

ASSIGNMENT 2 DECISION TREE & NEURAL NETWORK ANALYSIS

Lauchlan Thomson

s5315701

26th April 2024

2802ICT

Intelligent Systems

Table of Contents

1. Task 1 Introduction	4
2. Software Design	4
2.1. Data Preparation and Functions	4
2.1.1. Split Train-Test Data Function	4
2.1.2. Accuracy Calculation Function	5
2.1.3. Confusion Matrix Function	5
2.1.4. Learning Curve Plotting Function	6
3. Decision Tree Implementation	6
3.1. Node Class	6
3.2. DecisionTree Class	7
4. Results and Evaluation	8
4.1. Training and Testing Set Sizes	8
4.2. Accuracy	8
4.3. Confusion Matrix	8
4.4. Precision, Recall, and F1-Score	8
4.5. Definitions	9
4.6. Learning Curve	10
5. Discission	11
5.1. Precision, Recall, and F1-Score Analysis	11
5.2. Learning Curve Insights	11
5.3. Challenges and Limitations	12
5.4. Optimisations	12
6. Conclusion	12
1. Task 2 Introduction	13
2. Software Design	13
2.1. Data Preparation and Functions	13

2.1.1.	Loading and Normalizing Data	13
2.1.2.	Split Train-Test Data Function	14
2.1.3.	Learning Curve Plotting Function	14
2.2.	Neural Network Implementation	14
2.2.1.	Neural Network Class	14
2.2.2.	Training Function	15
2.2.3.	Mini-Batch Update Function.....	15
2.2.4.	Backpropagation Function	16
2.2.5.	Evaluation Function	17
3.	Experimental Setup.....	18
3.1.	Dataset Description	18
3.2.	Hyperparameter Experiments	18
3.3.	Experiment 1: Default Parameters.....	18
3.4.	Experiment 2: Modified Learning Rate	18
3.5.	Experiment 3: Modified Batch Sizes	19
3.6.	Experiment 4: Maximum Accuracy.....	19
4.	Results and Evaluations.....	19
4.1.	Training and Testing Set Sizes.....	19
4.2.	Accuracy.....	19
4.2.1.	Experiment 1	19
4.2.2.	Experiment 2	20
4.2.3.	Experiment 3	21
4.2.4.	Experiment 4	21
4.3.	Learning Rate Experiments	22
4.4.	Mini-Batch Size Experiments.....	22
4.5.	Optimal Hyperparameter Settings	23
5.	Discussion.....	23
5.1.	Learning Rate Analysis.....	23
5.2.	Mini-Batch Size Insights	24
5.3.	Challenges and Limitations	25
6.	Conclusion.....	25

1. Task 1 | Introduction

A decision tree is a supervised learning algorithm used for both classification and regression tasks. It creates a model that predicts the value of a target variable based on several input variables. The model is represented in the form of a tree structure. This report outlines the implementation and evaluation of a decision tree, focusing on the software design, results, and performance analysis.

2. Software Design

2.1. Data Preparation and Functions

The implementation of the decision tree involves several functions and data structures, which are described below:

2.1.1. Split Train-Test Data Function

Function: *split_data*

- **Description:** This function splits the dataset into training and testing sets based on a specified training percentage.
- **Data Structures:** The function uses numpy arrays and **pandas** DataFrames for data manipulation.

```
def split_data(x, y, training_percent):  
    indices = np.random.permutation(len(x))  
    training_size = int(len(x) * training_percent)  
    training_indices = indices[:training_size]  
    testing_indices = indices[training_size:]  
    return x[training_indices], x[testing_indices], y[training_indices], y[testing_indices]
```

Figure 1: Split Data Function

2.1.2. Accuracy Calculation Function

Function: *accuracy*

- **Description:** This function calculates the accuracy of the classifier by comparing the predicted labels with the actual labels.
- **Data Structures:** It uses **numpy** arrays.

```
def accuracy(y_test, y_pred):
    if len(y_test) == 0:
        return
    return np.sum(y_test == y_pred) / len(y_test)
```

Figure 2: Accuracy Function

2.1.3. Confusion Matrix Function

Function: *confusion_matrix*

- **Description:** This function creates a confusion matrix to evaluate the performance of the classifier.
- **Data Structures:** The function uses **numpy** arrays and **pandas** DataFrames for creating and visualizing the confusion matrix.

```
def confusion_matrix(actual, predictions):
    titles = ['unacc', 'acc', 'good', 'vgood']
    classes = np.unique(actual)
    matrix = np.zeros((len(classes), len(classes)))
    for i in range(len(classes)):
        for j in range(len(classes)):
            matrix[i, j] = np.sum((actual == classes[i]) & (predictions == classes[j]))
    # Create confusion matrix table with better visualisation
    matrix = pd.DataFrame(matrix, index=titles, columns=titles)

    # Calculate my F1-Score, Precision and Recall
    tp = np.diag(matrix)
    fp = np.sum(matrix, axis=0) - tp
    fn = np.sum(matrix, axis=1) - tp
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1_score = 2 * (precision * recall) / (precision + recall)

    # Calculate weighted and macro averages
    weighted_precision = np.sum(precision * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
    weighted_recall = np.sum(recall * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
    weighted_f1 = np.sum(f1_score * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
    macro_precision = np.mean(precision)
    macro_recall = np.mean(recall)
    macro_f1_score = np.mean(f1_score)

    # Store averages in arrays to put in dataframe
    weighted_avgs = [weighted_f1, weighted_precision, weighted_recall]
    macro_avgs = [macro_f1_score, macro_precision, macro_recall]

    # Create DataFrame table of metrics
    metrics = pd.DataFrame({'Precision': precision, 'Recall': recall, 'F1-Score': f1_score}, index=titles)
    metrics = metrics.T
    metrics['Weighted Avg'] = weighted_avgs
    metrics['Macro Avg'] = macro_avgs
    metrics.insert(4, '-', '-')
    metrics = metrics.round(3)
    metrics = metrics.T

    return matrix, metrics
```

Figure 3: Confusion Matrix Function

2.1.4. Learning Curve Plotting Function

Function: *plot_learning_curve*

- **Description:** This function plots the learning curve of the decision tree, showing how accuracy changes with the percentage of training data used.
- **Data Structures:** Uses **matplotlib** for plotting the learning curve.

```
def plot_learning_curve(x_train, y_train, x_test, y_test):
    training_sizes = np.linspace(0.0625, 1.0, 16)
    accuracies = []
    for size in training_sizes:
        x_train, x_test, y_train, y_test = split_data(x, y, size)
        tree.fit(x_train, y_train)
        predictions = tree.predict(x_test)
        accuracies.append(accuracy(y_test, predictions))
    plt.plot(training_sizes, accuracies, '-o')
    plt.grid()
    plt.xlabel('Training Data Used (%)')
    plt.ylabel('Accuracy (%)')
    plt.title('Learning Curve for Decision Tree Classifier')
    plt.yticks(np.linspace(0.8, 1, 12), ['80%', '82%', '84%', '86%',
                                         '88%', '90%', '92%', '94%', '96%', '98%', '100%', '102%'])
    plt.xticks(rotation=45) # Rotate x labels by 45 degrees
    plt.xticks(training_sizes, ['6.25%', '12.5%', '18.75%', '25%',
                               '31.25%', '37.5%', '43.75%', '50%',
                               '56.25%', '62.5%', '68.75%', '75%',
                               '81.25%', '87.5%', '93.75%', '100%'])
    plt.show()
```

Figure 4: Learning Curve Plot Function

3. Decision Tree Implementation

3.1. Node Class

The **Node** class represents each node in the decision tree. It contains attributes for the feature, threshold, left and right children, and the value for leaf nodes.

```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None
```

Figure 5: Node Class

3.2. DecisionTree Class

The DecisionTree class contains methods for fitting the model, finding the best splits, calculating information gain, and making predictions.

```
class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100, n_features=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_features = n_features
        self.root = None

    def fit(self, X, y):
        self.n_features = X.shape[1] if not self.n_features else min(self.n_features, X.shape[1])
        self.root = self.tree(X, y)

    def tree(self, X, y, depth=0):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        if depth >= self.max_depth or n_labels == 1 or n_samples < self.min_samples_split:
            leaf_value = self.labal(y)
            return Node(value=leaf_value)

        feat_idx = np.random.choice(n_features, self.n_features, replace=False)

        best_feature, best_thresh = self.best_split(X, y, feat_idx)

        left_idx, right_idx = self.split(X[:, best_feature], best_thresh)
        left = self.tree(X[left_idx, :], y[left_idx], depth + 1)
        right = self.tree(X[right_idx, :], y[right_idx], depth + 1)
        return Node(best_feature, best_thresh, left, right)

    def best_split(self, X, y, feat_idx):
        best_gain = -1
        split_idx, split_thresh = None, None
        for feat_idx in feat_idx:
            X_column = X[:, feat_idx]
            thresholds = np.unique(X_column)

            for threshold in thresholds:
                gain = self.information_gain(y, X_column, threshold)

                if gain > best_gain:
                    best_gain = gain
                    split_idx = feat_idx
                    split_thresh = threshold

        return split_idx, split_thresh

    def information_gain(self, y, X_column, split_thresh):
        parent_entropy = self.entropy(y)
        left_idx, right_idx = self.split(X_column, split_thresh)

        if len(left_idx) == 0 or len(right_idx) == 0:
            return 0

        n = len(y)
        n_l, n_r = len(left_idx), len(right_idx)
        e_l, e_r = self.entropy(y[left_idx]), self.entropy(y[right_idx])
        child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

        ig = parent_entropy - child_entropy
        return ig

    def split(self, X_column, split_thresh):
        left_idx = np.argwhere(X_column <= split_thresh).flatten()
        right_idx = np.argwhere(X_column > split_thresh).flatten()
        return left_idx, right_idx

    def entropy(self, y):
        hist = np.bincount(y)
        ps = hist / len(y)
        return -np.sum([p * np.log(p) for p in ps if p > 0], axis=0)

    def labal(self, y):
        counter = Counter(y)
        value = counter.most_common(1)[0][0]
        return value

    def search_tree(self, x, node):
        if node.is_leaf_node():
            return node.value

        if x[node.feature] <= node.threshold:
            return self.search_tree(x, node.left)
        return self.search_tree(x, node.right)

    def predict(self, X):
        return np.array([self.search_tree(x, self.root) for x in X])
```

Figure 6: Decision Tree Class

4. Results and Evaluation

4.1. Training and Testing Set Sizes

- Training Size: 1382 samples
- Testing Size: 346 samples

4.2. Accuracy

The classifier achieved an overall accuracy of 97.688% on the testing set. This high accuracy shows that the model has effectively learned to predict the data.

4.3. Confusion Matrix

The confusion matrix provides a detailed breakdown of the algorithm's performance across different classes:

	unacc	acc	good	vgood
unacc	231	3	0	0
acc	1	85	2	1
good	0	0	11	0
vgood	0	0	1	11

Table 1: Confusion Matrix

4.4. Precision, Recall, and F1-Score

These metrics provide a better representation of the model's performance:

- **Unacc:** Precision: 0.996, Recall: 0.987, F1-Score: 0.991
- **Acc:** Precision: 0.966, Recall: 0.955, F1-Score: 0.96
- **Good:** Precision: 0.786, Recall: 1.0, F1-Score: 0.88
- **Vgood:** Precision: 0.917, Recall: 0.917, F1-Score: 0.917
- **Weight Avg:** Precision: 0.955, Recall: 0.956, F1-Score: 0.955
- **Macro Avg:** Precision: 0.937, Recall: 0.916, F1-Score: 0.965

4.5. Definitions

- **Precision:** Precision is the ratio of correctly predicted positive observations to the total predicted positives. It indicates how accurate the program is in making correct predictions. High precision relates to the low false positive rate.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Where:

- **TP** is the number of True Positives
 - **FP** is the number of False Positives
-
- **Recall:** Recall is the ratio of correctly predicted positive results to all the results in the actual class. It is a measure of the program's completeness. High recall relates to the low false negative rate.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Where:

- **TP** is the number of True Positives
 - **FN** is the number of False Negatives
-
- **F1-Score:** The F1-Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. It is especially helpful when some classes have more examples than others.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.6. Learning Curve

The learning curve indicates how the program's accuracy improves as the size of the training set increases. The plot shows a significant initial improvement, followed by a stabilization as more data is used:

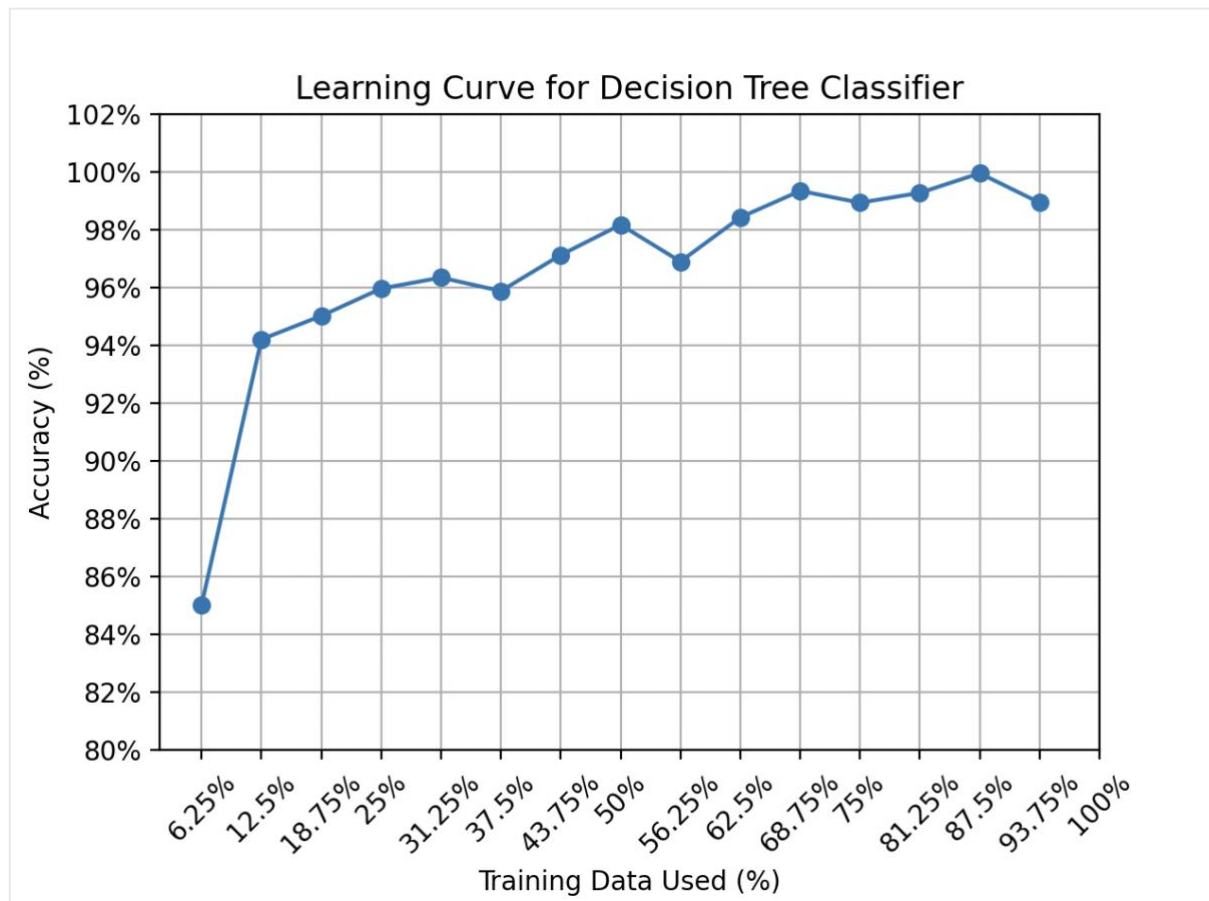


Figure 7: Learning Curve Plot

- **Initial Improvement:** The curve shows a steep increase in accuracy from approximately 86% to 95% as the training data increases from 6.25% to 12.5%. This indicates that even a small amount of training data can significantly improve the model's performance.
- **Steady Improvement:** From 12.5% to 50% of the training data, the accuracy steadily improves, stabilizing around 96%. This suggests that adding more data continues to enhance the model's ability to predict.
- **Stable:** Beyond 50% of the training data, the accuracy stabilizes around 98%, indicating that the model has learned most of the patterns in the data. Additional data provides less returns in terms of accuracy improvement.

- **Minor Fluctuations:** Small fluctuations are observed in the accuracy, particularly as the training data approaches 100%. This can be explained by overfitting, where the model may perform exceptionally well on the training data but not as consistently on unseen data.

5. Discission

The decision tree demonstrated strong performance on the given dataset, achieving an overall accuracy of 97.688%. The confusion matrix shows that the classifier performs exceptionally well in predicting the "unacc" and "acc" classes, with minor mistakes in the "good" and "vgood" classes.

5.1. Precision, Recall, and F1-Score Analysis

- The high precision and recall for the "unacc" class indicate that the program is highly effective at identifying this class with minimal false positives and false negatives.
- The "acc" class also shows high precision and recall, though slightly lower than the "unacc" class, indicating a few imperfections.
- The "good" class has perfect recall but lower precision, suggesting that while it correctly identifies all instances of the "good" class, it also misclassifies some instances of other classes as "good."
- The "vgood" class has balanced precision and recall, reflecting consistent performance but with some room for improvement.

5.2. Learning Curve Insights

The learning curve demonstrates a steep initial increase in accuracy as more training data is added, with accuracy improving from approximately 86% to 95% with just 12.5% of the training data. The curve stabilizes around 98% accuracy as more data is used, indicating that the classifier benefits from additional data but reaches a point of non valuable returns. Minor fluctuations in accuracy, particularly at higher percentages of training data, suggest potential overfitting.

5.3. Challenges and Limitations

- The classifier's performance on the "good" and "vgood" classes is lower than on the "unacc" and "acc" classes, suggesting potential room for improvement in distinguishing between these classes.
- Overfitting is a potential issue, as the classifier's performance on the training set may not generalize as well to unseen data. Techniques like pruning could be explored to fix this.

5.4. Optimisations

The **predict** method uses vectorized operations to make predictions for the entire dataset. This approach significantly speeds up the prediction phase, especially when dealing with large datasets, by avoiding the need to iterate through each sample individually.

6. Conclusion

The decision tree is an effective and efficient algorithm for classification tasks, demonstrating high accuracy and robust performance on the given dataset. It efficiently handles both balanced and unbalanced classes, making it a resourceful tool for classification tasks. Future improvements could focus on optimizing the tree-building process and implementing techniques to handle overfitting, such as pruning. Overall, the decision tree is a valuable tool in the machine learning toolkit.

1. Task 2 | Introduction

This report outlines the implementation and evaluation of a neural network for classifying fashion items using the Fashion-MNIST dataset. The neural network is designed with three layers: an input layer, a hidden layer, and an output layer. The primary objective is to train the network on the training dataset and evaluate its performance on the test dataset, experimenting with various hyperparameter settings to optimize accuracy.

2. Software Design

2.1. Data Preparation and Functions

2.1.1. Loading and Normalizing Data

Function: *load_data*

- **Description:** The data is loaded from compressed CSV files using `numpy.loadtxt` without the need to decompress the files manually. The pixel values are normalized to a range of 0 to 1 by dividing by 255.
- **Data Structures:** The function uses numpy arrays for data manipulation.

```
def load_data(train, test):  
    # Load and prepare the training data  
    train_set = np.loadtxt(train, skiprows=1, delimiter=',')  
    train_data = train_set[:, 1:]  
    train_labels = train_set[:, 0]  
    training_data = list(zip(train_data, train_labels))  
  
    # Load and prepare the testing data  
    test_set = np.loadtxt(test, skiprows=1, delimiter=',')  
    test_data = test_set[:, 1:]  
    test_labels = test_set[:, 0]  
    testing_data = list(zip(test_data, test_labels))  
  
    return training_data, testing_data
```

Figure 8: Load Data Function

2.1.2. Split Train-Test Data Function

Function: *N/A*

- **Description:** This function would be implied to split the dataset into training and testing sets based on a specified percentage, but it is managed within the *load_data* function in this case.
- **Data Structures:** Numpy arrays and lists.

2.1.3. Learning Curve Plotting Function

Function: *N/A*

- **Description:** This function would plot the learning curve showing the accuracy of the neural network over epochs. This is done in conjunction with running the Neural Network in the *main()* function.
- **Data Structures:** Uses matplotlib for plotting.

2.2. Neural Network Implementation

2.2.1. Neural Network Class

Class: *NeuralNetwork*

Description: The *NeuralNetwork* class contains all components of the neural network, including initialization, training, and evaluation.

Data Structures: Lists, numpy arrays.

```
class NeuralNetwork:
    def __init__(self):
        self.weights = None
        self.biases = None

    def init_parameters(self, input_neurons, hidden_neurons, output_neurons, epochs, lr, batch_size):
        # Initialize the neural network parameters
        self.input_neurons = input_neurons
        self.hidden_neurons = hidden_neurons
        self.output_neurons = output_neurons
        self.epochs = epochs
        self.learning_rate = lr
        self.batch_size = batch_size

        # Initialize weights and biases with random values
        self.biases = [np.random.randn(hidden_neurons, 1), np.random.randn(output_neurons, 1)]
        self.weights = [np.random.randn(hidden_neurons, input_neurons), np.random.randn(output_neurons, hidden_neurons)]
```

Figure 9: Neural Network Init Class

2.2.2. Training Function

Function: *train*

Description: The train method handles the training process over a specified number of epochs, updating the weights and biases using mini-batch gradient descent.

Data Structures: Lists for storing mini-batches and accuracies, numpy arrays for data manipulation.

```
def train(self, train_data, test_data):
    # Train the neural network using mini-batch gradient descent
    test_accuracies = [] # Define the variable "test_accuracies"
    for epoch in range(1, self.epochs + 1):
        start_time = time.time()
        random.shuffle(train_data) # Shuffle the training data

        # Create mini-batches from the training data
        mini_batches = [train_data[i:i + self.batch_size] for i in range(0, len(train_data), self.batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch)

        # Evaluate the model on the test data and print the accuracy
        accuracy = self.evaluate(test_data)
        end_time = time.time()
        test_accuracies.append(accuracy)
        print(f"Epoch #{epoch} \nAccuracy = {accuracy / 100}% \nTime Elapsed {round(end_time - start_time, 2)}s\n")
    print("Training complete")
```

Figure 10: Train Class Function

2.2.3. Mini-Batch Update Function

Function: *update_mini_batch*

Description: The update_mini_batch function updates the weights and biases for each mini-batch using the backpropagation algorithm. This is where I update my weights and biases using the equation given in the assignment document:

$$w \rightarrow w - \frac{\eta}{m} \sum_{i=1}^m error_i^w$$

Where:

- ***m*** is *len(mini_batch)*, the number of samples in a mini batch
- ***error_i^w*** is *wd* for *w*, *wd* in *zip(self.weights, weight_deltas)* which is the error of weight ***w*** for input ***i***
- ***η*** is *self.learning_rate* the learning rate of the neural network

Data Structures: Numpy arrays for storing weight and bias updates.


```
def update_mini_batch(self, mini_batch):
    # Initialize the changes in biases and weights to zero
    bias_deltas = [np.zeros(b.shape) for b in self.biases]
    weight_deltas = [np.zeros(w.shape) for w in self.weights]

    for x, y in mini_batch:
        # Compute the gradient for the current mini-batch
        delta_b, delta_w = self.backpropagate(x, y)
        bias_deltas = [bd + d for bd, d in zip(bias_deltas, delta_b)]
        weight_deltas = [wd + d for wd, d in zip(weight_deltas, delta_w)]

    # Update the weights and biases using the gradients
    self.weights = [w - (self.learning_rate / len(mini_batch)) * wd for w, wd in zip(self.weights, weight_deltas)]
    self.biases = [b - (self.learning_rate / len(mini_batch)) * bd for b, bd in zip(self.biases, bias_deltas)]
```

Figure 11: Update Mini Batch Class Function

2.2.4. Backpropagation Function

Function: *backpropagate*

Description: The backpropagate function computes the gradients for the weights and biases based on the error between the predicted and actual outputs. This function is where I use the quadratic error function to calculate the error for all my weights and biases of the neurons using the error function:

$$C(w, b) = \frac{1}{2n} \sum_{i=1}^n |f(x_i) - y_i|^2$$

Where:

- w = weights
- b = biases
- n = number of test instances
- x_i = i^{th} test instance vector
- y_i = i^{th} test label vector
- $f(x)$ = label predicted by the network for an input x

Data Structures: Numpy arrays for storing activations, gradients, and errors.

```
def backpropagate(self, x, y):  
    # Perform the backpropagation algorithm to compute the gradients  
    x = np.array(x).reshape((len(x), 1)) / 255.0 # Normalize input  
    y = int(y)  
  
    # Feedforward pass  
    z1 = np.dot(self.weights[0], x) + self.biases[0]  
    a1 = self.sigmoid(z1)  
    z2 = np.dot(self.weights[1], a1) + self.biases[1]  
    a2 = self.sigmoid(z2)  
  
    # Compute the output error  
    y_vector = np.zeros((self.output_neurons, 1))  
    y_vector[y] = 1.0  
    output_error = (a2 - y_vector) * self.sigmoid_derivative(a2)  
  
    # Compute the hidden layer error  
    hidden_error = np.dot(self.weights[1].T, output_error) * self.sigmoid_derivative(a1)  
  
    # Compute the gradient for biases and weights  
    delta_biases = [hidden_error, output_error]  
    delta_weights = [np.dot(hidden_error, x.T), np.dot(output_error, a1.T)]  
  
    return delta_biases, delta_weights
```

Figure 12: Backpropagation Class Function

2.2.5. Evaluation Function

Function: *evaluate*

Description: The evaluate function calculates the accuracy of the neural network on the test data.

Data Structures: Numpy arrays for storing test data and predictions.

```
def evaluate(self, test_data):  
    # Evaluate the model on the test data  
    correct = 0  
    for x, y in test_data:  
        if self.predict(x) == y:  
            correct += 1  
    return correct
```

Figure 13: Evaluate Class Function

3. Experimental Setup

3.1. Dataset Description

The Fashion-MNIST dataset consists of 60,000 training images and 10,000 test images. Each image is 28x28 pixels, grayscale, and labelled into one of ten categories: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.

3.2. Hyperparameter Experiments

The neural network was initially tested with the default hyperparameters 784, 30, 10, 30, 3, 20, resulting in 86.48% accuracy, which served as a base case while attempting to tweak the hyperparameters to influence the highest possible accuracy. I conducted 4 experiments as my approach to achieving the maximum accuracy within my hyperparameters.

3.3. Experiment 1: Default Parameters

- **Input neurons:** 784
- **Hidden neurons:** 30
- **Output neurons:** 10
- **Epochs:** 30
- **Learning rate:** 3
- **Batch size:** 20

3.4. Experiment 2: Modified Learning Rate

- **Input neurons:** 784
- **Hidden neurons:** 30
- **Output neurons:** 10
- **Epochs:** 30
- **Learning rate:** [0.001, 0.01, 1, 10 ,100]
- **Batch size:** 20

3.5. Experiment 3: Modified Batch Sizes

- **Input neurons:** 784
- **Hidden neurons:** 30
- **Output neurons:** 10
- **Epochs:** 30
- **Learning rate:** 3
- **Batch size:** [1, 5, 20, 100, 300]

3.6. Experiment 4: Maximum Accuracy

- **Input neurons:** 784
- **Hidden neurons:** 30
- **Output neurons:** 10
- **Epochs:** 30
- **Learning rate:** 7
- **Batch size:** 40

4. Results and Evaluations

4.1. Training and Testing Set Sizes

Training Set: 60,000 samples

Testing Set: 10,000 samples

4.2. Accuracy

4.2.1. Experiment 1

The initial accuracy with default hyperparameters reached 86.48% on the test set.

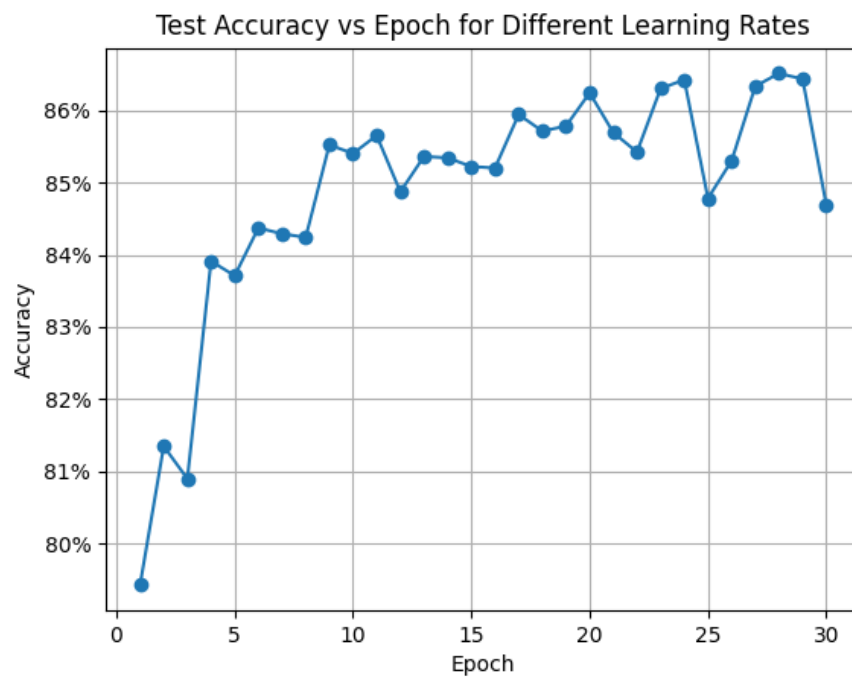


Figure 14: Default Hyper Parameter Accuracy vs Epoch Plot

4.2.2. Experiment 2

This experiment consisted of testing 5 different learning rates on the default hyper parameters hypothesising what effect learning rate has on accuracy in this neural network.

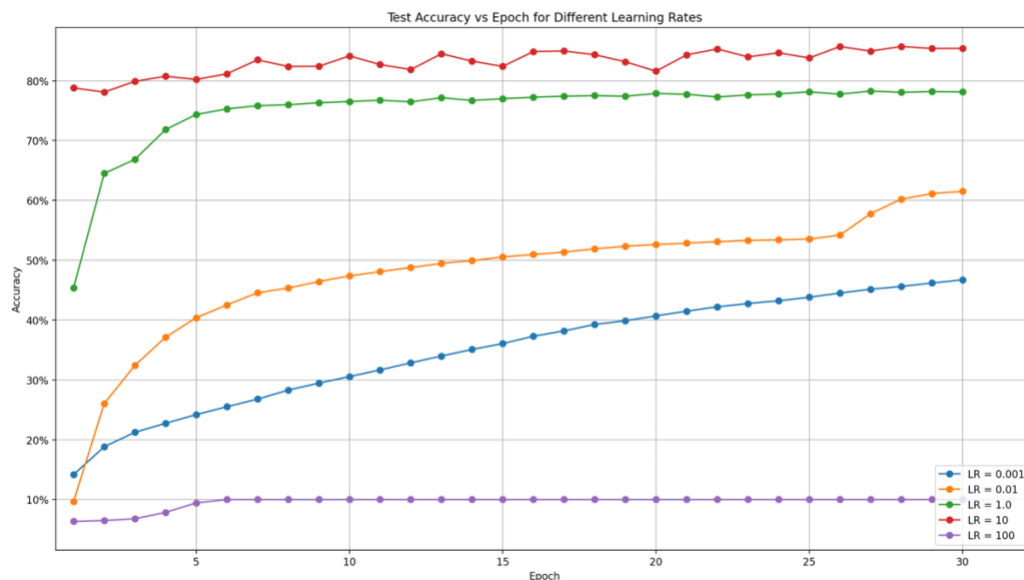


Figure 15: Alternate Learning Rates Accuracy vs Epoch Plot

4.2.3. Experiment 3

Alternatively, this experiment trialled 5 separate batch sizes showing any insight into the effect batch size has on accuracy.

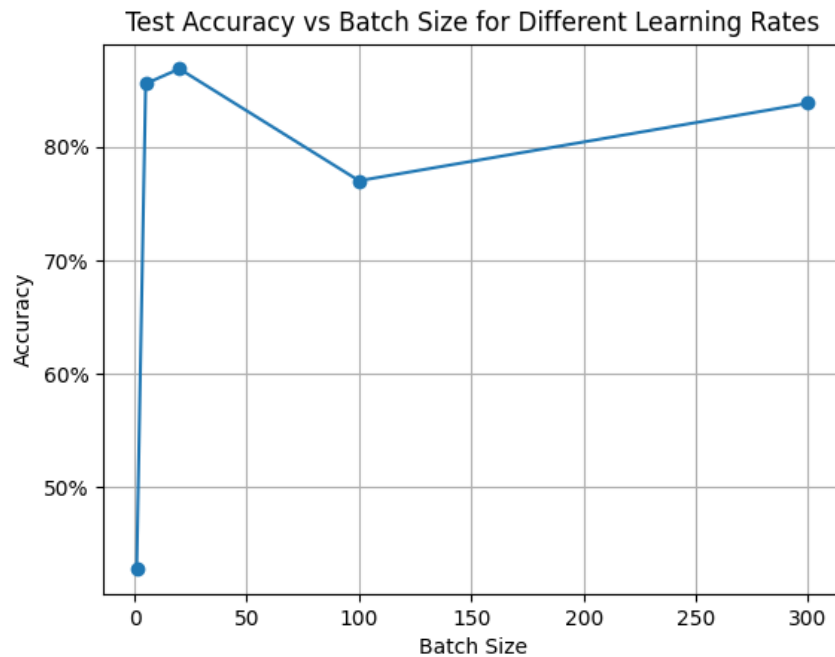


Figure 16: Alternate Batch Size Accuracy vs Epoch Plot

4.2.4. Experiment 4

Finally, this experiment demonstrates the most optimal hyperparameters to result in the highest accuracy given the results of the previous experiments.

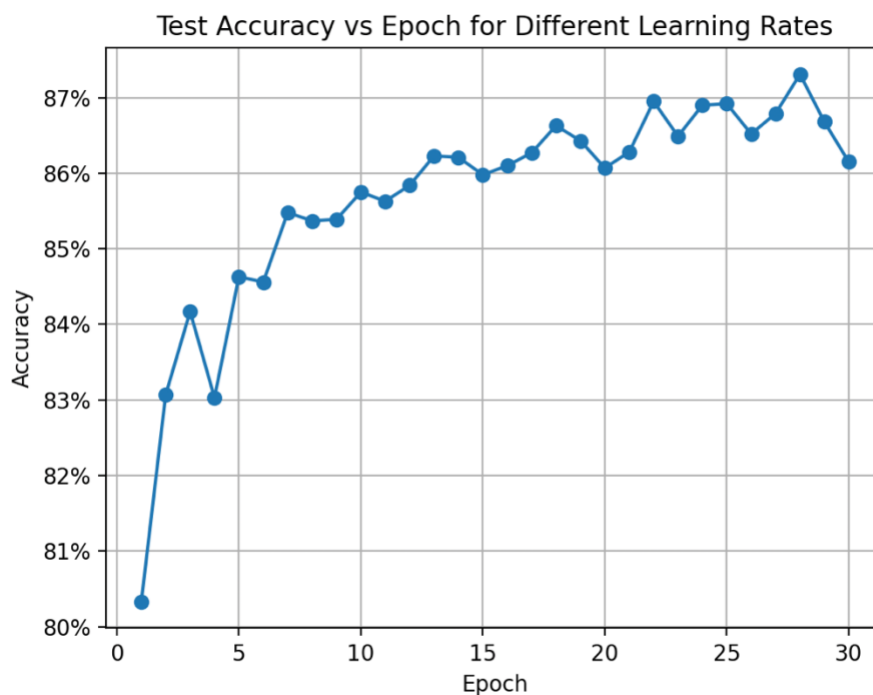
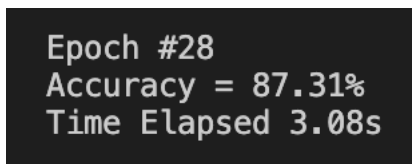


Figure 17: Best Result Accuracy vs Epoch Plot



```
Epoch #28
Accuracy = 87.31%
Time Elapsed 3.08s
```

Figure 18: Best Result Output

4.3. Learning Rate Experiments

Experiments were conducted with learning rates of 0.001, 0.01, 1.0, 10, and 100. The results are summarized in the following table:

Learning Rate	Maximum Accuracy (%)
0.001	57.21
0.01	61.13
1.0	78.6
10	86.74
1	10.00

Table 2: Alternate Learning Rate Result Table

4.4. Mini-Batch Size Experiments

Experiments were conducted with mini-batch sizes of 1, 5, 10, 20, 100, and 300. The results are summarized in the following table:

Mini-Batch Size	Maximum Accuracy (%)
1	42.64
5	85.97
20	86.55
100	77.96
300	83.44

Table 3: Alternate Batch Size Result Table

4.5. Optimal Hyperparameter Settings

After experimenting with various hyperparameters, the optimal settings were found to be:

- Learning rate: 7
- Mini-batch size: 40
- Epochs: 30

Results:

Maximum accuracy achieved: 87.31% on the test set.

5. Discussion

5.1. Learning Rate Analysis

The learning rate significantly affects the speed of learning and final accuracy of the neural network. A learning rate of 10.0 provided the best balance between learning speed and accuracy, achieving a maximum accuracy of 86.74%. Lower learning rates such as 0.001 and 0.01 resulted in slower confident predictions, while the highest learning rate (100) caused instability and drastically lower accuracy.

Detailed Explanation:

- **Low Learning Rates (0.001 and 0.01):** These resulted in slower learning and lower accuracy because the small step sizes in each iteration meant that the network took a long time to come to a minimum. This slow adoption can lead to underfitting, where the model fails to capture the basic trends in the data.
- **Moderate Learning Rate (1.0):** This rate achieved a better balance, allowing the network to adapt more quickly and accurately. The steps were large enough to make significant progress in each iteration without overshooting the minimum.
- **High Learning Rates (10.0 and 100):** While a learning rate of 10.0 was optimal, the highest rate (100) caused the network to become unstable, often overshooting the optimal weights and failing to learn. This instability leads to high variance in the accuracy and can cause the model to diverge rather than converge, resulting in poor performance.

5.2. Mini-Batch Size Insights

Smaller mini-batch sizes (e.g., 1, 5) introduced more noise into the gradient estimates, helping the network escape local minima but increasing computational time. A mini-batch size of 20 provided the best balance, achieving a maximum accuracy of 86.55%. Larger mini-batch sizes (e.g., 100, 300) resulted in lower accuracy due to reduced noise in gradient estimates.

Detailed Explanation:

- **Small Mini-Batch Sizes (1 and 5):** These sizes introduced a high variance in the gradient estimates, which can help in escaping local minima and finding better general solutions. However, this also increases the computational cost as more updates are needed.
- **Optimal Mini-Batch Size (20):** This size provided a good balance between the variance in gradient estimates and computational efficiency, resulting in the highest accuracy.
- **Large Mini-Batch Sizes (100 and 300):** These sizes reduced the variance in the gradient estimates, leading to more stable but less frequent updates. This stability can sometimes cause the neural network to get stuck in local minima and fail to generalize well, resulting in lower accuracy.

5.3. Challenges and Limitations

The primary challenges faced during the experiments included overfitting and underfitting. Overfitting was found in experiments with high learning rates and larger numbers of epochs, while underfitting occurred with lower learning rates and insufficient training epochs. Additionally, the computational time increased significantly with smaller mini-batch sizes.

Detailed Explanation:

- **Overfitting:** This occurs when the model learns the noise in the training data rather than the actual pattern, resulting in high accuracy on the training set but poor generalization to the test set. This was observed with high learning rates and large numbers of epochs, where the network could memorize the training data but failed to perform well on unseen data.
- **Underfitting:** This happens when the model is too simple to capture the underlying patterns in the data, resulting in poor performance on both the training and test sets. Lower learning rates and insufficient training epochs led to underfitting, where the model could not learn the necessary features from the data.
- **Computational Time:** Smaller mini-batch sizes increased the number of updates required, leading to longer training times. While these smaller batches can help in finding better solutions, they also significantly increase the computational cost, making the training process more time-consuming.

6. Conclusion

The neural network implemented for classifying fashion items from the Fashion-MNIST dataset achieved a maximum accuracy of 87.31% with optimal hyperparameters. The experiments demonstrated the importance of tuning hyperparameters such as learning rate and mini-batch size to balance convergence speed, accuracy, and computational efficiency. Future improvements could involve exploring different network architectures and regularization techniques to further enhance performance.

References

Both of these videos are youtube videos used to get an understanding of how a neural network works fundamentally. They cover from start to finish how the layers, weight, biases, activation functions and batches work.

<https://www.youtube.com/playlist?list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3>

<https://www.youtube.com/watch?v=w8yWXqWQYmU&t=1205s>

Appendix 1: Decision Tree Source Code

```
import numpy as np
from collections import Counter
import pandas as pd
import matplotlib.pyplot as plt

class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100, n_features=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_features = n_features
        self.root = None

    def fit(self, X, y):
        self.n_features = X.shape[1] if not self.n_features else min(self.n_features, X.shape[1])
        self.root = self.tree(X, y)

    def tree(self, X, y, depth=0):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        if depth >= self.max_depth or n_labels == 1 or n_samples < self.min_samples_split:
            leaf_value = self.labal(y)
            return Node(value=leaf_value)
```

```
feat_idx = np.random.choice(n_features, self.n_features, replace=False)

best_feature, best_thresh = self.best_split(X, y, feat_idx)

left_idx, right_idx = self.split(X[:, best_feature], best_thresh)
left = self.tree(X[left_idx, :], y[left_idx], depth + 1)
right = self.tree(X[right_idx, :], y[right_idx], depth + 1)
return Node(best_feature, best_thresh, left, right)

def best_split(self, X, y, feat_idx):
    best_gain = -1
    split_idx, split_thresh = None, None
    for feat_idx in feat_idx:
        X_column = X[:, feat_idx]
        thresholds = np.unique(X_column)

        for threshold in thresholds:
            gain = self.information_gain(y, X_column, threshold)

            if gain > best_gain:
                best_gain = gain
                split_idx = feat_idx
                split_thresh = threshold
    return split_idx, split_thresh

def information_gain(self, y, X_column, split_thresh):
    parent_entropy = self.entropy(y)
    left_idx, right_idx = self.split(X_column, split_thresh)

    if len(left_idx) == 0 or len(right_idx) == 0:
        return 0

    n = len(y)
    n_l, n_r = len(left_idx), len(right_idx)
    e_l, e_r = self.entropy(y[left_idx]), self.entropy(y[right_idx])
    child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

    ig = parent_entropy - child_entropy
    return ig
```

```
def split(self, X_column, split_thresh):
    left_idx = np.argwhere(X_column <= split_thresh).flatten()
    right_idx = np.argwhere(X_column > split_thresh).flatten()
    return left_idx, right_idx

def entropy(self, y):
    hist = np.bincount(y)
    ps = hist / len(y)
    return -np.sum([p * np.log(p) for p in ps if p > 0], axis=0)

def labal(self, y):
    counter = Counter(y)
    value = counter.most_common(1)[0][0]
    return value

def search_tree(self, x, node):
    if node.is_leaf_node():
        return node.value

    if x[node.feature] <= node.threshold:
        return self.search_tree(x, node.left)
    return self.search_tree(x, node.right)

def predict(self, X):
    return np.array([self.search_tree(x, self.root) for x in X])

def split_data(x, y, training_percent):
    indices = np.random.permutation(len(x))
    training_size = int(len(x) * training_percent)
    training_indices = indices[:training_size]
    testing_indices = indices[training_size:]
    return x[training_indices], x[testing_indices], y[training_indices], y[testing_indices]

def accuracy(y_test, y_pred):
    if len(y_test) == 0:
        return
    return np.sum(y_test == y_pred) / len(y_test)

def confusion_matrix(actual, predictions):
```

```
titles = ['unacc', 'acc', 'good', 'vgood']
classes = np.unique(actual)
matrix = np.zeros((len(classes), len(classes)))
for i in range(len(classes)):
    for j in range(len(classes)):
        matrix[i, j] = np.sum((actual == classes[i] & (predictions == classes[j])))
# Create confusion matrix table with better visualisation
matrix = pd.DataFrame(matrix, index=titles, columns=titles)

# Calculate my F1-Score, Precision and Recall
tp = np.diag(matrix)
fp = np.sum(matrix, axis=0) - tp
fn = np.sum(matrix, axis=1) - tp
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1_score = 2 * (precision * recall) / (precision + recall)

# Calculate weighted and macro averages
weighted_precision = np.sum(precision * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
weighted_recall = np.sum(recall * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
weighted_f1 = np.sum(f1_score * [i for i in np.diag(matrix)] / np.sum(np.asarray(matrix)))
macro_precision = np.mean(precision)
macro_recall = np.mean(recall)
macro_f1_score = np.mean(f1_score)

# Store averages in arrays to put in dataframe
weighted_avgs = [weighted_f1, weighted_precision, weighted_recall]
macro_avgs = [macro_f1_score, macro_precision, macro_recall]

# Create DataFrame table of metrics
metrics = pd.DataFrame({'Precision': precision, 'Recall': recall, 'F1-Score': f1_score}, index=titles)
metrics = metrics.T
metrics['Weighted Avg'] = weighted_avgs
metrics['Macro Avg'] = macro_avgs
metrics.insert(4, '-', '-')
metrics = metrics.round(3)
metrics = metrics.T

return matrix, metrics
```

```
# plot the learning curve in increments of 6.25% of the data used on a grid with the labels being in %
```

```
def plot_learning_curve(x_train, y_train, x_test, y_test):  
    training_sizes = np.linspace(0.0625, 1.0, 16)  
    accuracies = []  
    for size in training_sizes:  
        x_train, x_test, y_train, y_test = split_data(x, y, size)  
        tree.fit(x_train, y_train)  
        predictions = tree.predict(x_test)  
        accuracies.append(accuracy(y_test, predictions))  
    plt.plot(training_sizes, accuracies, '-o')  
    plt.grid()  
    plt.xlabel('Training Data Used (%)')  
    plt.ylabel('Accuracy (%)')  
    plt.title('Learning Curve for Decision Tree Classifier')  
    plt.yticks(np.linspace(0.8, 1, 12), ['80%', '82%', '84%', '86%',  
                                         '88%', '90%', '92%', '94%', '96%', '98%', '100%', '102%'])  
    plt.xticks(rotation=45) # Rotate x labels by 45 degrees  
    plt.xticks(training_sizes, ['6.25%', '12.5%', '18.75%', '25%',  
                               '31.25%', '37.5%', '43.75%', '50%',  
                               '56.25%', '62.5%', '68.75%', '75%',  
                               '81.25%', '87.5%', '93.75%', '100%'])  
    plt.show()
```

```
# Label encoding for data
```

```
df = pd.read_csv('car.csv')
```

```
label_mapping = {
```

```
    '1': 1,  
    '2': 2,  
    '3': 3,  
    '4': 4,  
    'vhigh': 4,  
    'high': 3,  
    'med': 2,  
    'low': 1,  
    'big': 3,  
    'small': 1,  
    'more': 5,  
    '5more': 5,  
    'vgood': 4,
```



```
'good': 3,
'acc': 2,
'unacc': 1
}
df = df.map(lambda x: label_mapping.get(x, x))

#####
# MAIN PROGRAM #
#####

tree = DecisionTree(max_depth=10)
# Split data into training and testing
training = df.sample(frac=0.8)
testing = df.drop(training.index)
# Drop the attribute to be predicted
x = df.drop('class', axis=1).values
y = df['class'].values
x_train, x_test, y_train, y_test = split_data(x, y, 0.8)
# Start training
tree.fit(x_train, y_train)
# Make predictions
predictions = tree.predict(x_test)
# Calculate accuracy
acc = accuracy(y_test, predictions)

matrix, metrics = confusion_matrix(y_test, predictions)

print("\nTraining Size: {:5}".format(len(x_train)))
print("Testing Size: {:5}".format(len(x_test)))
print("Accuracy:    {:.3f}%".format(acc * 100))
print("\nConfusion Matrix: \n", matrix)
print("\nMetrics: \n", metrics, "\n")
plot_learning_curve(x_train, y_train, x_test, y_test)
```

Appendix 2: Neural Network Code

```
import random
import numpy as np
import time
import sys
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

class NeuralNetwork:
    def __init__(self):
        self.weights = None
        self.biases = None

    def init_parameters(self, input_neurons, hidden_neurons, output_neurons, epochs, lr, batch_size):
        # Initialize the neural network parameters
        self.input_neurons = input_neurons
        self.hidden_neurons = hidden_neurons
        self.output_neurons = output_neurons
        self.epochs = epochs
        self.learning_rate = lr
        self.batch_size = batch_size

        # Initialize weights and biases with random values
        self.biases = [np.random.randn(hidden_neurons, 1), np.random.randn(output_neurons, 1)]
        self.weights = [np.random.randn(hidden_neurons, input_neurons), np.random.randn(output_neurons,
hidden_neurons)]

    def train(self, train_data, test_data):
        # Train the neural network using mini-batch gradient descent
        accuracies = [] # Define the variable "test accuracies"
        for epoch in range(1, self.epochs + 1):
            start_time = time.time()
            random.shuffle(train_data) # Shuffle the training data

            # Create mini-batches from the training data
            mini_batches = [train_data[i:i + self.batch_size] for i in range(0, len(train_data), self.batch_size)]
```

```
for mini_batch in mini_batches:
    self.update_mini_batch(mini_batch)

# Evaluate the model on the test data and print the accuracy
accuracy = self.evaluate(test_data)
end_time = time.time()
accuracies.append(accuracy)
print(f"Epoch #{epoch} \nAccuracy = {accuracy / 100}% \nTime Elapsed {round(end_time - start_time,
2)}s\n")
print("Training complete")
return accuracies

def update_mini_batch(self, mini_batch):
    # Initialize the changes in biases and weights to zero
    bias_deltas = [np.zeros(b.shape) for b in self.biases]
    weight_deltas = [np.zeros(w.shape) for w in self.weights]

    for x, y in mini_batch:
        # Compute the gradient for the current mini-batch
        delta_b, delta_w = self.backpropagate(x, y)
        bias_deltas = [bd + d for bd, d in zip(bias_deltas, delta_b)]
        weight_deltas = [wd + d for wd, d in zip(weight_deltas, delta_w)]

    # Update the weights and biases using the gradients
    self.weights = [w - (self.learning_rate / len(mini_batch)) * wd for w, wd in zip(self.weights, weight_deltas)]
    self.biases = [b - (self.learning_rate / len(mini_batch)) * bd for b, bd in zip(self.biases, bias_deltas)]

def backpropagate(self, x, y):
    # Perform the backpropagation algorithm to compute the gradients
    x = np.array(x).reshape((len(x), 1)) / 255.0 # Normalize input
    y = int(y)

    # Feedforward pass
    z1 = np.dot(self.weights[0], x) + self.biases[0]
    a1 = self.sigmoid(z1)
    z2 = np.dot(self.weights[1], a1) + self.biases[1]
    a2 = self.sigmoid(z2)

    # Compute the output error
    y_vector = np.zeros((self.output_neurons, 1))
```

```
y_vector[y] = 1.0
output_error = (a2 - y_vector) * self.sigmoid_derivative(a2)

# Compute the hidden layer error
hidden_error = np.dot(self.weights[1].T, output_error) * self.sigmoid_derivative(a1)

# Compute the gradient for biases and weights
delta_biases = [hidden_error, output_error]
delta_weights = [np.dot(hidden_error, x.T), np.dot(output_error, a1.T)]

return delta_biases, delta_weights

def evaluate(self, test_data):
    # Evaluate the model on the test data
    correct = 0
    for x, y in test_data:
        if self.predict(x) == y:
            correct += 1
    return correct

def predict(self, x):
    # Predict the output for a given input
    x = np.array(x).reshape((len(x), 1)) / 255.0
    z1 = np.dot(self.weights[0], x) + self.biases[0]
    a1 = self.sigmoid(z1)
    z2 = np.dot(self.weights[1], a1) + self.biases[1]
    a2 = self.sigmoid(z2)
    return np.argmax(a2)

def sigmoid(self, z):
    # Sigmoid activation function
    z = np.clip(z, -500, 500) # Clip values to avoid overflow
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(self, a):
    # Derivative of the sigmoid function
    return a * (1 - a)

def load_data(train, test):
```

```
# Load and prepare the training data
train_set = np.loadtxt(train, skiprows=1, delimiter=',')
train_data = train_set[:, 1:]
train_labels = train_set[:, 0]
training_data = list(zip(train_data, train_labels))

# Load and prepare the testing data
test_set = np.loadtxt(test, skiprows=1, delimiter=',')
test_data = test_set[:, 1:]
test_labels = test_set[:, 0]
testing_data = list(zip(test_data, test_labels))

return training_data, testing_data

def main():
    # Run program: python3 nn.py 784 30 10 fashion-mnist_train.csv.gz fashion-mnist_test.csv.gz
    # Set hyperparameters
    epochs = 30
    learning_rate = 7
    batch_size = 40
    testing = True

    if testing:
        NInput = 784
        NHidden = 30
        NOutput = 10
        train_file = "fashion-mnist_train.csv.gz"
        test_file = "fashion-mnist_test.csv.gz"
    else:
        NInput = sys.argv[1]
        NHidden = sys.argv[2]
        NOutput = sys.argv[3]
        train_file = sys.argv[4]
        test_file = sys.argv[5]

    print("Loading training and testing data...")
    training_data, testing_data = load_data(train_file, test_file)
    print("Data Loaded...")

    print("Training the Neural Network...")
```

```
# Create and train the neural network
nn = NeuralNetwork()
learning_rates = [0.001, 0.01, 1.0, 10, 100]
minibatch_sizes = [1, 5, 20, 100, 300]

print("<HYPERPARAMETERS> | Learning Rate = ", learning_rate, "| Batch Size = ", batch_size, "| Cost
Function: Quadratic")

nn.init_parameters(NInput, NHidden, NOutput, epochs, learning_rate, batch_size)
accuracy = nn.train(training_data, testing_data)
plt.plot(range(1, epochs+1), accuracy, '-o', label=f"LR = {learning_rate}")
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.gca().yaxis.set_major_formatter(FuncFormatter(lambda y, _: f'{y/100:.0f}%'.format(y)))
plt.title('Experiment 4: Achieving Highest Accuracy')
plt.show()

main()

#####
# Code for Task 2 and Task 3 Graphs #
#####

# for i in learning_rates:
#     print("Task 2")
#     print("<HYPERPARAMETERS> | Learning Rate = ", i, "| Batch Size = ", batch_size, "| Cost Function:
Cross Entropy")
#     nn.init_parameters(NInput, NHidden, NOutput, epochs, i, batch_size)
#     accuracies = nn.train(training_data, testing_data)
#     plt.plot(range(1, epochs+1), accuracies, '-o', label=f"LR = {i}")
#     plt.grid()
#     plt.xlabel('Epoch')
#     plt.ylabel('Accuracy')
#     plt.gca().yaxis.set_major_formatter(FuncFormatter(lambda y, _: f'{y/100:.0f}%'.format(y)))
#     plt.title('Test Accuracy vs Epoch for Different Learning Rates')
#     plt.legend()
#     plt.show()

# acc = []
```

```
# for i in minibatch_sizes:
#     print ("Task 3")
#     print("<HYPERPARAMETERS> | Learning Rate = ", learning_rate, "| Batch Size = ", i, "| Cost Function:
Cross Entropy")
#     nn.init_parameters(NInput, NHidden, NOutput, epochs, learning_rate, i)
#     accuracies = nn.train(training_data, testing_data)
#     acc.append(max(accuracies))

# plt.plot(minibatch_sizes, acc, '-o')
# plt.grid()
# plt.xlabel('Batch Size')
# plt.ylabel('Accuracy')
# plt.gca().yaxis.set_major_formatter(FuncFormatter(lambda y, _: f'{y/100:.0f}%'.format(y)))
# plt.title('Test Accuracy vs Batch Size for Different Learning Rates')
# plt.show()
```