoch, color='r', linestyle='-')
plt.title(f'Model accuracy',  color='k')
plt.ylabel('Accuracy',  color='k')
plt.xlabel('Epoch',  color='k')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.tick_params(colors='k')
plt.xlim(0)
plt.savefig("accuracy_plot.png", dpi=300)
plt.show()
```

5.2 Baseline model training

We trained the vanilla perceptron on the training dataset, till convergence of the accuracy on the validation dataset. We then selected the final model weights to be those corresponding to the epoch with best performance on the validation dataset. See the code to train this model in the answer to question 5.1.

We implemented early stopping, where training stopped if the mean accuracy from the most recent 15 epochs, was less than the mean accuracy for the preceding 15 epochs. This averaging in the early stopping metric was necessary to smooth, and thereby compensate for, the choppiness of the validation accuracy. The training had to precede for a minimum of 30 epochs before early stopping could be triggered.

In our case here, we achieved a best validation accuracy of 0.976 on the 13th epoch, with early stopping triggered at 32 epochs. The training accuracy during this epoch was 0.958. We include here our plot of training and validation accuracy (see Figure 9).

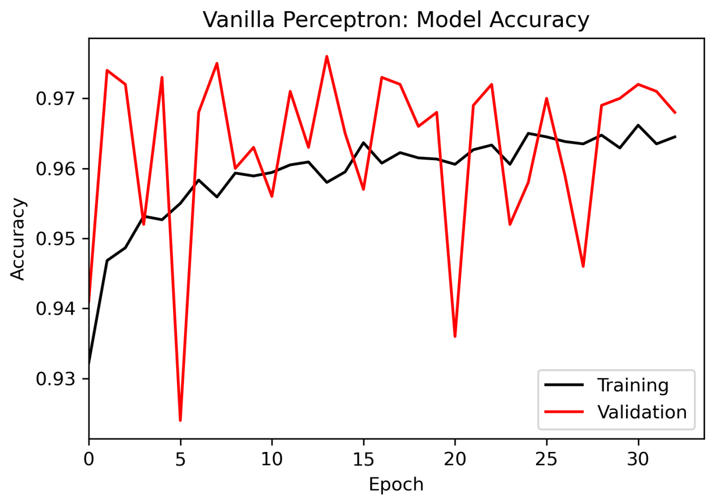


Figure 9: Training and validation accuracy by epoch for the vanilla perceptron algorithm

5.3 Outperforming the baseline model

The full code for used to train and identify our best 2 layer model is in Appendix B as developed for question 4. The 3 layer MLP version of this code used in answering this question is found in Appendix C. Our best single network consisted of a 2 layer MLP with 3 nodes in the hidden layer (network developed in question 4). The network was trained using stochastic gradient descent, with a learning rate of 0.01, and momentum coefficient of 0.9. It achieved a best validation accuracy of 0.986 after 116 epochs of training, and an associated training accuracy of 0.990 (Model 1, Table 6). Training included an early stopping criterion for lack of improvement in the validation loss. The process used to attempt to find a best model is described in the answer to the next sub-question.

5.4 Approach to identifying best model

After an initial comparison of our results from models developed in previous questions, the 2 layer MLP performed best after hyperparameter selection in terms of validation accuracy (best 2 layer MLP: 0.986 validation accuracy). Therefore, the next step was to test whether a 3 layer MLP would result in further improvements in performance after hyperparameter selection. We trained a 3 layer MLP with mini-batch gradient descent, and undertook hyperparameter grid search over batch size, learning rate, and momentum. We kept the number of nodes in each hidden layer at 100 and 60, reflecting our experiences from initial scoping experiments we performed as part of our initial 3 layer model development for question 3.3. The full code for the 3 layer model implementation in this question is shown in Appendix C. The summary of results of the grid search for the 3 layer MLP are shown in Table 6, and the accuracy and loss plots for training and validation datasets are shown in Figure 10. Of note from the grid search results, is that better accuracy on the validation dataset seemed to correlate with use of a higher learning rate, a smaller batch size, and the inclusion of momentum. Despite this, our best 3 layer MLP still marginally underperformed in terms of validation accuracy (0.985 validation accuracy) compared to the best 2 layer MLP trained in question 4 (0.986 validation accuracy).

Here is the code corresponding to the grid search:

```
# Set dimensionality of each layer
layers_dims = [784, 100, 60, 1]

# Select hyperparameters using grid search
```

```
learning_rates = [0.1,0.01]
batch_sizes = [32,64,256]
momentums = [0.9, 0.0]

stored_best_models = {}

# Loop through hyperparameters (grid search)
for i in range(len(learning_rates)):
    for j in range(len(momentums)):
        for b in range(len(batch_sizes)):
            rate = learning_rates[i]
            momentum = momentums[j]
            batch = batch_sizes[b]
            print("\n")
            print("\n")
            print(f"LEARNING RATE: {rate}")
            print(f"MOMENTUM: {momentum}")
            print(f"BATCH SIZE: {batch}")

            parameters, best_model = L_layer_model(trainxs_flat,
                                                   trainys,
                                                   layers_dims,
                                                   learning_rate = rate,
                                                   max_epochs=5000,
                                                   momentum_coefficient=momentum,
                                                   batch_size=batch)

            stored_best_models[str(rate) + "-" + str(momentum) + "-" + str(batch)] =
            ↪  best_model

            # Print out results for epoch with best validation accuracy
            best_dev_accuracy, best_train_accuracy, best_epoch, best_dev_parameters =
            ↪  best_model
            print(f"best_dev_accuracy: {best_dev_accuracy}")
            print(f"train_accuracy: {best_train_accuracy}")
            print(f"best_epoch: {best_epoch}")
```

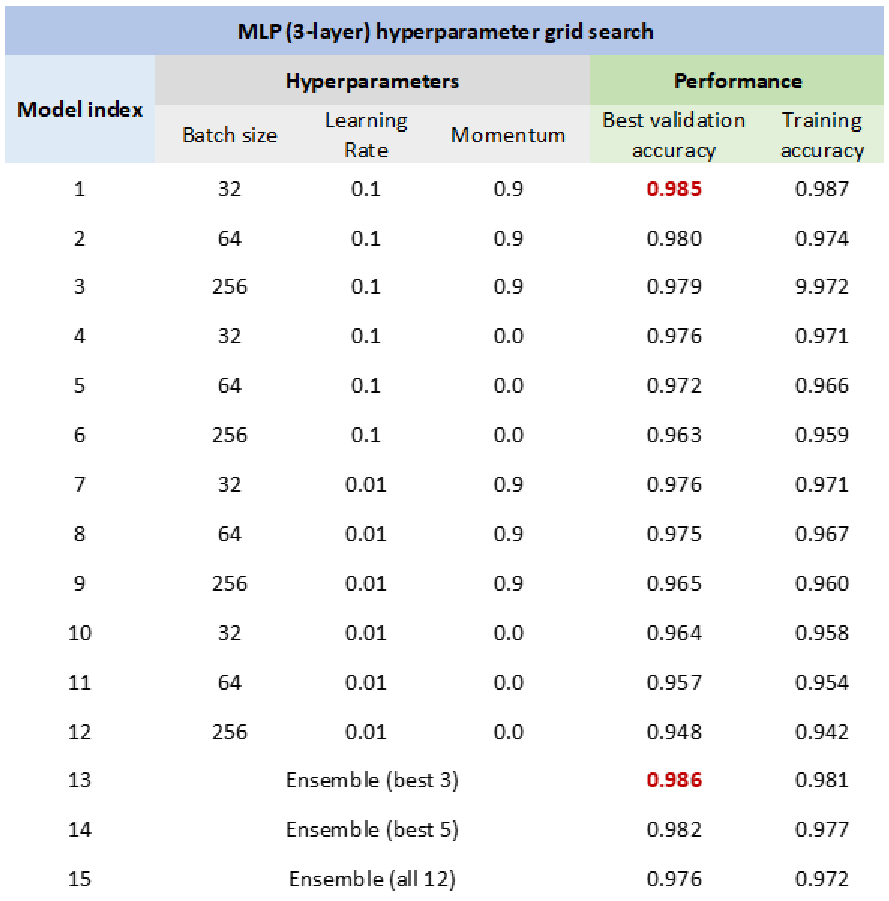
We also tested whether ensembled selections of the 3 layer MLP networks would improve performance. The code for the ensembling is shown below. The final layer activations for each network were averaged (mean), and this averaged version of activations was used to make classification predictions. Each network in the ensemble received equal weighting. We tried three varieties of 3 layer MLP ensembles: the best 3 networks, best 5 networks, and all 12 of the networks. The results are reflected in Table 6. Only the ensemble of the best 3 networks outperformed (Model 13, Table 6, best validation accuracy: 0.986) our best single 3 layer MLP network (Model 1, Table 6, best validation accuracy: 0.985), but very marginally so. We expect that main reason ensembling did not result in a significant improvement here was that all ensembled models had the same architecture, despite differences in other hyperparameters. Further work could involve trying a wider range of architectures ensembled together, for example architectures including convolutional layers. We expect ensembling a greater variety of architectures like this may result in a more significantly improved performance.

Given the ensemble of three 3-layer MLPs attained a validation accuracy just the same as our much simpler best 2-layer MLP, we concluded that the 2-layer MLP represents the better model.

Here is the code corresponding to the ensembling:

```
#Ensembling the best models (those with highest validation accuracy)

#Accuracy on validation dataset
summed_activations = np.zeros((1,dev_n))
model_count = 0
```

```
for best_model in stored_best_models.values():
    best_dev_accuracy , _ , _ , parameters = best_model
    if best_dev_accuracy >= 0.95: #ensemble models with a validation accuracy in specified
     ↪  range
        model_count += 1
        activations, _ = forward_pass(devxs_flat, parameters)
        summed_activations += activations

val_ensembled_activations = summed_activations/model_count

val_accuracy = model_accuracy(val_ensembled_activations, devys)

print(f"Validation accuracy: {val_accuracy}")

#Accuracy on training dataset
summed_activations = np.zeros((1,train_n))
model_count = 0

for best_model in stored_best_models.values():
    best_dev_accuracy , _ , _ , parameters = best_model
    if best_dev_accuracy >= 0.95: #ensemble models with a validation accuracy in specified
     ↪  range
        model_count += 1
        activations, _ = forward_pass(trainxs_flat, parameters)
        summed_activations += activations

train_ensembled_activations = summed_activations/model_count

train_accuracy = model_accuracy(train_ensembled_activations, trainys)

print(f"Training accuracy: {train_accuracy}")
```

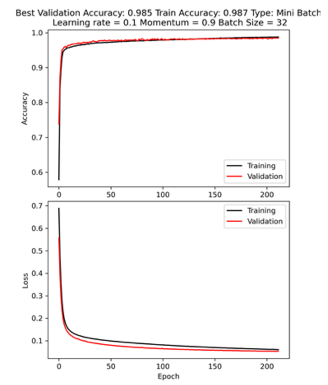


MLP (3-layer) hyperparameter grid search					
Model index	**Hyperparameters**			**Performance**	
	Batch size	Learning Rate	Momentum	Best validation accuracy	Training accuracy
1	32	0.1	0.9	**0.985**	0.987
2	64	0.1	0.9	0.980	0.974
3	256	0.1	0.9	0.979	9.972
4	32	0.1	0.0	0.976	0.971
5	64	0.1	0.0	0.972	0.966
6	256	0.1	0.0	0.963	0.959
7	32	0.01	0.9	0.976	0.971
8	64	0.01	0.9	0.975	0.967
9	256	0.01	0.9	0.965	0.960
10	32	0.01	0.0	0.964	0.958
11	64	0.01	0.0	0.957	0.954
12	256	0.01	0.0	0.948	0.942
13	Ensemble (best 3)			**0.986**	0.981
14	Ensemble (best 5)			0.982	0.977
15	Ensemble (all 12)			0.976	0.972

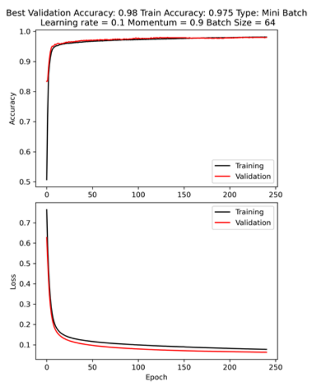
Table 6: Summary of results of hyperparameter grid search for the 3 layer MLP trained with mini-batch gradient descent

Accuracy and loss plots for MLP (3 layer) hyperparameter grid search

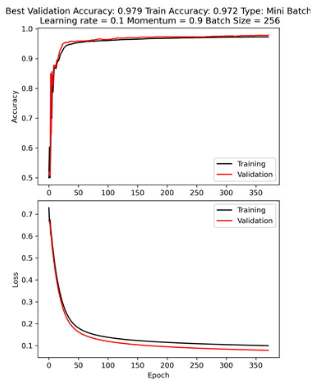
Model 1



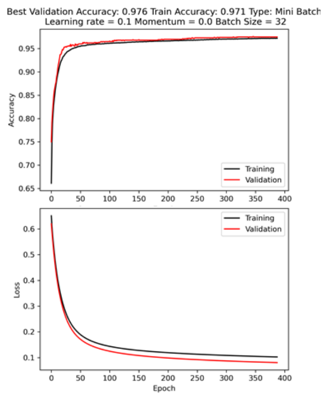
Model 2



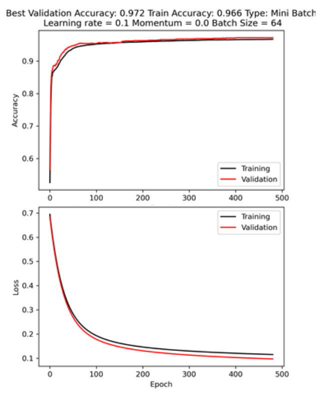
Model 3



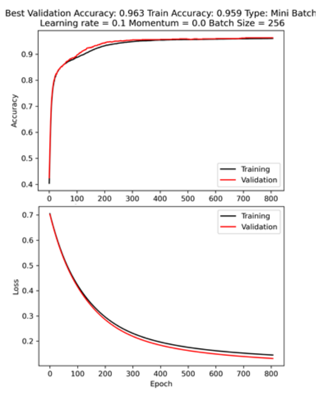
Model 4



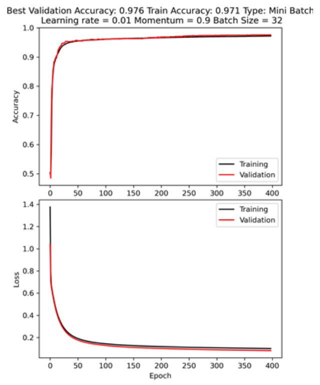
Model 5



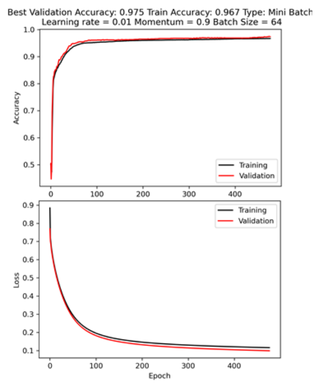
Model 6



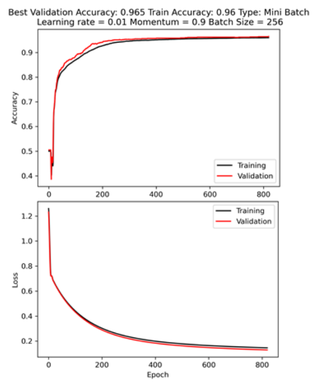
Model 7



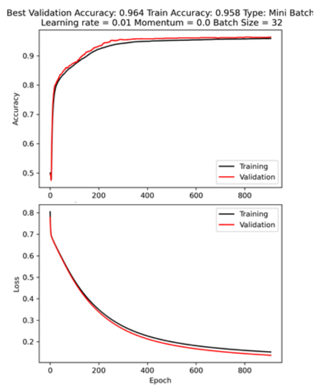
Model 8



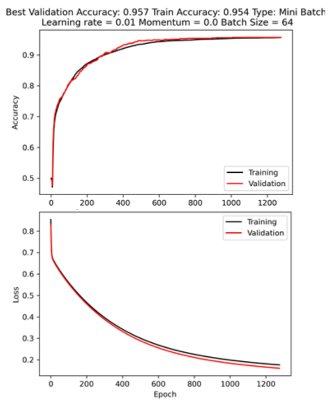
Model 9



Model 10



Model 11



Model 12

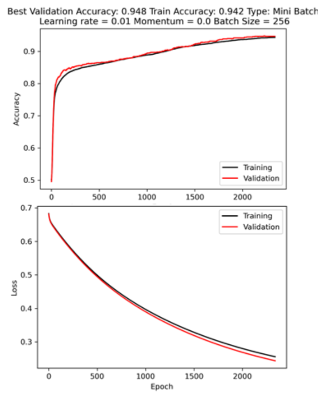


Figure 10: 3 layer MLP grid search plots

5.5 Best model: Loss plot

The plot of loss for the training and validation datasets, for our best performing model, are shown in Figure 11.

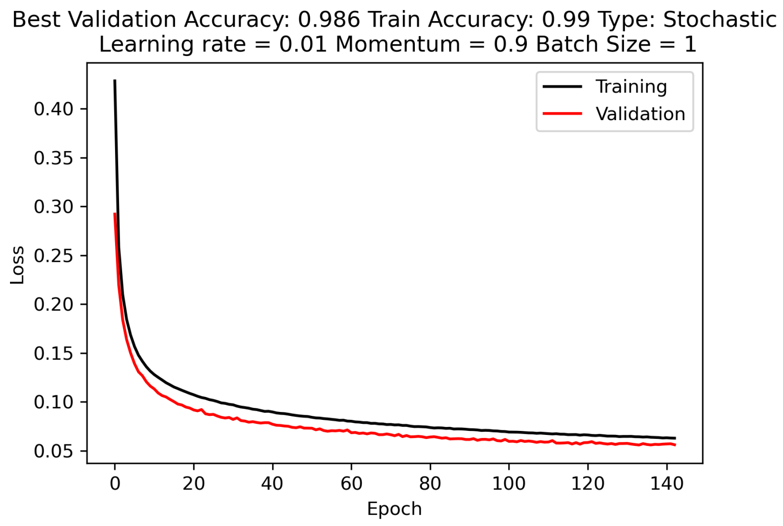


Figure 11: Best model loss plot for training and validation datasets

5.6 Best model: Accuracy plot

The plot of accuracy for the training and validation datasets, for our best performing model, are shown in Figure 12.

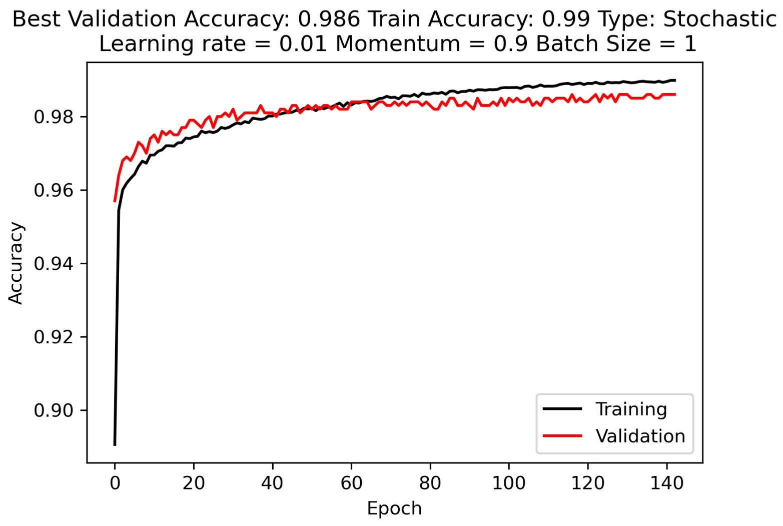


Figure 12: Best model accuracy plot for training and validation datasets

5.7 Best model: Train and validation accuracy for best epoch

As shown in Table 7, our best single model achieved a highest accuracy of 0.986 on the validation dataset after 116 epochs of training. This epoch was associated with an accuracy of 0.990 on the training dataset.

Set	Accuracy
Training	0.990
Validation	0.986

Table 7: The training and validation accuracy at epoch 116

References

[1] Richard E Bellman. *Adaptive control processes: a guided tour*. Vol. 2045. Princeton university press, 2015.

[2] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *The Journal of Machine Learning Research* 13.1 (2012), pp. 281–305.

Appendix A 3-layer MLP Network - version used in Q3

A.1 MLP

```python
# Input a list of dimensions of all layers from input to output to randomly generate our
↪  parameters
def parameters_initialization(layer_dims):

    L = len(layer_dims)             # Number of layers in the network
    parameters = {}                 # Create a dictionary to store all parameter values

    for l in range(1, L): # Layer 0 is the input layer, so no weights to initialize
        # Initialize using Glorot initialization
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l - 1]) *
        ↪  np.sqrt(6/(layer_dims[l - 1] + layer_dims[l]))
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters

# The Forward Pass that generates an output. In our case it is
# a repetitive application of sigmoid functions on linear functions
def forward_pass(X, parameters):

    store_values = []

    L = len(parameters) // 2 # Number of layers in the neural network. Dividing by 2 as
    ↪  parameter variable includes W and b per layer. See parameters_initialization

    a = X
    # Compute linear and sigmoid till last layer.
    for l in range(1, L):
        a_prev = a

        W=parameters['W' + str(l)] # Calling corresponding W from dictionary 'parameters'
        b=parameters['b' + str(l)] # Calling corresponding b from dictionary 'parameters'

        # Linear calculation
        Z = (W @ a_prev) + b
        # Store parameters that generate Z, for backprop
        linear_cache = (a_prev, W, b)

        # Sigmoid Activation Function
        a = 1/(1+np.exp(-Z))
        #Store Z that generates A, for backprop
        activation_cache = Z

        # Store both cache together and add to 'caches'
        both = (linear_cache, activation_cache)
        store_values.append(both)
    # Same as code in loop, just for the last layer which will be used to compute loss
    a_prev = a

    W=parameters['W' + str(L)]
    b=parameters['b' + str(L)]

    Z = (W @ a_prev) + b
    linear_cache = (a_prev, W, b)


    output = 1/(1 + np.exp(-Z))
    activation_cache = Z

    both = (linear_cache, activation_cache)
    store_values.append(both)

    return output, store_values

#Negative Log-likelihood Loss Function
def loss_fun(output, Y):

    m = Y.shape[0] #Tested the shape of Y, found that the data is in rows

    # Compute loss from output (AL) and expected output (Y).
    loss = -(1/m) * np.sum((Y * np.log(output)) + (1-Y) * np.log(1-output))

    # Make sure loss is regular array, not array in array
    loss = np.squeeze(loss)

    return loss

#Prediction, based on last layer output
def predict(out):
    if out >= 0.5:
        return 1
    else:
        return 0

#Accuracy metric
def model_accuracy(dataX, Y):
    m = Y.shape[0]

    correct = 0
    for i in range(dataX.shape[1]):
        item = np.array(dataX[:, i])
        itemY = Y[i]
        if itemY == predict(item):
            correct += 1

    accuracy = (correct / m)

    accuracy = np.squeeze(accuracy)
    return accuracy

#General form of backpropagation
def backprop(dA, both):
    linear_cache, activation_cache = both

    #First we calculate dL/dZ
    #Call our Z
    Z = activation_cache
    #Apply Sigmoid on them
    s = 1/(1 + np.exp(-Z))
    #dL/dZ in general form
    dZ = dA * s * (1-s)

    #Using this, we can calculate the other differentials
    #Call our parameters
    a_prev, W, b = linear_cache
    m = a_prev.shape[1]
    #dL/dW, dL/db and dL/dA_prev in general form
    dW = (1/m) * (dZ @ a_prev.T)
    db = (1/m) * np.sum(dZ,axis=1,keepdims=True)
    da_prev = W.T @ dZ

    return da_prev, dW, db

#Backpropagation for L layers. Takes model output AL and expected output
#As well as cache of parameter values
def backprop_loop(output, y, both):

    gradients = {} #Dictionary to store our gradients

    y = y.reshape(output.shape) # Y is the same shape as AL
    m = output.shape[1]

    L = len(both) # the number of layers

    # Initialising backprop.
    dOut = -(np.divide(y, output) - np.divide(1 - y, 1 - output))

    # Gradients from the last layer.
    current_cache = both[L-1]
    gradients["da" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] =
    ↪  backprop(dOut, current_cache)

    #Gradients from rest of layers
    for l in reversed(range(L-1)):
        current_cache = both[l]
        da_prev_, dW_, db_ = backprop(gradients['da' + str(l+2)], current_cache)
        gradients["da" + str(l + 1)] = da_prev_
        gradients["dW" + str(l + 1)] = dW_
        gradients["db" + str(l + 1)] = db_

    return gradients


#Parameter Updater
def update_parameters(parameters, grads, learning_rate):
    L = len(parameters) // 2 # Number of layers in the neural network. We are dividing by 2
    ↪  because parameters contains W and b

    for l in range(L):
        parameters["W" + str(l+1)] =parameters["W" + str(l+1)] - learning_rate * grads["dW" +
        ↪  str(l+1)]
        parameters["b" + str(l+1)] =parameters["b" + str(l+1)] - learning_rate * grads["db" +
        ↪  str(l+1)]

    return parameters

def MLP(X, y, layers_dims, learning_rate, epochs):

    #Keep track of loss and accuracy of each epoch
    loss = []
    accs = []

    dev_loss = []
    dev_accuracies = []

    # Parameters initialisation.
    parameters = parameters_initialization(layers_dims)

    #Run epochs
    for i in range(0, epochs):

        #Forward pass
        output, both = forward_pass(X, parameters)

        #Loss
        los = loss_fun(output, y)

        #Accuracy
        acc = model_accuracy(output, y)

        # Backprop
        grads = backprop_loop(output, y, both)

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        #Record loss and Acuracy
        loss.append(los)
        accs.append(acc)

        # Print the loss every 100 training example
        if i % 100 == 0:
            print ("Loss after Epoch %i: %f" % (i, los))
            print ("Accuracy after Epoch %i: %f" % (i, acc))


        # Calculate Loss and Accuracy on Validation set using current epoch's parameters
        out = forward_pass(devxs_flat, parameters)

        dev_los = loss_fun(out[0], devys)
        dev_accuracy = model_accuracy(out[0], devys)
        # Record Validation Loss and Acuracy
        dev_loss.append(dev_los)
        dev_accuracies.append(dev_accuracy)

        if len(dev_loss) >= 300:
            if (np.mean(dev_loss[-20:]) - np.mean(dev_loss[-40:-20])) > -.001:
                print("Converged or reached max epochs")
                break


    # plot the loss
    plt.plot(np.squeeze(loss), color='k', linestyle='-')
    plt.plot(np.squeeze(dev_loss), color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.title(f"Loss Plot for Training and Validation Set")
    plt.show()

    # plot the accuracy
    plt.plot(np.squeeze(accs), color='k')
    plt.plot(np.squeeze(dev_accuracies), color='r')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.title("Accuracy Plot for Training and Validation Set")
    plt.show()

    return parameters
```

A.2 Gradient checking

```
#Set layer dimensions, here using few parameters as just trying to see if gradient descent is
↪  working
layers_dims = [784, 3, 3, 1]

#Function to calculate finite methods
def fin_method(parameters, gradients, X, Y, epsilon = 10**-10):

    parameters = parameters

    #Dictionary to hold Finite Methods gradients
    gradients = {}

    #retrieve shape of each parameter key
    for key in parameters:
        shape = parameters[key].shape
        shape_i = shape[0]
        shape_j = shape[1]

        #Doing finite method by changing each parameter
        grads = np.empty(shape)
        for i in range(0, shape_i):
            for j in range(0, shape_j):
                parameter_ij = parameters[key][i,j]
                parameters[key][i,j] += epsilon / 2.0
                output, both = forward_pass(trainxs_flat, parameters)
                lhs = loss_fun(output, trainys)
                #Restore parameter values
                parameters[key][i,j] = parameter_ij

                parameters[key][i,j] -= epsilon / 2.0
                output, both = forward_pass(trainxs_flat, parameters)
                rhs = loss_fun(output, trainys)
                #Restore parameter values
                parameters[key][i,j] = parameter_ij

                #Finite difference
                fd_ij = (lhs - rhs) / epsilon

                grads[i][j] = fd_ij

        #Change name of key to match backprops names
        key = "d" + key
        #Add to gradients
        gradients[key] = grads
        print(key)

    return gradients

#Calculate difference between backprop and finite methods
def abs_dif(backprop_grads, finite_methods):

    difference = []

    for key in finite_methods:
        dif = np.absolute(finite_methods[key] - backprop_grads[key]).flatten()

        difference.append(dif)

    difference = np.concatenate(difference).ravel()

    for i in difference:
        if i > 10**-5:
            print('considerable difference!')

    average = np.mean(difference)
    return difference


#Initialize parameters, pass them forward, calculate loss and obtain gradients via
↪  backpropagation
parameters = parameters_initialization(layers_dims)
output, both = forward_pass(trainxs_flat, parameters)
loss = loss_fun(output, trainys)
backprop_grads = backprop_loop(output, trainys, both)


#Get finite differences and compare them to gradients
```

```
finite_methods = fin_method(parameters, backprop_grads, trainxs_flat, trainys, epsilon =
↪  10**-10)
difs = abs_dif(backprop_grads, finite_methods)
print('The largest difference between the Analytical Gradients and the Finite Differences is:
↪  ', max(difs))
```

```
The largest difference between the Analytical Gradients and the Finite Differences is:
↪  2.9182475596535237e-06
```

Appendix B 2-layer MLP Network - version used in Q4 & Q5

```
# Implementation and hyperparameter selection for 2-layer MLP
# Hyperparameters grid searched through: gradient descent method (SGD, mini-batch, full
↪  batch), batch size, learning rate, momentum

def logistic(x):
  """
  The logistic function
  """
  return 1.0 / (1.0 + np.exp(-x))

def logistic_prime(x):
  """
  Differential of logistic function
  """
  return np.multiply(logistic(x), (1 - logistic(x)))

def predict(z):
  """
  Convert final layer activations into predictions
  """
  if z >= 0.5:
    return 1
  else:
    return 0

def build_model(layer_dims):
    """
    Specify the structure of the network and initialise the parameters
    """
    parameters = {}                 #Create a dictionary to store all parameter values
    L = len(layer_dims)             # number of layers in the network

    for l in range(1, L): #Layer 0 is the input layer, so no weights to initialise
        #Intilisation using Glorot Initilisation
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l - 1]) *
        ↪  np.sqrt(6/(layer_dims[l - 1] + layer_dims[l]))
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters


def forward_pass(X, parameters):
    """
    Pass data through network as a forward pass
    Repetitive application of sigmoid on linear function
    """
    caches = []

    A = X
    L = len(parameters) // 2    # number of layers in the neural network. Dividing by 2 as
    ↪  parameter variable includes W and b per layer. See initialize_parameters_deep

    # Compute linear and sigmoid till last layer.
    for l in range(1, L):
        A_prev = A

        W=parameters['W' + str(l)] #Calling corresponding W from dictionary 'parameters'
        b=parameters['b' + str(l)] #Calling corresponding b from dictionary 'parameters'

        #Linear calculation
        Z = np.dot(W,A_prev)+b
        #Store parameters that generate Z, for backprop
        linear_cache = (A_prev, W, b)

        #Sigmoid Activation Function
        A = 1/(1+np.exp(-Z))
        #Store Z that generates A, for backprop
        activation_cache = Z

        #Store both cache together and add to 'caches'
        cache = (linear_cache, activation_cache)
        caches.append(cache)

    # Same as code in loop, just for the last layer which will be used to compute loss
    A_prev = A

    W=parameters['W' + str(L)]
    b=parameters['b' + str(L)]

    Z = np.dot(W,A_prev)+b
    linear_cache = (A_prev, W, b)


    AL = 1/(1+np.exp(-Z))
    activation_cache = Z

    cache = (linear_cache, activation_cache)
    caches.append(cache)

    return AL, caches


def loss_fun(AL, Y):
    """
    Negative log-likelihood loss function
    """
    m = Y.shape[0] #Tested the shape of Y, found that the data is in rows

    # Compute loss from output (AL) and expected output (Y).
    loss = -(1/m) * np.sum((Y * np.log(AL)) + (1-Y) * np.log(1-AL))

    # Make sure loss is regular array, not array in array
    loss = np.squeeze(loss)

    return loss

def model_accuracy(dataX, Y):
    """
    Evaluate how accurate the model is on a given dataset
    """
    m = Y.shape[0]

    correct = 0
    for i in range(dataX.shape[1]):
        item = np.array(dataX[:, i])
        itemY = Y[i]
        if itemY == predict(item):
            correct += 1

    accuracy = (correct / m)

    accuracy = np.squeeze(accuracy)
    return accuracy

def backprop(dA, cache):
    """
    General function for backprop gradients from one activation to a previous activation
    """

    linear_cache, activation_cache = cache

    #First we calculate dL/dZ
    #Call our Z
    Z = activation_cache
    #Apply Sigmoid on them
    s = 1/(1+np.exp(-Z))
    #dL/dZ in general form
    dZ = dA * s * (1-s)

    #Using this, we can calculate the other differentials
    #Call our parameters
    A_prev, W, b = linear_cache
    m = A_prev.shape[1]
    #dL/dW, dL/db and dL/dA_prev in general form
    dW = (1/m)*np.dot(dZ,np.transpose(A_prev))
    db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(np.transpose(W),dZ)

    return dA_prev, dW, db


def backprop_loop(AL, Y, caches):
    """
    Calculate gradients for backpropagation for all L layers.
    Takes model output AL and expected output as well as cache of parameter values.
    """

    grads = {} #Dictionary to store our gradients
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # Y is the same shape as AL

    # Initialising backprop.
    dAL = AL - Y
    # Gradients from the last layer.
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = backprop(dAL,
    ↪  current_cache)

    #Gradients from rest of layers
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = backprop(grads['dA' + str(l+2)], current_cache)
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

#Parameter Updater
def update_parameters(parameters, grads, learning_rate, momentum_coefficient, velocity_old):
    """
    Update all the parameters using the gradients calculated as part of backpropagation
    """

    L = len(parameters) // 2 # number of layers in the neural network. We are dividing by 2
    ↪  because parameter contains W and b
    weight_increment = {}
    velocity_new = {}
    for l in range(L):
        weight_increment["W"+str(l+1)] = learning_rate * grads["dW" + str(l+1)]
        weight_increment["b"+str(l+1)] = learning_rate * grads["db" + str(l+1)]
        velocity_new["W" + str(l+1)] = momentum_coefficient * velocity_old["W" + str(l+1)] +
        ↪  weight_increment["W" + str(l+1)]
        velocity_new["b" + str(l+1)] = momentum_coefficient * velocity_old["b" + str(l+1)] +
        ↪  weight_increment["b" + str(l+1)]

        # With momentum
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - velocity_new["W" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - velocity_new["b" + str(l+1)]
        # Without momentum
        # parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - weight_increment["W" +
        ↪  str(l+1)]
        # parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - weight_increment["b" +
        ↪  str(l+1)]

    return parameters, velocity_new

def batchify(array, n, ax):
    """
    Split the dataset into batches of a specified size ready for mini-batch gradient descent
    """

    amount = dims // n
    return np.array_split(array, amount, axis=ax)

def shuffle(X, Y):
  """
  Shuffle the order of the training dataset images in the numpy array containing them.
  This can improve convergence.
  """

    shuffled_indices = [*range(0, train_n, 1)]
    np.random.shuffle(shuffled_indices)
    newX = np.zeros((dims, train_n))
    newY = np.zeros((train_n))

    for j, i in enumerate(shuffled_indices):
        newX[:, j] = trainxs_flat[:, i]
        newY[j] = trainys[i]

    return newX, newY

def L_layer_model(X, Y, layers_dims, learning_rate, momentum_coefficient, max_epochs,
↪   batch_size):
    # Specify the type of gradient descent being performed (will update plot titles
    ↪   automatically)
    train_type = "Mini Batch"

    # Keep track of loss and accuracy of each epoch
    loss = []
    accs = []
    converged = False
    i = 0

    dev_loss = []
    dev_accuracies = []

    best_dev_accuracy = 0

    parameters = build_model(layers_dims)

    velocity = {"W1": np.array([0.0]), "W2": np.array([0.0]), "b1": np.array([0.0]), "b2":
    ↪   np.array([0.0])}
    while (converged == False) and (i <= max_epochs):
        newX, newY = shuffle(X, Y)

#       Mini-Batch if batch_size set >1 or SGD if batch size set = 1

        batchesX = batchify(newX, batch_size, 1)
        batchesY = batchify(newY, batch_size, 0)
        acc_total = 0
        loss_total = 0
        for b in range(len(batchesX)):
            bx = batchesX[b]
            by = batchesY[b]

            AL, caches = forward_pass(bx, parameters)
            loss_single_batch = loss_fun(AL, by)
            loss_total += loss_single_batch
            acc_single_batch = model_accuracy(AL, by)
            acc_total += acc_single_batch
            grads = backprop_loop(AL, by, caches)
            parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
            ↪   momentum_coefficient, velocity)
            velocity = velocity_new
        loss_fulldataset = loss_total/(len(batchesX))
        accuracy_fulldataset = acc_total/(len(batchesX))

        """
        # Full-Batch - uncomment this if you wish to run full batch


        AL, caches = forward_pass(newX, parameters)
        # prev = parameters
        loss_fulldataset = loss_fun(AL, newY)
        accuracy_fulldataset = model_accuracy(AL, newY)
        grads = backprop_loop(AL, newY, caches)
        parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
    ↪   momentum_coefficient, velocity)
        velocity = velocity_new
        """

        #Add the epochs training loss and accuracy values to lists, ready for plotting
        loss.append(loss_fulldataset)
        accs.append(accuracy_fulldataset)

        #Print out the training loss and accuracy every 100 epochs
        if i % 100 == 0:
            print ("Loss after iteration %i: %f" % (i, loss_fulldataset))
            print ("Accuracy after iteration %i: %f" % (i, accuracy_fulldataset))

        #Calculate loss and accuracy on validation dataset using current epoch's parameters
        out = forward_pass(devxs_flat, parameters)
        dev_los = loss_fun(out[0], devys)
        dev_accuracy = model_accuracy(out[0], devys)
        if i == 0 or i % 1 == 0:
            dev_loss.append(dev_los)
            dev_accuracies.append(dev_accuracy)

        #Store the epochs results and parameters if it is the highest validation accuracy so
        ↪   far
        if dev_accuracy > best_dev_accuracy:
            best_dev_accuracy = dev_accuracy
            best_train_accuracy = accuracy_fulldataset
            best_epoch = i
            best_dev_parameters = parameters

        # Early stopping criteria
        if i >= 100:
            if (np.mean(dev_loss[-15:]) - np.mean(dev_loss[-30:-15])) > -.001:
                converged = True
                print("Model Converged")

        i += 1

    # format title of plots to reflect the hyperparameters and metrics
    title = "Best Validation Accuracy: " +  str(np.round(best_dev_accuracy, 3)) + " Train
    ↪   Accuracy: " + str(np.round(best_train_accuracy, 3)) +  " Type: " + train_type + "\n
    ↪   Learning rate = " + str(learning_rate)
    if momentum_coefficient != -1: # Omit momentum in title if not needed
```

```
        title += " Momentum = " + str(momentum_coefficient)
      if batch_size != -1: # Put -1 in batch_size to omit batch size from plot titles (e.g. if
    ↪   doing SGD)
        title += " Batch Size = " + str(batch_size)

      # plot the training and validation loss
      plt.plot(np.squeeze(loss), color='k')
      plt.plot(np.squeeze(dev_loss), color='r')
      plt.legend(['Training', 'Validation'], loc='upper right')
      plt.ylabel('Loss')
      plt.xlabel('Epoch')
      plt.title(title)
      plt.savefig(f"loss_plot_{str(np.round(best_dev_accuracy, 4))}_{best_epoch}.png", dpi=300)
      plt.show()

      # plot the training and validation accuracy
      plt.plot(np.squeeze(accs), color='k')
      plt.plot(np.squeeze(dev_accuracies), color='r')
      plt.legend(['Training', 'Validation'], loc='lower right')
      plt.ylabel('Accuracy')
      plt.xlabel('Epoch')
      plt.title(title)
      plt.savefig(f"accuracy_plot_{str(np.round(best_dev_accuracy, 4))}_{best_epoch}.png",
    ↪   dpi=300)
      plt.show()

      best_model = (best_dev_accuracy, best_train_accuracy, best_epoch, best_dev_parameters)

      return parameters, best_model

  # Set dimensionality (number of nodes) in each layer
  layers_dims = [784, 3, 1]

  # Specify hyperparameters search through using grid search
  learning_rates = [0.1,0.01,0.001]
  batch_sizes = [32,64,128,256,len(trainxs_flat)]
  momentums = [0.95, 0.9, 0.0]

  stored_best_models = {}

  # Loop through hyperparameters (grid search)
  for i in range(len(learning_rates)):
      for j in range(len(momentums)):
          for b in range(len(batch_sizes)):
              rate = learning_rates[i]
              momentum = momentums[j]
              batch = batch_sizes[b]
              print("\n")
              print("\n")
              print(f"LEARNING RATE: {rate}")
              print(f"MOMENTUM: {momentum}")
              print(f"BATCH SIZE: {batch}")


              parameters, best_model = L_layer_model(trainxs_flat,
                                                      trainys,
                                                    layers_dims,
                                                    learning_rate = rate,
                                                    max_epochs=10000,
                                                    momentum_coefficient=momentum,
                                                    batch_size=batch)

stored_best_models[str(rate) + "-" + str(momentum) + "-" + str(batch)] =
↪   best_model

best_dev_accuracy, best_train_accuracy, best_epoch, best_dev_parameters =
↪   best_model
print(f"best_dev_accuracy: {best_dev_accuracy}")
print(f"best_train_accuracy: {best_train_accuracy}")
print(f"best_epoch: {best_epoch}")
```

Appendix C 3-layer MLP Network - version used in Q5

```
# Implementation and hyperparameter selection for 3-layer MLP trained using mini batch
↪  gradient descent
# Hyperparameters grid searched through: batch size, learning rate, momentum

def logistic(x):
  """
  The logistic function
  """
  return 1.0 / (1.0 + np.exp(-x))

def logistic_prime(x):
  """
  Differential of logistic function
  """
  return np.multiply(logistic(x), (1 - logistic(x)))

def predict(z):
  """
  Convert final layer activations into predictions
  """
  if z >= 0.5:
    return 1
  else:
    return 0

def build_model(layer_dims):
    """
    Specify the structure of the network and initialise the parameters
    """
    parameters = {}                #Create a dictionary to store all parameter values
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L): #Layer 0 is the input layer, so no weights to initialise
        # Initialise using Glorot Initilisation
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l - 1]) *
        ↪  np.sqrt(6/(layer_dims[l - 1] + layer_dims[l]))
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

    return parameters


def forward_pass(X, parameters):
    """
    Pass data through network as a forward pass
    Repetitive application of sigmoid on linear function
    """
    caches = []

    A = X
    L = len(parameters) // 2    # Number of layers in the neural network. Dividing by 2 as
    ↪  parameter variable includes W and b per layer. See initialize_parameters_deep

    # Compute linear and sigmoid till last layer.
    for l in range(1, L):
        A_prev = A

        W=parameters['W' + str(l)] #Calling corresponding W from dictionary 'parameters'
        b=parameters['b' + str(l)] #Calling corresponding b from dictionary 'parameters'

        #Linear calculation
        Z = np.dot(W,A_prev)+b
        #Store parameters that generate Z, for backprop
        linear_cache = (A_prev, W, b)

        #Sigmoid Activation Function
        A = 1/(1+np.exp(-Z))
        #Store Z that generates A, for backprop
        activation_cache = Z

        #Store both cache together and add to 'caches'
        cache = (linear_cache, activation_cache)
        caches.append(cache)

    # Same as code in loop, just for the last layer which will be used to compute loss
    A_prev = A

    W=parameters['W' + str(L)]
    b=parameters['b' + str(L)]

    Z = np.dot(W,A_prev)+b
    linear_cache = (A_prev, W, b)


    AL = 1/(1+np.exp(-Z))
    activation_cache = Z

    cache = (linear_cache, activation_cache)
    caches.append(cache)

    return AL, caches


def loss_fun(AL, Y):
    """
    Negative log-likelihood loss function
    """
    m = Y.shape[0] #Tested the shape of Y, found that the data is in rows

    # Compute loss from output (AL) and expected output (Y).
    loss = -(1/m) * np.sum((Y * np.log(AL)) + (1-Y) * np.log(1-AL))

    # Make sure loss is regular array, not array in array
    loss = np.squeeze(loss)

    return loss

def model_accuracy(dataX, Y):
    """
    Evaluate how accurate the model is on a given dataset
    """
    m = Y.shape[0]

    correct = 0
    for i in range(dataX.shape[1]):
        item = np.array(dataX[:, i])
        itemY = Y[i]
        if itemY == predict(item):
            correct += 1

    accuracy = (correct / m)

    accuracy = np.squeeze(accuracy)
    return accuracy

def backprop(dA, cache):
    """
    General function for backprop gradients from one activation to a previous activation
    """

    linear_cache, activation_cache = cache

    #First we calculate dL/dZ
    #Call our Z
    Z = activation_cache
    #Apply Sigmoid on them
    s = 1/(1+np.exp(-Z))
    #dL/dZ in general form
    dZ = dA * s * (1-s)

    #Using this, we can calculate the other differentials
    #Call our parameters
    A_prev, W, b = linear_cache
    m = A_prev.shape[1]
    #dL/dW, dL/db and dL/dA_prev in general form
    dW = (1/m)*np.dot(dZ,np.transpose(A_prev))
    db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(np.transpose(W),dZ)

    return dA_prev, dW, db


def backprop_loop(AL, Y, caches):
    """
    Calculate gradients for backpropagation for all L layers.
    Takes model output AL and expected output as well as cache of parameter values.
    """

    grads = {} #Dictionary to store our gradients
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # Y is the same shape as AL

    # Initialising backprop.
    dAL = AL - Y
    # Gradients from the last layer.
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = backprop(dAL,
    ↪  current_cache)

    #Gradients from rest of layers
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = backprop(grads['dA' + str(l+2)], current_cache)
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

def update_parameters(parameters, grads, learning_rate, momentum_coefficient, velocity_old):
    """
    Update all the parameters using the gradients calculated as part of backpropagation
    """

    L = len(parameters) // 2 # Number of layers in the neural network. We are dividing by 2
    ↪  because parameter contains W and b
    weight_increment = {}
    velocity_new = {}

    for l in range(L):
        weight_increment["W"+str(l+1)] = learning_rate * grads["dW" + str(l+1)]
        weight_increment["b"+str(l+1)] = learning_rate * grads["db" + str(l+1)]

        velocity_new["W" + str(l+1)] = momentum_coefficient * velocity_old["W" + str(l+1)] +
        ↪  weight_increment["W" + str(l+1)]
        velocity_new["b" + str(l+1)] = momentum_coefficient * velocity_old["b" + str(l+1)] +
        ↪  weight_increment["b" + str(l+1)]

        # With momentum
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - velocity_new["W" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - velocity_new["b" + str(l+1)]

    return parameters, velocity_new

def batchify(array, n, ax):
    """
    Split the dataset into batches of a specified size ready for mini-batch gradient descent
    """

    amount = dims // n
    return np.array_split(array, amount, axis=ax)

def shuffle(X, Y):
  """
  Shuffle the order of the training dataset images in the numpy array containing them.
  This can improve convergence.
  """

    shuffled_indices = [*range(0, train_n, 1)]
    np.random.shuffle(shuffled_indices)
    newX = np.zeros((dims, train_n))
    newY = np.zeros((train_n))

    for j, i in enumerate(shuffled_indices):
        newX[:, j] = trainxs_flat[:, i]
        newY[j] = trainys[i]

    return newX, newY

def L_layer_model(X, Y, layers_dims, learning_rate, momentum_coefficient, max_epochs,
↪   batch_size):
    # Specify the type of gradient descent being performed (will update plot titles
    ↪   automatically)
    train_type = "Mini Batch"

    # Keep track of loss and accuracy of each epoch
    loss = []
    accs = []
    converged = False
    i = 0

    dev_loss = []
    dev_accuracies = []

    best_dev_accuracy = 0

    parameters = build_model(layers_dims)

    # Empty the velocity
    velocity = {"W1": np.array([0.0]), "W2": np.array([0.0]), "W3": np.array([0.0]), "b1":
    ↪   np.array([0.0]), "b2": np.array([0.0]), "b3": np.array([0.0])} # 3-layer version
#     velocity = {"W1": np.array([0.0]), "W2": np.array([0.0]), "b1": np.array([0.0]), "b2":
↪   np.array([0.0])} #2-layer version

    while (converged == False) and (i <= max_epochs):
        newX, newY = shuffle(X, Y)

#       Mini-Batch if batch_size set >1 or SGD if batch size set = 1

        batchesX = batchify(newX, batch_size, 1)
        batchesY = batchify(newY, batch_size, 0)
        acc_total = 0
        loss_total = 0
        for b in range(len(batchesX)):
            bx = batchesX[b]
            by = batchesY[b]

            AL, caches = forward_pass(bx, parameters)
            loss_single_batch = loss_fun(AL, by)
            loss_total += loss_single_batch
            acc_single_batch = model_accuracy(AL, by)
            acc_total += acc_single_batch
            grads = backprop_loop(AL, by, caches)
            parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
            ↪   momentum_coefficient, velocity)
            velocity = velocity_new
        loss_fulldataset = loss_total/(len(batchesX))
        accuracy_fulldataset = acc_total/(len(batchesX))

        """
        # Full-Batch - uncomment this if you wish to run full batch


        AL, caches = forward_pass(newX, parameters)
        # prev = parameters
        loss_fulldataset = loss_fun(AL, newY)
        accuracy_fulldataset = model_accuracy(AL, newY)
        grads = backprop_loop(AL, newY, caches)
        parameters, velocity_new = update_parameters(parameters, grads, learning_rate,
↪   momentum_coefficient, velocity)
        velocity = velocity_new
        """

        #Add the epochs training loss and accuracy values to lists, ready for plotting
        loss.append(loss_fulldataset)
        accs.append(accuracy_fulldataset)

        #Print out the training loss and accuracy every 100 epochs
        if i % 100 == 0:
            print ("Loss after iteration %i: %f" % (i, loss_fulldataset))
            print ("Accuracy after iteration %i: %f" % (i, accuracy_fulldataset))

        #Calculate loss and accuracy on validation dataset using current epoch's parameters
        out = forward_pass(devxs_flat, parameters)
        dev_los = loss_fun(out[0], devys)
        dev_accuracy = model_accuracy(out[0], devys)
        if i == 0 or i % 1 == 0:
            dev_loss.append(dev_los)
            dev_accuracies.append(dev_accuracy)

        #Store the epochs results and parameters if it is the highest validation accuracy so
        ↪   far
        if dev_accuracy > best_dev_accuracy:
            best_dev_accuracy = dev_accuracy
            best_train_accuracy = accuracy_fulldataset
            best_epoch = i
            best_dev_parameters = parameters

        # Early stopping criteria
        if i >= 100:
            if (np.mean(dev_loss[-15:]) - np.mean(dev_loss[-30:-15])) > -.001:
                converged = True
                print("Model Converged")

        i += 1

    # format title of plots to reflect the hyperparameters and metrics
    title = "Best Validation Accuracy: " +  str(np.round(best_dev_accuracy, 3)) + " Train
    ↪   Accuracy: " + str(np.round(best_train_accuracy, 3)) +  " Type: " + train_type + "\n
    ↪   Learning rate = " + str(learning_rate)
    if momentum_coefficient != -1: # Omit momentum in title if not needed
```

```
      title += " Momentum = " + str(momentum_coefficient)
    if batch_size != -1: # Put -1 in batch_size to omit batch size from plot titles (e.g. if
    ↪   doing SGD)
      title += " Batch Size = " + str(batch_size)

    # plot the training and validation loss
    plt.plot(np.squeeze(loss), color='k')
    plt.plot(np.squeeze(dev_loss), color='r')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.title(title)
    plt.savefig(f"loss_plot_{str(np.round(best_dev_accuracy, 4))}_{best_epoch}.png", dpi=300)
    plt.show()

    # plot the training and validation accuracy
    plt.plot(np.squeeze(accs), color='k')
    plt.plot(np.squeeze(dev_accuracies), color='r')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.title(title)
    plt.savefig(f"accuracy_plot_{str(np.round(best_dev_accuracy, 4))}_{best_epoch}.png",
    ↪   dpi=300)
    plt.show()

    best_model = (best_dev_accuracy, best_train_accuracy, best_epoch, best_dev_parameters)

    return parameters, best_model

# Set dimensionality (number of nodes) of each layer
layers_dims = [784, 100, 60, 1]

# Specify hyperparameters to search through using grid search
learning_rates = [0.1,0.01]
batch_sizes = [32,64,256]
momentums = [0.9, 0.0]

stored_best_models = {}

# Loop through hyperparameters (grid search)
for i in range(len(learning_rates)):
    for j in range(len(momentums)):
        for b in range(len(batch_sizes)):
            rate = learning_rates[i]
            momentum = momentums[j]
            batch = batch_sizes[b]
            print("\n")
            print("\n")
            print(f"LEARNING RATE: {rate}")
            print(f"MOMENTUM: {momentum}")
            print(f"BATCH SIZE: {batch}")

            parameters, best_model = L_layer_model(trainxs_flat,
                                                   trainys,
                                                   layers_dims,
                                                    learning_rate = rate,
                                                    max_epochs=5000,
                                                    momentum_coefficient=momentum,
                                                    batch_size=batch)

            stored_best_models[str(rate) + "-" + str(momentum) + "-" + str(batch)] =
            ↪  best_model

            # Print out results for epoch with best validation accuracy
            best_dev_accuracy, best_train_accuracy, best_epoch, best_dev_parameters =
            ↪  best_model
            print(f"best_dev_accuracy: {best_dev_accuracy}")
            print(f"train_accuracy: {best_train_accuracy}")
            print(f"best_epoch: {best_epoch}")
```

Code for ensembling

```
#Ensembling the best models (those with highest validation accuracy)

#Accuracy on validation dataset
summed_activations = np.zeros((1,dev_n))
model_count = 0

for best_model in stored_best_models.values():
    best_dev_accuracy , _ , _ , parameters = best_model
    if best_dev_accuracy >= 0.95: #ensemble models with a validation accuracy in specified
    ↪  range
        model_count += 1
        activations, _ = forward_pass(devxs_flat, parameters)
        summed_activations += activations

val_ensembled_activations = summed_activations/model_count

val_accuracy = model_accuracy(val_ensembled_activations, devys)

print(f"Validation accuracy: {val_accuracy}")

#Accuracy on training dataset
summed_activations = np.zeros((1,train_n))
model_count = 0

for best_model in stored_best_models.values():
    best_dev_accuracy , _ , _ , parameters = best_model
    if best_dev_accuracy >= 0.95: #ensemble models with a validation accuracy in specified
    ↪  range
        model_count += 1
        activations, _ = forward_pass(trainxs_flat, parameters)
        summed_activations += activations

train_ensembled_activations = summed_activations/model_count

train_accuracy = model_accuracy(train_ensembled_activations, trainys)

print(f"Training accuracy: {train_accuracy}")
```