MLP Assignment

Exercise 1: copy and adapt the above XOR_MLP code so that it uses 3 neurons in the hidden layer. Train such a MLP and see if it learns faster than the previous one.

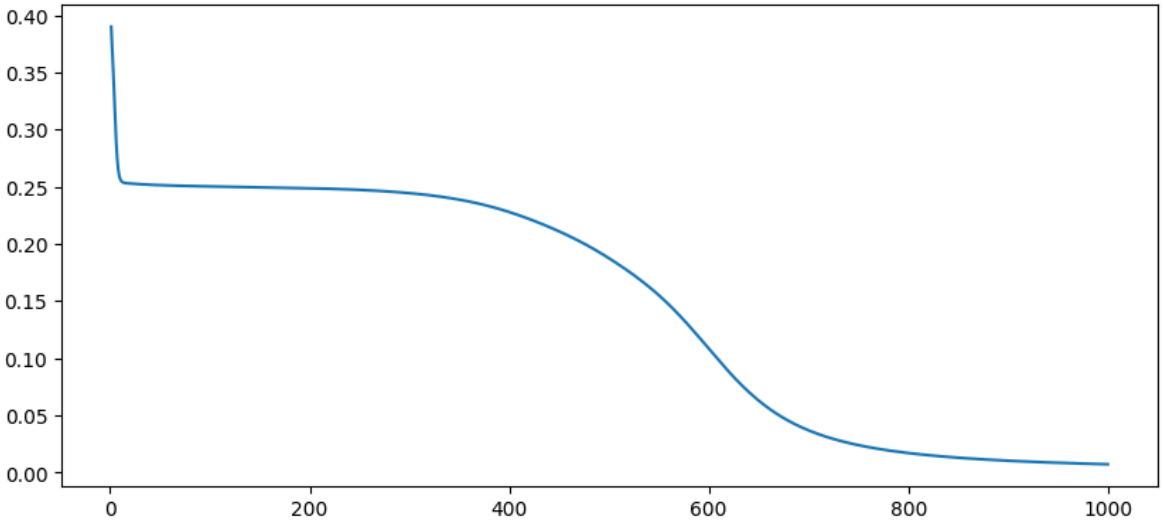
To achieve this I edited this piece of code to change the previously 2 hidden layers to 3 hidden layers

```
# hidden layer of 3 neurons
self.w2 = np.random.randn(3,2)
self.b2 = np.random.randn(3,1)

# output layer has 1 neuron
self.w3 = np.random.randn(1,3)
self.b3 = np.random.randn(1,1)
```

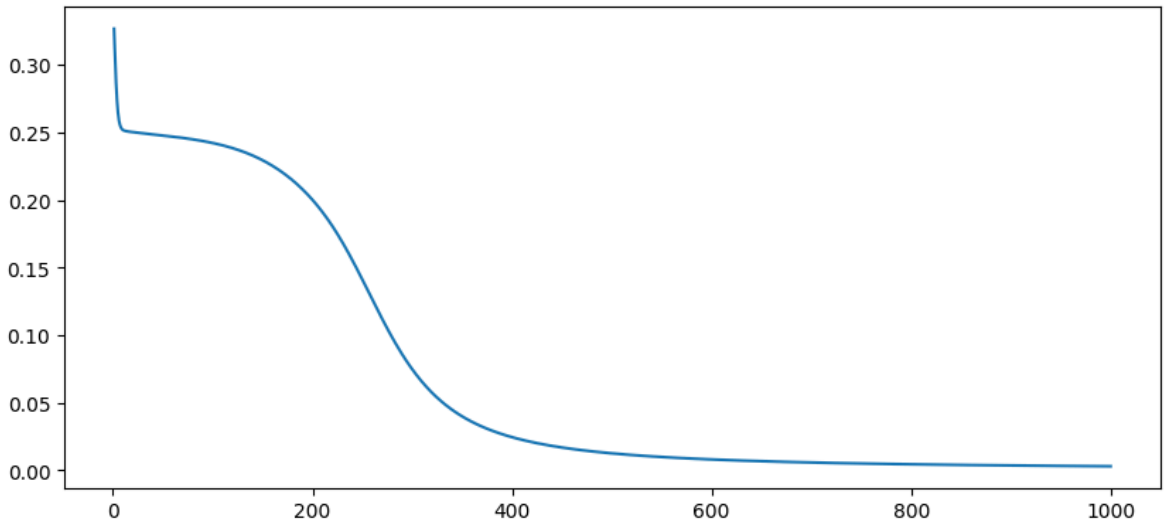
To show how the effect of adding a hidden layer changed the learning rate, I made 3 graphs to show the cost value change at different stages in the epochs.

Graph 1: This graph shows a global view of how the cost changes over all epochs.

2 hidden neuron's:



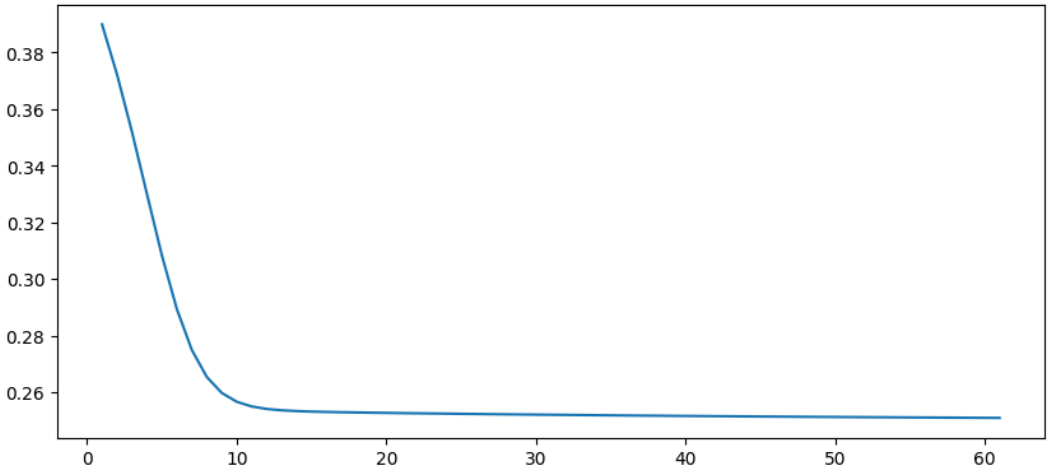
3 hidden neuron's:



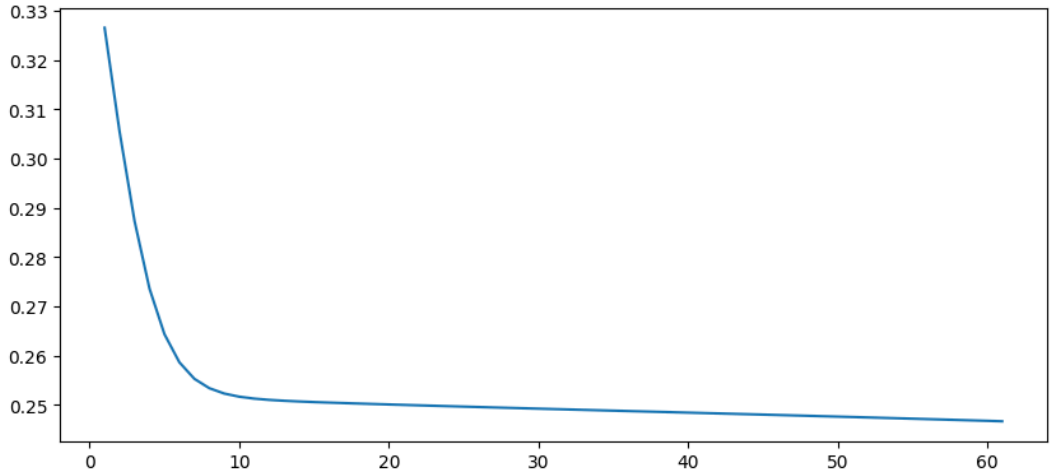
Already from this we can tell the perceptron learns a lot quicker with a drop off coming just after 200 epochs for the perceptron with 3 hidden neutrons and a drop off around 600 epochs for the perceptron with 2 hidden neuron's

Graph 2: Zooms into the first 61 epochs, which often shows rapid changes in the cost

2 hidden neuron’s

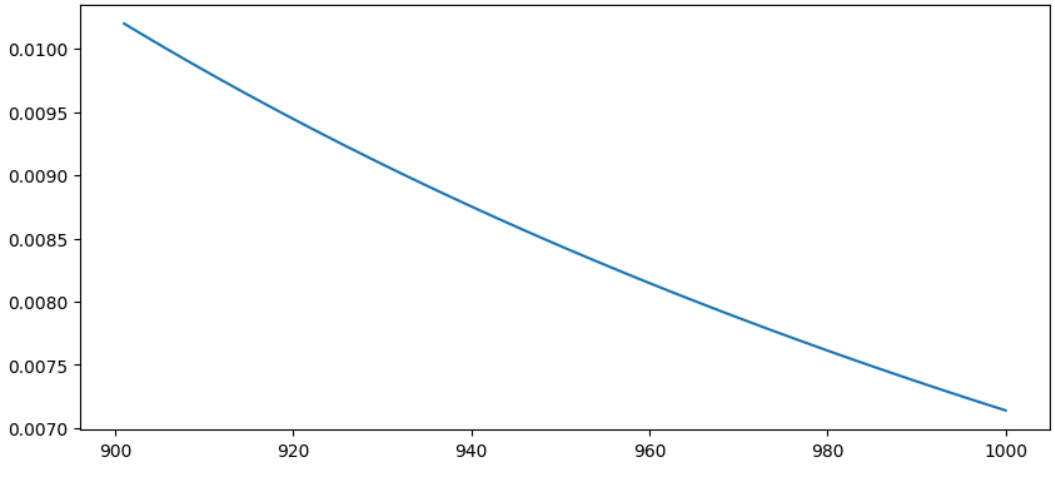


3 hidden neuron’s

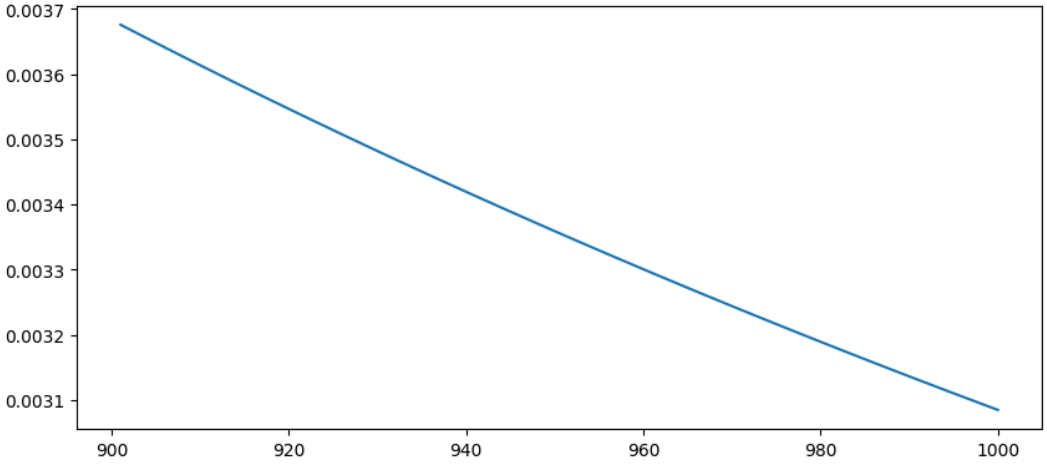


The initial phase for both perceptrons is not drastically changed at the beginning but there is a slight advantage for the second perceptron, achieving an error 0.01 less than the first perceptron over the first 61 epochs.

Graph 3: focuses on the convergence phase, where the cost often decreases slowly and stabilises.
2 hidden layers



3 hidden layers



As we can see from these 2 graphs the rate of change in cost in the final 100 epochs is much quicker for the first perceptron because the second perceptron has already reached a very low number, about half that of the first perceptron. As the cost decreases the longer it takes to decrease its value.

In conclusion, the perceptron with 3 hidden neuron's is significantly quicker than the perceptron with just 2 hidden neuron's especially during the middle of the iterations, reaching a lower number significantly quicker and by the end of the test, the cost is around 50% the value of the first perceptrons's cost.

Generalised MLP with parameters m, n and o for input, hidden and output neuron's

1. Generalised weight and bias initialisation

```
# Initialize weights and biases
self.w2 = np.random.randn(n, m)  # Hidden layer weights
self.b2 = np.random.randn(n, 1)  # Hidden layer biases

self.w3 = np.random.randn(o, n)  # Output layer weights
self.b3 = np.random.randn(o, 1)  # Output layer biases
```

I changed the hard coded values to implement the m input neuron's, n hidden neuron's, and the o output neuron's.
This allows the map to handle any number of input, hidden and output neuron's, making this perceptron adaptable to different problems.

2. Flexible input and output handling

```
a1 = x.reshape(-1, 1)  # Input as column vector
y = y.reshape(-1, 1)  # Expected output as column vector
```

Previously the input dimension was fixed to 2. The reshape method now dynamically adjusts based on the number of input neuron's m.
This also ensure the expected output is shaped correctly as a column vector.

3. Training Process

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

Training data is passed as parameters to the train method, instead of being hardcoded, allowing the MLP to be trained on different datasets without modifying the class.

4. Added prediction function

```
def predict(self, xs):
    outputs = self.feedforward(xs)
    return (outputs > 0.5).astype(int)
```

This method was added to make predictions after training.

5. Improved plotting

```
# Plot training cost
plt.plot(range(epochs), cost)
plt.title(f"Learning Rate: {eta}")
plt.xlabel("Epochs")
plt.ylabel("Cost")
plt.show()
```

I added to the plot to make the graphs more understandable and to display relevant information such as the eta

Problem 1:
This problem required me to match these inputs to these outputs using my previously created MLP and adjust the learning rate to see which value will make the learning unstable.

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

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

.T is used to transpose the data to align with the expected format of the neural network

We initialise m, n and o

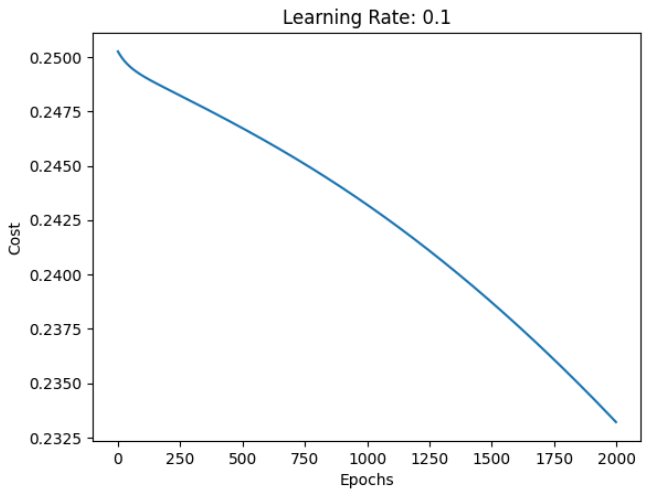
```
# Initialize the MLP
m = inputs.shape[0]
n = 4
o = outputs.shape[0]
```

Shape counts the columns for inputs and outputs to set the number of input and output neuron's. I was told to set hidden neuron's to 4

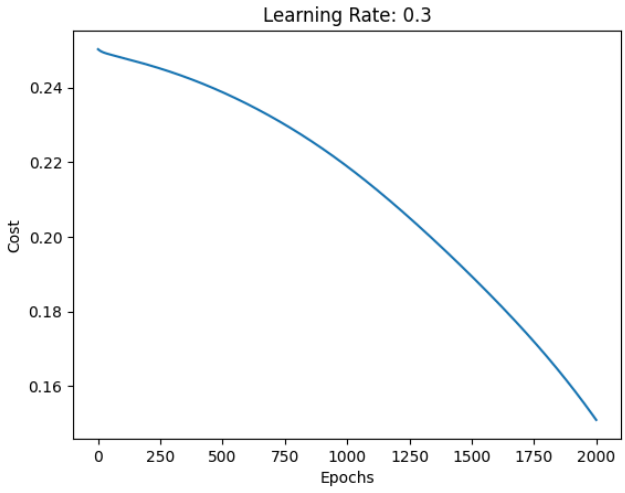
I then created the MLP with these values and tested it against multiple learning rates at 2000 epochs. The costs are plotted inside the MLP and I added code to print the inputs alongside the predicted outputs to check if the perceptron had been successful in training the data.

```
# Train the MLP with different learning rates
epochs = 2000
learning_rates = np.array([0.1, 0.3, 0.5, 1, 5, 10, 20])

for eta in learning_rates:
    # Reinitialize the MLP for each learning rate to reset weights and biases
    mlp = MLP(m, n, o)

    print(f"\nTraining with Learning Rate: {eta}")
    mlp.train(inputs, outputs, epochs, eta)

    # Test the MLP
    predictions = mlp.predict(inputs)

    # Print the predictions
    print("Input and Predicted Output:")
    for i in range(inputs.shape[1]):
        print(f"Input: {inputs[:, i]}, Predicted Output: {predictions[:, i][0]}")
```

For eta 0. 1 and 0.3 the perceptron was unsuccessful and gave me incorrect outputs.
But 0.3 eta had a lower cost

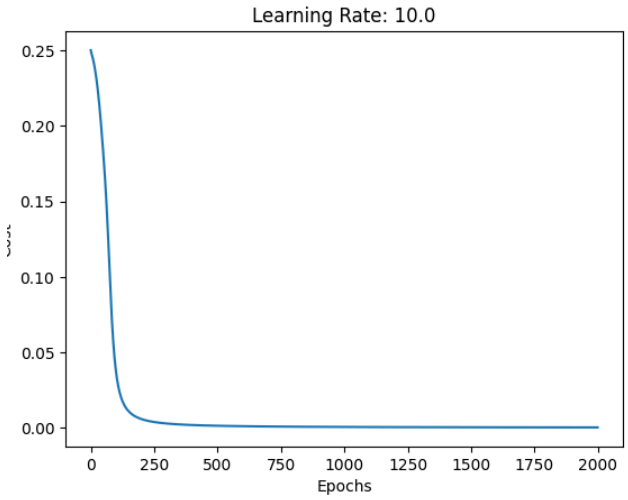


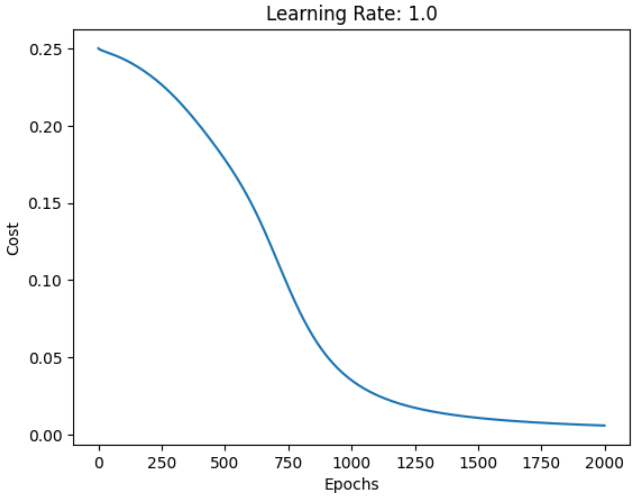

```
Input and Predicted Output:
Input: [0 0 1], Predicted Output: 0
Input: [0 1 1], Predicted Output: 1
Input: [1 0 1], Predicted Output: 0
Input: [1 1 1], Predicted Output: 1
```



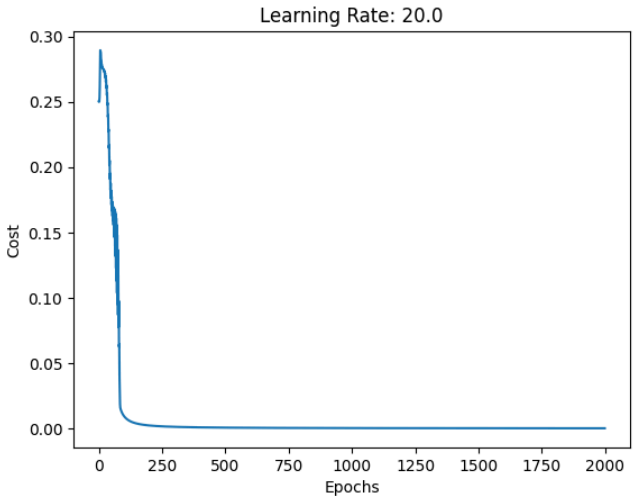

```
Input and Predicted Output:
Input: [0 0 1], Predicted Output: 0
Input: [0 1 1], Predicted Output: 1
Input: [1 0 1], Predicted Output: 1
Input: [1 1 1], Predicted Output: 1
```

At eta 0.5 the target output was finally reached but was not very quick, reaching a cost of below 0.5 at around 2000 epochs.
As the learning rate was increased from here the perceptron became faster every time at reducing the cost and achieving the correct output.



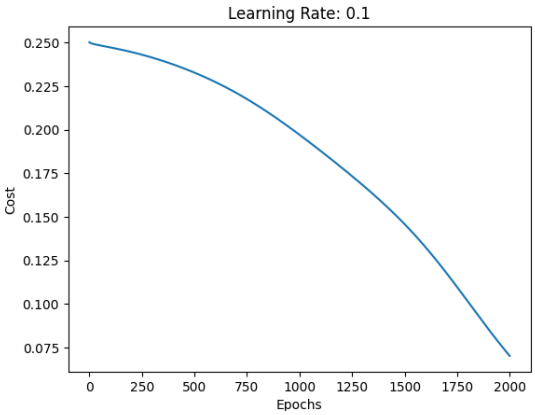


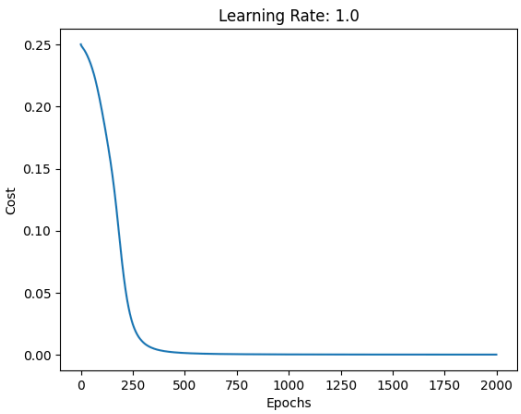

```
Input and Predicted Output:
Input: [0 0 1], Predicted Output: 0
Input: [0 1 1], Predicted Output: 1
Input: [1 0 1], Predicted Output: 1
Input: [1 1 1], Predicted Output: 0
```



I stopped increasing the eta at 2 as it was a near vertical line and felt the number was getting too high and the weight values would be unnecessarily shifted a lot.
From observing these graphs, it was clear that a higher eta suited this sample data in achieving the required results quickly and a eta of 0.3 or below would be too slow and not even achieve the result in 2000 epochs.

Using cross-entropy cost function greatly improves learning





For the eta 0.1 and 1 the cost value decreases at a much higher than before I implemented the cross-entropy cost function.

Problem 2:
For problem 2 I was told to match these input to these output values and test the perceptron with different numbers of hidden neuron's and observe the change

```
X_training = np.array([
    [1, 1, 0],
    [1, -1, -1],
    [-1, 1, 1],
    [-1, -1, 1],
    [0, 1, -1],
    [0, -1, -1],
    [1, 1, 1]
]).T
```

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

I used these values for my hidden neuron's which I thought were different enough to observe a meaningful change

```
hidden_neurons_list = [2, 4, 8, 16, 64]  # Different hidden layer sizes
epochs = 2000
eta = 0.3  # Learning rate
```

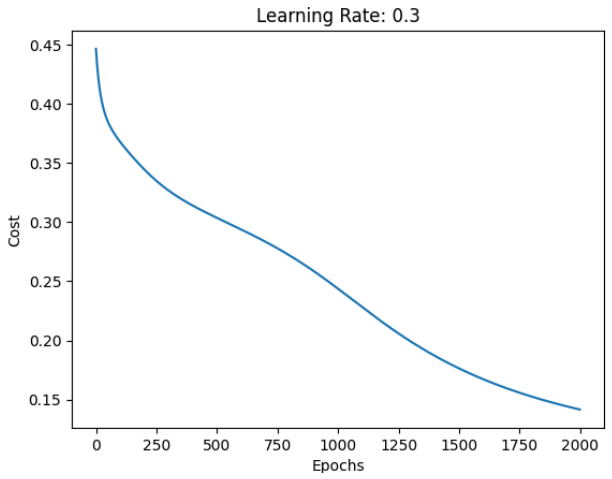
I iterated through this array of hidden neuron's and tested the perceptron training against every value

```
for n in hidden_neurons_list:
    print(f"\nTraining MLP with {n} hidden neurons:")
    mlp = MLP(m, n, o)  # Initialize the MLP with n hidden neurons
    mlp.train(X_training, y_training, epochs, eta)

    # Test the network
    predictions = mlp.predict(X_training)
```

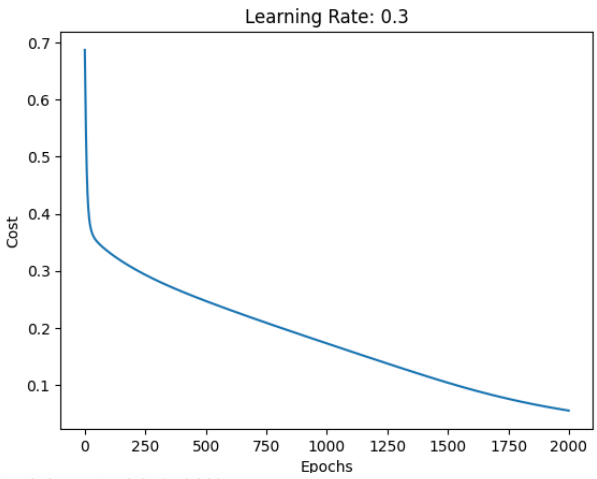
I found that every hidden neuron value reached the expected value for each input in the 200 epochs, but the speed at which it reduced the cost changed each time.

```
Input and Predicted Output:
Input: [1 1 0], Predicted Output: [1 0], Required output: [1 0]
Input: [ 1 -1 -1], Predicted Output: [0 1], Required output: [0 1]
Input: [-1  1  1], Predicted Output: [1 1], Required output: [1 1]
Input: [-1 -1  1], Predicted Output: [1 1], Required output: [1 0]
Input: [ 0  1 -1], Predicted Output: [1 0], Required output: [1 0]
Input: [ 0 -1 -1], Predicted Output: [1 1], Required output: [1 1]
Input: [1 1 1], Predicted Output: [1 1], Required output: [1 1]
```

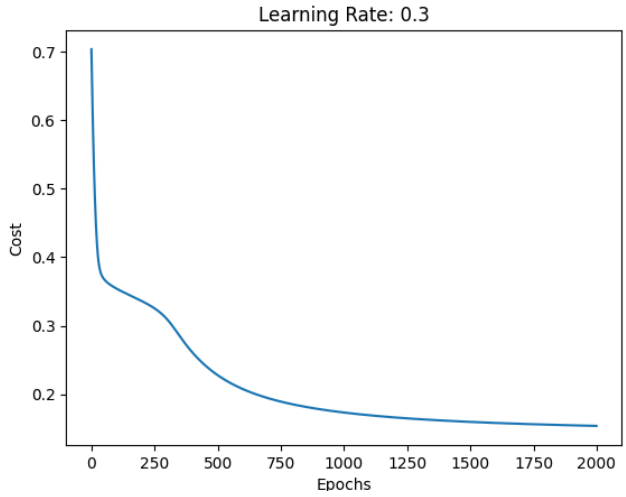
Training MLP with 2 hidden neurons:



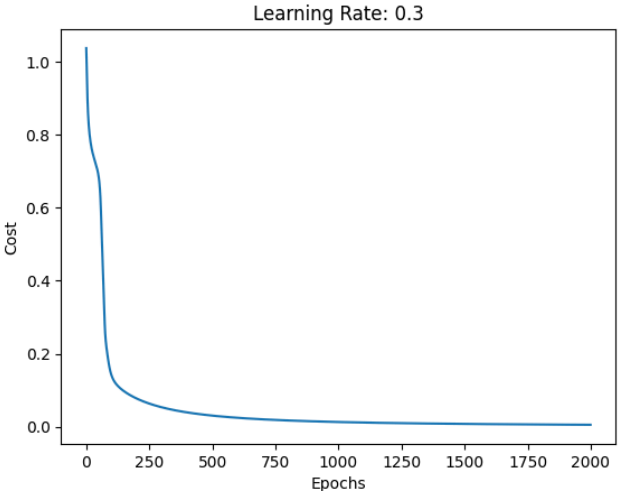
Training MLP with 8 hidden neurons:



Training MLP with 16 hidden neurons:

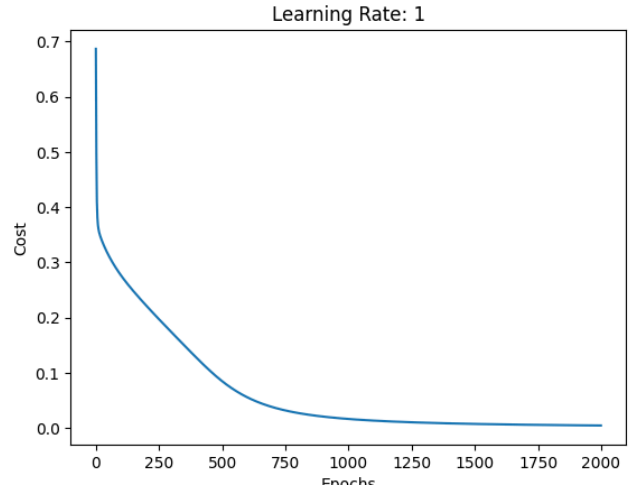


Training MLP with 64 hidden neurons:

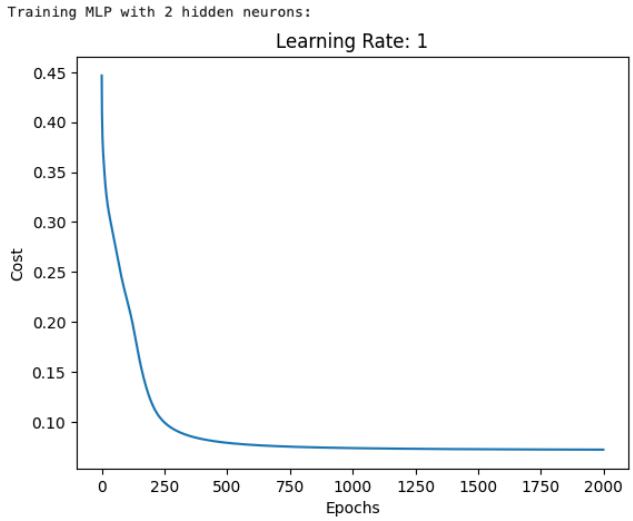


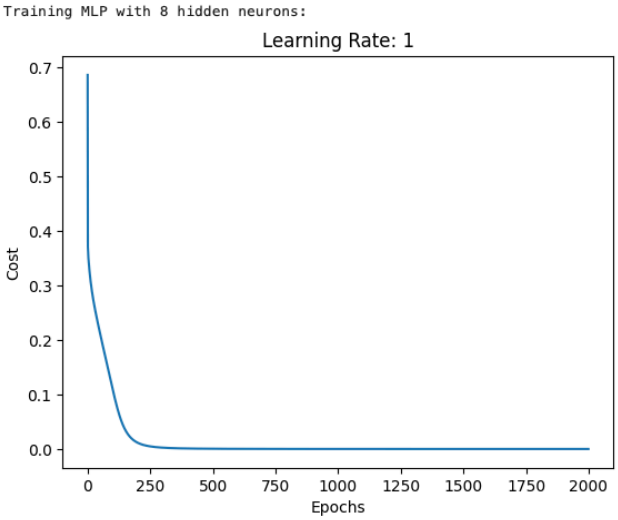
Clearly as the number of hidden neuron's in increased the learning rate for the perceptron is increased drastically.
I also tested with an eta of 1 and 8 hidden neuron's which also sped up the learning rate and made it look like the plot with 16 neuron's at an eta of 0.3.
In conclusion I found that increasing both the number of hidden neuron's and the value of the eta, also increase the learning speed of the perceptron.

Training MLP with 8 hidden neurons:



I tested this with the cross-entropy cost function





The tests for 2 and 8 hidden neuron's are drastically improved with a sharp decreasing of the cost value early on indicating fast learning, compared to the previous slower learning rate with the mean-squared error cost function.

Problem 3
Train the perceptron against the sample data then test it against another input.
I transformed this training data to this code to represent each input as numbers:

Gender	Car ownership	Travel Cost	Income Level	Transportation mode
Male	0	Cheap	Low	Bus
Male	1	Cheap	Medium	Bus
Female	1	Cheap	Medium	Train
Female	0	Cheap	Low	Bus
Male	1	Cheap	Medium	Bus
Male	0	Standard	Medium	Train
Female	1	Standard	Medium	Train
Female	1	Expensive	High	Car
Male	2	Expensive	Medium	Car
Female	2	Expensive	High	Car

```
X_training = np.array([
    [0, 0, 0, 0],  # Male, 0 cars, C
    [0, 1, 0, 1],  # Male, 1 car, Ch
    [1, 1, 0, 1],  # Female, 1 car,
    [1, 0, 0, 0],  # Female, 0 cars,
    [0, 1, 0, 1],  # Male, 1 car, Ch
    [0, 0, 1, 1],  # Male, 0 cars, S
    [1, 1, 1, 1],  # Female, 1 car,
    [1, 0, 2, 2],  # Female, 0 cars,
    [0, 2, 2, 1],  # Male, 2 cars, E
    [1, 2, 2, 2],  # Female, 2 cars,
])

# Output data (transportation mode:
Y_training = np.array([
    [1, 0, 0],  # Bus
    [1, 0, 0],  # Bus
    [0, 1, 0],  # Train
    [1, 0, 0],  # Bus
    [1, 0, 0],  # Bus
    [0, 1, 0],  # Train
    [0, 1, 0],  # Train
    [0, 0, 1],  # Car
    [0, 0, 1],  # Car
    [0, 0, 1],  # Car
```

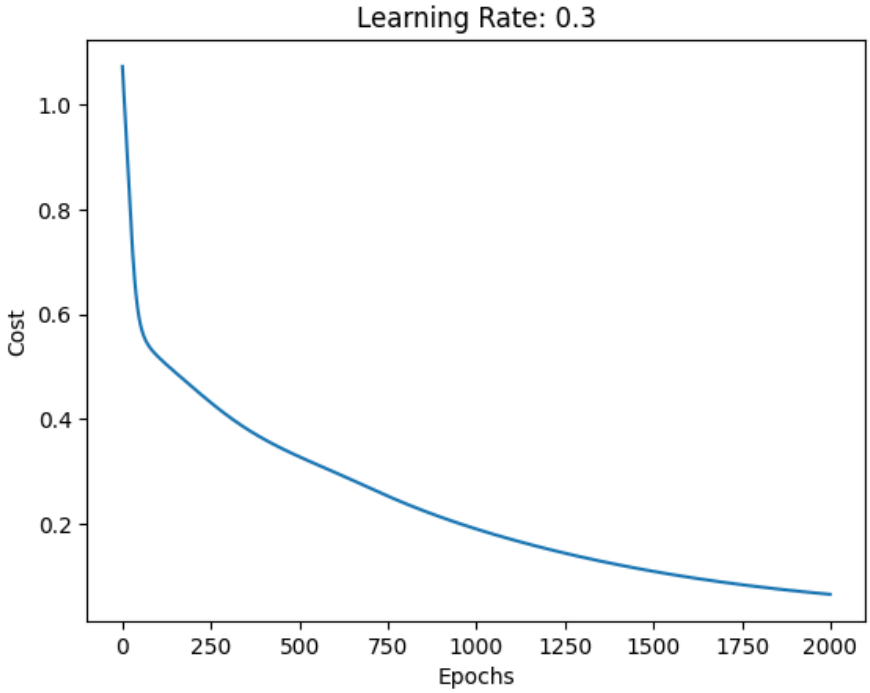
The test input was a woman with no car, willing to pay expensive travel cost and has a medium income level. I represented that as values that the perceptron could interpret and tested it with these settings:

```
epochs = 2000
eta = 0.3
mlp.train(X_training.T, Y_training.T, epochs, eta)

# Predict for a new input
new_input = np.array([[1, 0, 2, 1]]).T  # Female,
```

In previous problems 2000 epochs and 0.3 eta was usually enough to get the correct answer. I added the list classes to correspond the mode of transport to the output of the neural network. The function np.argmax( ) returns the index of the largest value from prediction array which will then correspond to the name of the mode of transport and be printed on the screen.

```
prediction = mlp.predict(new_input)

# Interpret the prediction
classes = ["Bus", "Train", "Car"]
predicted_class = classes[np.argmax(prediction)]
print(f"Predicted Transportation Mode: {predicted_class}")
```

My output from this trial:




```
Predicted Transportation Mode: Train
```

The error decreases from 1 to around 0.1 in the 200 epochs and the predicted value matches the expected value of Train.

To copy this data to a data frame and save it as a css file on my machine I had to re format the data to this:

```
data = {
    "Gender": [1, 1, 0, 0, 1, 1, 0, 0, 1, 0],  # Male = 1
    "Car ownership": [0, 1, 1, 0, 1, 0, 1, 0, 2, 2],
    "Travel Cost": [0, 0, 0, 0, 0, 1, 1, 2, 2, 2],  # Che
    "Income Level": [0, 1, 1, 0, 1, 1, 1, 2, 1, 2],  # Lo
    "Transportation Mode": [0, 0, 1, 0, 0, 1, 1, 2, 2, 2]
}
df = pd.DataFrame(data)
```

Formatting the data as a dictionary allows the panda method DataFrame to to convert it into a data frame.
I then extract the input features from the data frame and convert then into a NumPy array suitable for training and convert the transportation mode column into a one-hot encoded NumPy array which is suitable for neural networks, and save the data frame to a css file names transport.csv.

```
inputs = df[["Gender", "Car ownership", "Travel Cost", "Income Level"]].to_numpy().T
outputs = pd.get_dummies(df["Transportation Mode"]).to_numpy().T  # One-hot encode the output

# Save to transport.csv
df.to_csv("transport.csv", index=False)
```

Problem 4:
This problem requires me to download the iris_data.csv dataset and load it as a dataset then train the perceptron on this dataset so it can make predictions on the type of flower according to its measurements.

```
df = pd.read_csv("iris_data.csv", header=None)
```

There was no header in the civ file.

```
df.columns = ["Sepal Length", "Sepal Width", "Petal Length", "Petal Width", "Species"]

# Prepare inputs (features) and outputs (labels)
inputs = df[["Sepal Length", "Sepal Width", "Petal Length", "Petal Width"]].to_numpy().T
```

I then set column names and select the input columns and convert them to arrays in the inputs variable.

In order to convert the categorical data flower names I used LabelEncoder form the scikit-Learn library. I could have hard coded this but chose to use the general approach so I could use this code again for datasets where I didn't know all of the categorical data.

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
encoder = LabelEncoder()
outputs = encoder.fit_transform(df["Species"])
outputs = pd.get_dummies(outputs).to_numpy().T
```

This assigns a unique label a unique integer.
e.g:
Iris-setosa -> 0
Iris-versicolor -> 1
Iris-virginica -> 2

fit_transfrom fits the encoder into a column, transforming the original labels into their corresponding numeric values
get_dummies and .to_numpy one-hot codes the outputs
This converts the now categorical label into a vector of binary values, where only the index corresponding to the label is 1, and all indexes are 0.
e.g Bus -> [1, 0, 0], Train -> [0, 1, 0]

I assign the appropriate values to m and o and set the amount of hidden layers to 8. Then create the map instance

```
epochs = 2000
eta = 0.3
mlp.train(inputs, outputs, epochs, eta)
```

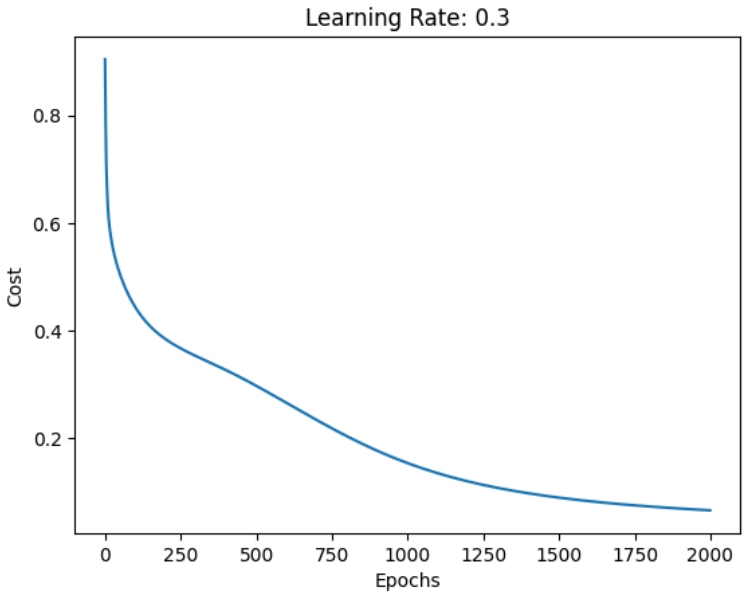
I chose to train the neural network with 2000 epochs and 0.3 eta as this is my usual starting point now.
I use the predict method to predict the values for the entire dataset then decode the prediction outcomes into species names.

```
predictions = mlp.predict(inputs)

# Decode predictions back to species names
predicted_labels = [encoder.inverse_transform([np.argmax(predictions[:, i])])[0] for i in range(predictions.shape[1])]
```

This converts the numeric predictions back to the original species name. The trained labelEncoder is used to do this.
Argmax finds the index with the maximum value in each prediction.

```
actual_labels = [encoder.inverse_transform([np.argmax(outputs[:, i])])[0] for i in range(outputs.shape[1])]

# Compare predictions with actual labels
print("Predictions vs Actual Labels:")
for i in range(len(actual_labels)):
    print(f"Sample {i + 1}: Predicted - {predicted_labels[i]}, Actual - {actual_labels[i]}")
```

Actual values converts the hot-encoded outputs back to the original species name similar to the previous step so wen can compare the predicted labels against the actual labels.
A for loop is used to iterate through each sample in the dataset and prints the index, prediction and actual label.
This is the result:




```
Predictions vs Actual Labels:
Sample 1: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 2: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 3: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 4: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 5: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 6: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 7: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 8: Predicted - Iris-setosa, Actual - Iris-setosa
Sample 94: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 95: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 96: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 97: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 98: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 99: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 100: Predicted - Iris-versicolor, Actual - Iris-versicolor
Sample 101: Predicted - Iris-virginica, Actual - Iris-virginica
Sample 102: Predicted - Iris-virginica, Actual - Iris-virginica
Sample 103: Predicted - Iris-virginica, Actual - Iris-virginica
Sample 104: Predicted - Iris-virginica, Actual - Iris-virginica
Sample 105: Predicted - Iris-virginica, Actual - Iris-virginica
```

The neural network successfully reduces the cost from 1 to below 0.1 and is correct in predicting each label of flower.

Extra dataset:
To test my neural network more I used a new dataset which contains types of wine along with different pieces of data about the wine. My goal was to predict the type of wine off of its features.

```
# Load the wine dataset (no header in the CSV)
df = pd.read_csv("wine_data.csv", header=None)

# Assign column names based on the description
df.columns = [
    "Class", "Alcohol", "Malic acid", "Ash", "Alcalinity of ash", "Magnesium",
    "Total phenols", "Flavanoids", "Nonflavanoid phenols", "Proanthocyanins",
    "Color intensity", "Hue", "OD280/OD315 of diluted wines", "Proline"
]
```

I loaded in the wine dataset and gave labels to the columns.
I originally copied my previous method of handling the dataset and added all of the input columns to the inputs variable and ignored scaling the data.
This was an issue because large values such as Proline ranges from 0 to 1000, but Hue ranges from 0 to 2, so the larger scaled features dominate the smaller ones leading to incorrect values.
Standardisation helps gradients flow evenly across the network, avoiding large updates for some weights and small updates for others.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
inputs = scaler.fit_transform(inputs.T).T
```

I imported Standard Scaler which transforms the data so each column has:
Mean = 0
Standard deviation = 1
I then transposed the array so that each feature of the wine is a column and each sample of wine is a row.
Fit calculates the mean and standard deviation for each column and transform scales each feature using this formula

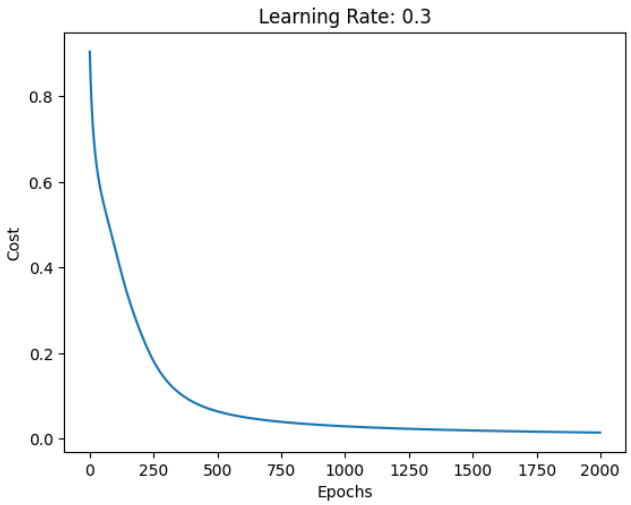
$$z = \frac{x_i - \mu}{\sigma}$$

Now that the values are standardised, my next issue was that the dataset was too large to print every prediction so I split the dataset into training sets.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(inputs.T, outputs.T, test_size=0.2, random_state=42)

# Transpose the split data to match the perceptron input format
X_train = X_train.T
X_test = X_test.T
y_train = y_train.T
y_test = y_test.T
```

train_test_split splits the dataset into two parts:
Training set: Used to train the model and represents the majority of the model
Testing Set: Used to evaluate the model's performance on unseen data and it represents the smaller portion of the data.
test_size = 0.2 allocates 20% to the testing set
random_state = 42 ensures that the random splitting process produces the same result every time
I ruin the program so there is consistency across runs
I then transpose the data back to restore the format needed by the perceptron

The rest of the code is similar to the above iris problem
I trained the data on 16 hidden neuron's, 2000 epochs and 0.3 eta
This is the result:




```
Predictions vs Actual Labels (Testing Subset):
Sample 1: Predicted - 1, Actual - 1
Sample 2: Predicted - 1, Actual - 1
Sample 3: Predicted - 3, Actual - 3
Sample 4: Predicted - 1, Actual - 1
Sample 5: Predicted - 2, Actual - 2
Sample 6: Predicted - 1, Actual - 1
Sample 7: Predicted - 2, Actual - 2
Sample 8: Predicted - 3, Actual - 3
Sample 9: Predicted - 2, Actual - 2
Sample 10: Predicted - 3, Actual - 3
Sample 11: Predicted - 1, Actual - 1
Sample 12: Predicted - 3, Actual - 3
Sample 13: Predicted - 1, Actual - 1
Sample 14: Predicted - 2, Actual - 2
Sample 15: Predicted - 1, Actual - 1
Sample 16: Predicted - 2, Actual - 2
Sample 17: Predicted - 2, Actual - 2
Sample 18: Predicted - 2, Actual - 2
Sample 19: Predicted - 1, Actual - 1
Sample 20: Predicted - 2, Actual - 2
Sample 21: Predicted - 1, Actual - 1
Sample 22: Predicted - 2, Actual - 2
Sample 23: Predicted - 2, Actual - 2
Sample 24: Predicted - 3, Actual - 3
Sample 25: Predicted - 3, Actual - 3
Sample 26: Predicted - 3, Actual - 3
Sample 27: Predicted - 2, Actual - 2
```

All of the Predicted values matched the actual values for the class of wine.
What I learnt from using g a bigger dataset with highly varying values.
Scaling data is very important to prevent larger-valued columns from dominating smaller ones which ensures weights are appropriate.
Splitting the information into training and testing sets allows one to truthfully evaluate a model's generalisation by testing the model on unseen dat. This is a step to make the model reliable for new inputs on the dataset.

I tested the cross-entropy-cost function on this dataset but saw little to no difference in the result.

I believe this is because the Wine dataset is a well-posed classification problem with well-separated classes. Both MSE and cross-entropy perform well on this dataset because the separation of the classes reduces the impact of the differences between the cost functions. Because the classes are already easy to distinguish, the benefits of cross-entropy, which are faster convergence and better handling of probabilistic outputs might not be evident.

The reason the cross-entropy function worked so well with problem 1 and 2 was because cross-entropy is aligned with classification tasks. It penalises incorrect prediction score strongly.

MSE assumes as regression-style task, minimising the squared difference between outputs. This works well for continuous outputs but is suboptimal for probabilities or discrete class labels.