
# Multilayered Perceptron Assignment

Ronan Singpurwala

# Contents

# 1 The MLP Class

The code in the Jupyter Notebook defines a class called MLP that represents a multi-layer perceptron (MLP), a type of artificial neural network.

## 1.1 Constructor

```
class MLP:
    def __init__(self, m, n, o, train_inputs,
    ↪  train_outputs):
        # Set the number of input, hidden, and output
        ↪  neurons
        self.m = m
        self.n = n
        self.o = o

        # Create random weights for the input and hidden
        ↪  layers
        self.w2 = np.random.randn(self.n, self.m)
        self.w3 = np.random.randn(self.o, self.n)

        # Create random biases for the input and hidden
        ↪  layers
        self.b2 = np.random.randn(self.n, 1)
        self.b3 = np.random.randn(self.o, 1)

        # Store the training inputs and outputs
        self.train_inputs = train_inputs
        self.train_outputs = train_outputs
```

Listing 1: constructor function

The MLP has three layers: an input layer with m neurons, a hidden layer with n neurons, and an output layer with o neurons. The constructor (1) initializes the weights and biases of the input and hidden layers to random values, and stores the training data.

## 1.2 Feedforward (forward propogation)

```
def feedforward(self, xs):
    a2s = sigm(self.w2.dot(xs) + self.b2)
    a3s = sigm(self.w3.dot(a2s) + self.b3)
    return a3s
```

Listing 2: feedforward function

The feedforward function (2) calculates the output of the network given a set of inputs. It does this by taking the dot product of the inputs and the weights of the input layer, adding the biases of the input layer, and passing the result through the sigmoid function to produce the outputs of the hidden layer. The hidden layer outputs are then passed through the same process to produce the final output of the network.

## 1.3 Backpropogation

```
def backprop(self, xs, ys):
    del_w2 = np.zeros(self.w2.shape, dtype=float)
    del_b2 = np.zeros(self.b2.shape, dtype=float)
    del_w3 = np.zeros(self.w3.shape, dtype=float)
    del_b3 = np.zeros(self.b3.shape, dtype=float)
    cost = 0.0
    for x,y in zip(xs,ys):
        a1 = x.reshape(x.size, 1)
        z2 = self.w2.dot(a1) + self.b2
        a2 = sigm(z2)
        z3 = self.w3.dot(a2) + self.b3
        a3 = sigm(z3)
        delta3 = (a3 - (y.reshape(y.size, 1)) ) *
        ↪  sigm_deriv(z3)
        delta2 = sigm_deriv(z2) * (self.w3.T.dot(delta3))
        del_b3 += delta3
        del_w3 += delta3.dot(a2.T)
        del_b2 += delta2
        del_w2 += delta2.dot(a1.T)
        cost += ((a3 - (y.reshape(y.size, 1)) )**2).sum()

    n = len(ys)
    return del_b2/n, del_w2/n, del_b3/n, del_w3/n, cost/n
```

Listing 3: backpropogation function

The backprop function (3) performs the backpropagation algorithm to adjust the weights and biases of the network based on the training data. Backpropagation is a supervised learning algorithm that involves calculating the error between the predicted output of the network and the correct output, and then using that error to adjust the weights and biases in a way that minimizes the error. This is done through a process of gradient descent, where the gradients of the weights and biases are calculated with respect to the error, and the weights and biases are updated in the opposite direction of the gradient. This process is repeated for a specified number of epochs, with the learning rate controlling the size of the updates.

## 1.4 Predict

```
def predict(self, inputs):
    hidden_outputs = sigm(self.w2.dot(inputs.T) + self.b2)
    final_outputs = sigm(self.w3.dot(hidden_outputs) +
    ↪  self.b3)
    return final_outputs[0][0]
```

Listing 4: predict function

The predict function (4) applies the trained weights and biases of the network to make predictions given new inputs. It does this by passing the inputs through the same process as in the feedforward function (2) to produce the final output of the network.

## 1.5 Train

```
def train(self, epochs, eta):
    xs = self.train_inputs
    ys = self.train_outputs
    cost = np.zeros((epochs,))

    for e in range(epochs):
        d_b2, d_w2, d_b3, d_w3, cost[e] =
        ↪  self.backprop(xs,ys)

        self.b2 -= eta * d_b2
        self.w2 -= eta * d_w2
        self.b3 -= eta * d_b3
        self.w3 -= eta * d_w3
    plt.plot(cost)
    return cost
```

Listing 5: train function

Finally, the train function (5) trains the network by repeatedly calling the feedforward (2) and backprop (3) functions for a specified number of epochs and learning rate. Weights and biases of the network are updated accordingly every iteration. It also keeps track of the cost (i.e. the error) at each epoch to monitor the training progress. Once all epochs are completed, the cost is plotted on a graph using the Matplotlib library.

# 2 Question 1 - Basic Neural Network

In this question, we are given a set of inputs that we must train to meet the expected outputs. We are also given the number of neurons for each layer.

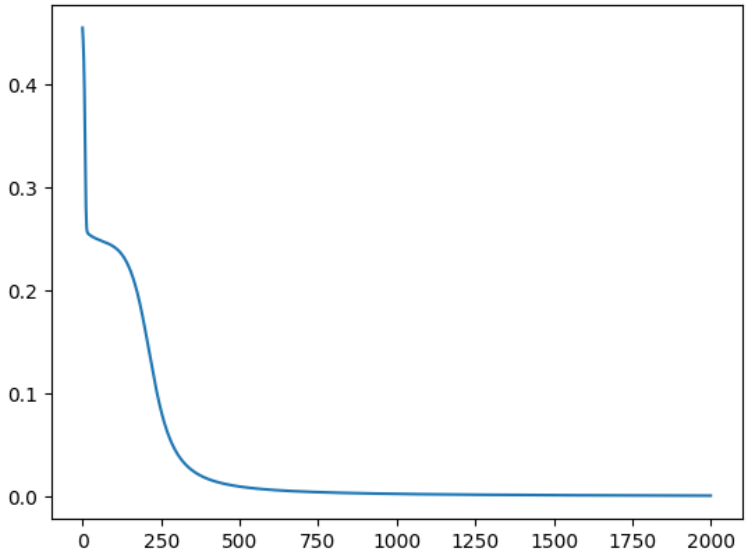
```
m = 3
n = 4
o = 1

# iterations and eta
epochs = 2000
learning_rate = 3

# Create the training inputs and outputs
train_inputs = np.array([[0,0,1],
                         [0,1,1],
                         [1,0,1],
                         [1,1,1]])

train_outputs = np.array([[0],
                          [1],
                          [1],
                          [0]])
```

Listing 6: Question 1 - training inputs and outputs

The code in the Jupyter Notebook sets the number of neurons in the input layer (m), hidden layer (n), and output layer (o) to 3, 4, and 1, respectively. It then creates training data consisting of four input vectors and their corresponding outputs.

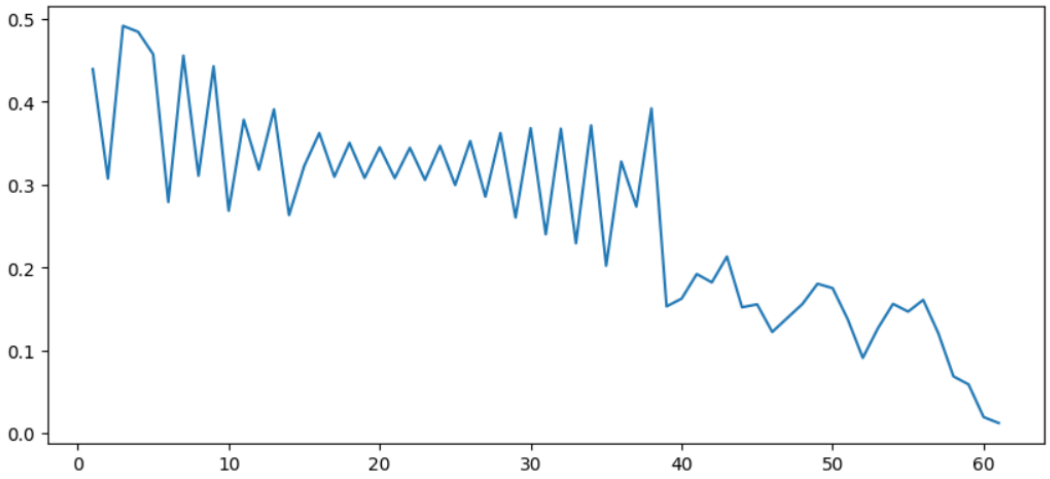
Next, the code creates an instance of the MLP class with the given number of neurons and training inputs and outputs, and trains the network using the train function (5). It then calculates the outputs of the network before and after training, and prints them. Finally, it plots the cost (i.e. the error) at each epoch during training.

```
before:
[[0.70640056 0.73489734 0.693573   0.72065097]]

after:
[[0.03325924 0.97088568 0.96872174 0.03037113]]
```




After training, the network's outputs before and after training are printed. The results show that the network's outputs are closer to the correct outputs after training than before. This indicates that the network has learned to make better predictions on this data set. The plots of the cost should show how the error decreases as the network is trained.

## 2.1 Unstable Learning Rate




The learning rate is a hyper-parameter that determines the step size at which the model updates the weights during training. I noticed if the learning rate is very high, the model overshoots the minimum and oscillate or sometimes even diverge. This lead to unstable and erratic training, resulting in poor performance. This can be seen in the cost graph above where the learning rate is set 50.

# 3 Question 2 - 3 inputs and 2 outputs

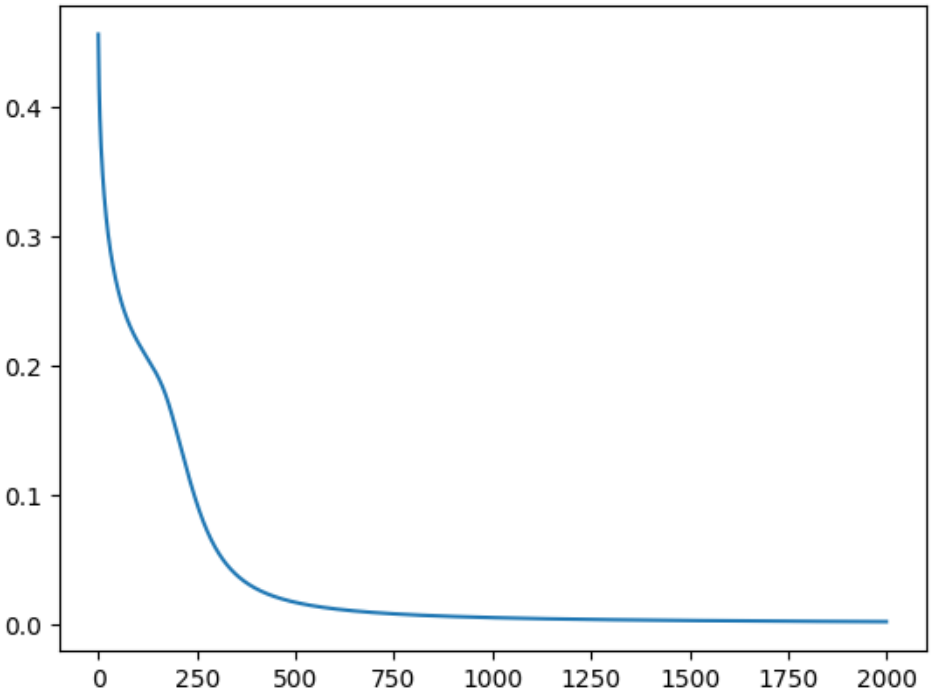
```
# Set the number of input, hidden, and output neurons
m = 3
n = 5
o = 2

# iterations and eta
epochs = 2000
learning_rate = 2

# Create the training inputs and outputs
train_inputs = np.array([[1, 1, 0],
                         [1,-1,-1],
                         [-1,1, 1],
                         [-1,-1,1],
                         [0, 1,-1],
                         [0,-1,-1],
                         [1, 1, 1]])

train_outputs = np.array([[1, 0],
                          [0, 1],
                          [1, 1],
                          [1, 0],
                          [1, 0],
                          [1, 1],
                          [1, 1]])
```

Listing 7: Question 2 - training inputs and outputs

This question is similar to question 1 however we need more input neurons. The code creates an instance of the MLP class with 3 input neurons, 5 hidden neurons, and 2 output neurons. It then sets the number of epochs, learning rate, training inputs and training outputs for training the model.

The feedforward function (2) is called on the training inputs before and after training to get the output of the network before and after training.

```
before:
[[0.35196225 0.37900102 0.44421457 0.5359174  0.29728359 0.39824774
  0.39990945]
 [0.42767486 0.69467732 0.37434786 0.60538134 0.44199129 0.68777179
  0.42815541]]

after:
[[0.9780567  0.05046172 0.99957399 0.99990216 0.99102103 0.95955517
  0.99081461]
 [0.044846   0.98435272 0.97159185 0.03816033 0.01191815 0.96706732
  0.95810279]]
```




The train function (5) is called to perform training, and the resulting cost is plotted. We can see how the mean squared error changes over the course of training as the weights and biases of the network are adjusted. The results show that the network's outputs are closer to the correct outputs after training than before. This indicates that the network has learned to make better predictions on this data set.

# 4 Question 3 - Transportation Mode Choice

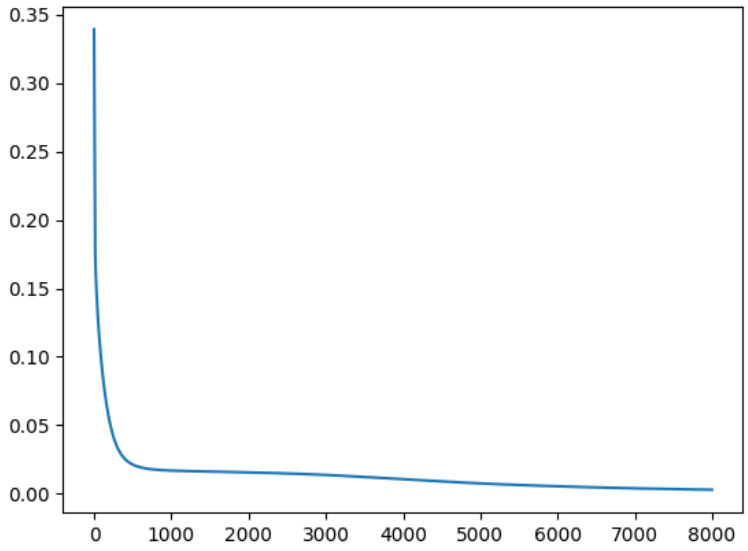
In this question, we are given training data from a transportation study regarding the mode choice to select bus, car or train among commuters along a major route in a city, gathered through a questionnaire study. With this data, the code can predict what transportation mode that person would take depending on the inputs.

| | gender | car_ownership | travel_cost | income_level | transportation_mode |
|---|---|---|---|---|---|
| **0** | 1 | 0.0 | 0.0 | 0.0 | 0.0 |
| **1** | 1 | 0.5 | 0.0 | 0.5 | 0.0 |
| **2** | 0 | 0.5 | 0.0 | 0.5 | 0.5 |
| **3** | 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **4** | 1 | 0.5 | 0.0 | 0.5 | 0.0 |
| **5** | 1 | 0.0 | 0.5 | 0.5 | 0.5 |
| **6** | 0 | 0.5 | 0.5 | 0.5 | 0.5 |
| **7** | 0 | 0.5 | 1.0 | 1.0 | 1.0 |
| **8** | 1 | 1.0 | 1.0 | 0.5 | 1.0 |
| **9** | 0 | 1.0 | 1.0 | 1.0 | 1.0 |

The code is training the model to predict a person's preferred mode of transportation based on their gender, car ownership status, travel cost, income level. The input data is first read from a CSV file and then converted into numerical values, with each categorical variable being mapped to a numerical value. The resulting data is then used to create the training inputs and outputs for the neural network.

```
predict inputs: [[0.  0.  1.  0.5]]
prediction: 0.9191168628951103
transport mode: Car
```

After the training data has been prepared and the MLP has been trained, the code is using the trained network to make a prediction on a single input and printing the prediction. The prediction is made by calling the predict function (4) of the MLP instance and passing the input to be predicted. The output of the predict method is the predicted preferred transportation mode, which is then mapped back to the original categorical values (bus, train, or car) using a dictionary.




We also plot the cost to visualize the progress of the training process as done in the previous two questions.

# 5 Question 4 - Iris flower classification

In question, we are asked to train the iris data set and be able to predict the name of the iris based on inputs.

| | sepal_length | sepal_width | petal_length | petal_width | species |
|---|---|---|---|---|---|
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | 0.0 |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | 0.0 |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | 0.0 |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | 0.0 |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | 0.0 |
| **...** | ... | ... | ... | ... | ... |
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | 1.0 |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | 1.0 |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | 1.0 |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | 1.0 |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | 1.0 |

This code is processing a CSV file containing data on iris flowers, including their sepal length, sepal width, petal length, and petal width. It is converting the categorical data in the CSV file (the species of the iris) into numerical values and using these numerical values as the target outputs for the training data (similar question 3). It is then using the sepal length, sepal width, petal length, and petal width as the inputs for the training data.

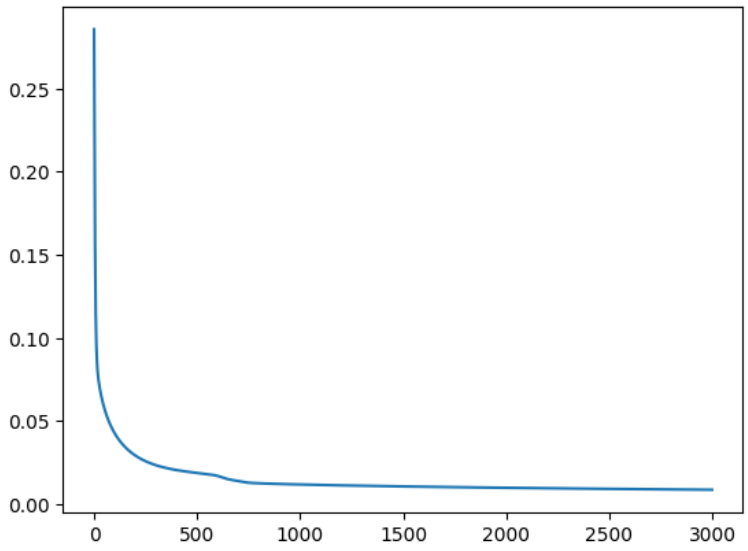
```
train_outputs = iris_df["species"].to_numpy()
train_inputs = iris_df[["sepal_length", "sepal_width",
↪  "petal_length", "petal_width"]].to_numpy()
predicton_input = np.array([4.6, 3.3, 1.5, 0.2])

# Set the number of input, hidden, and output neurons
m = 4
n = 6
o = 1

# iterations and eta
epochs = 3000
learning_rate = 1
```

Listing 8: Question 4 - training inputs and outputs

The code defines an MLP with 4 input neurons, 6 hidden neurons, and 1 output neuron. It is training the network on the prepared training data by calling the train function (4) of the MLP instance and passing it the number of epochs and the learning rate.

```
predict inputs: [4.6 3.3 1.5 0.2]
prediction: 0.03889368430600557
flower: Iris-setosa
```

After training, the code is using the trained network to make a prediction on a single input and printing the prediction. The prediction is made by calling the predict function (4) of the MLP instance and passing it the input to be predicted. The output of the predict method is then mapped back to the original categorical values (Iris-setosa, Iris-versicolor, or Iris-virginica) using a dictionary.




We also plot the cost to visualize the progress of the training process as done in the previous two questions.

## 5.1 Extra Dataset - Wheat Seeds

I also used the Wheat Seeds dataset to train my network in the same way as the I did with the Iris dataset. The seed classification is performed based on 7 physical features: area of wheat, perimeter of wheat, compactness, length of the kernel, width of the kernel, asymmetry coefficient, and kernel groove length. The dataset is collected from the UCI library and has 200 occurrences of wheat kernels. It contains three distinct types of wheat kernels: (Kama, Rosa, Canadian) designated as numerical variables 1, 2 and 3 respectively.

```
predict inputs: [ 1.3    13.34    0.8684  5.243   5.9    10.637   5.063 ]
prediction: 0.9999644525638054
wheat kernel: Kama
```