Question 1 of 6

Basic Questions for class (2)

Below is a short block of code describing a neural network using a python library called tensorflow.keras. Note, it takes a few minutes to run, so you might want to get it started now.

Create a prompt for the large language model of your choice that includes this code. Interview the model about the code until you're convinced you understand exactly how it works and what it's doing. Here are some things you'll need to know to be able to describe the network:

1. What data is it training on? The model is training on the MNIST dataset, which consists of 60,000 training images and 10,000 validation images of handwritten digits. Each image is 28x28 pixels, grayscale, representing digits from 0 to 9.
2. What is it learning to do? The model is learning to classify images of handwritten digits into one of 10 classes (digits 0-9). It aims to map each input image to the correct digit label.
3. How many layers does it have? The model has four layers:

A Flatten layer. Two Dense (fully connected) layers. A final Dense output layer.

1. What is a layer? A layer in a neural network is a collection of units (neurons) that apply transformations to the input data. Each layer receives input from the previous layer, applies computations (e.g., weighted sums, activations), and passes the output to the next layer.
2. How many units in the layers? The first Dense layer has 128 units. The second Dense layer has 64 units. The output Dense layer has 10 units.
3. What is a unit? A unit (or neuron) is a basic computational element in a layer. It takes inputs, multiplies them by weights, adds a bias, applies an activation function, and produces an output. The output is then passed to the next layer.
4. What are the units' activation functions? The activation functions used are:

The sigmoid function for both the first and second Dense layers. It squashes the output to a range between 0 and 1. The final layer has no activation function, meaning it outputs raw logits (real numbers) for each of the 10 classes.

1. How are the units connected to each other (i.e., how are the weights set initially?) The units in each layer are fully connected to the units in the next layer. The weights between the layers are initialized using random values according to a default initializer (typically Glorot uniform initialization in Keras), which ensures the weights are small enough to allow effective training but not too small to prevent learning.

2. What is a connection weight? A connection weight is a parameter that determines the strength of the connection between two units from adjacent layers. During training, these weights are updated based on how well the network is performing, helping the model learn the mapping from input to output.

3. What is the output? The output is a vector of 10 logits (real numbers) from the final Dense layer, each corresponding to one of the digit classes (0-9). Each logit represents the unnormalized score for a particular class.

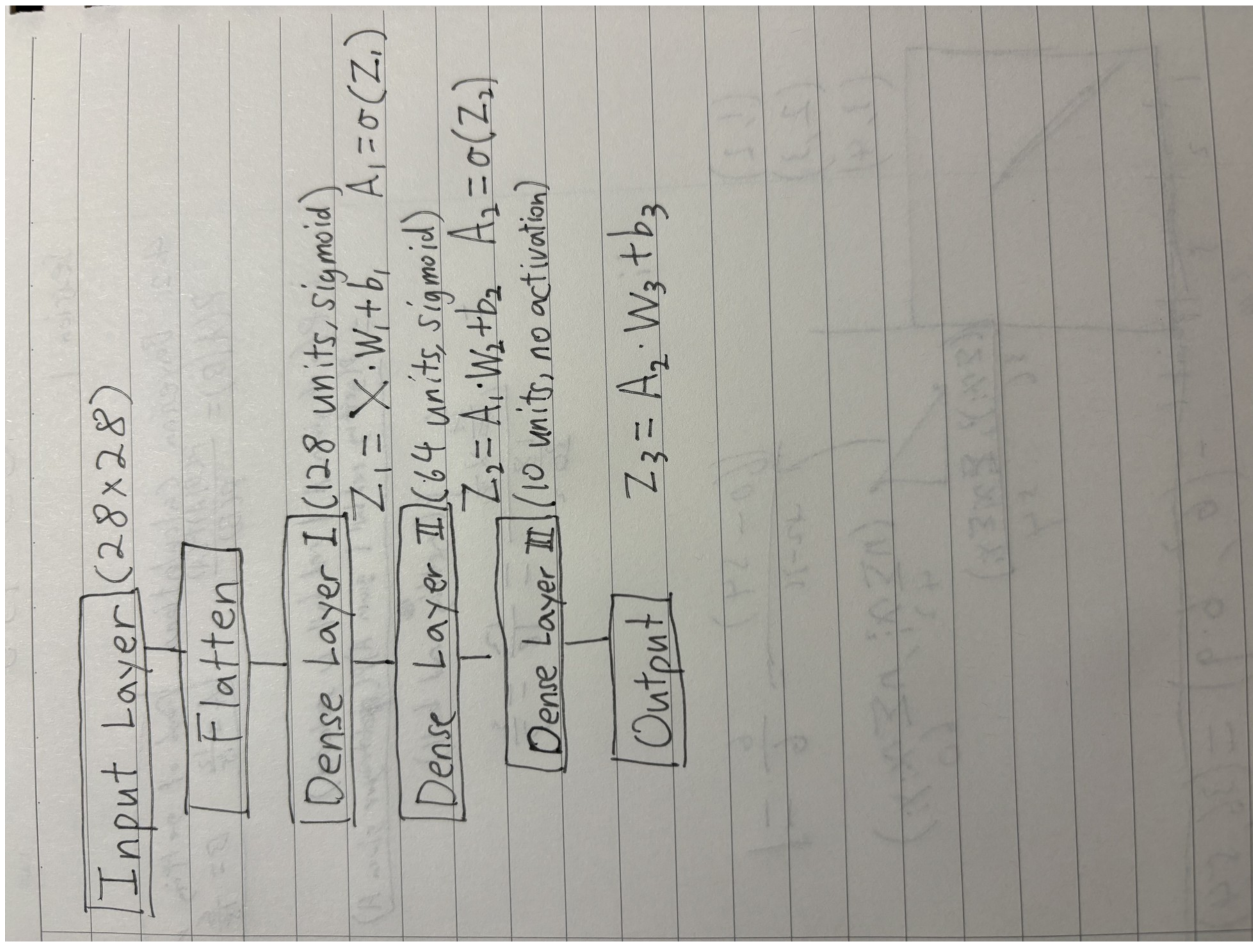
4. How well is it performing the classification? The performance metric used is accuracy. However, the model uses binary cross-entropy loss, which is not appropriate for this multi-class classification task. Using categorical cross-entropy would be more suitable, and using binary cross-entropy might result in suboptimal performance.

Question 2 of 6

Draw the network (you can draw each layer as a rectangle, you don't need to draw each unit), clearly label the connection, and write out the matrix multiplication that corresponds to doing a forward pass through the network. Upload this drawing in the file upload cell below.

```python
from IPython.display import Image, display

# Display the image
display(Image(filename='cs156_session7.jpeg'))
```


Input Layer (28×28)
Flatten
Dense Layer I (128 units, sigmoid)
Z₁ = X·W₁+b₁   A₁=σ(Z₁)
Dense Layer II (64 units, sigmoid)
Z₂ = A₁·W₂+b₂   A₂=σ(Z₂)
Dense Layer III (10 units, no activation)
Output
Z₃ = A₂·W₃+b₃

Variables:

- X: Input data (shape $(n \times 784)$, where n is the batch size and 784 is the number of features after flattening the 28x28 image).
- W_1: Weights of the first Dense layer (shape (784×128)).
- b_1: Bias of the first Dense layer (shape (128)).
- W_2: Weights of the second Dense layer (shape (128×64)).
- b_2: Bias of the second Dense layer (shape (64)).
- W_3: Weights of the output Dense layer (shape (64×10)).
- b_3: Bias of the output Dense layer (shape (10)).

Step 1: First Dense Layer

$$Z_1 = X \cdot W_1 + b_1$$

Where:

- Z_1 is the result before activation (shape $(n \times 128)$).
- Apply sigmoid activation:

$$A_1 = \sigma(Z_1)$$

Step 2: Second Dense Layer

$$Z_2 = A_1 \cdot W_2 + b_2$$

Where:

- Z_2 is the result before activation (shape $[n \times 64]$).
- Apply sigmoid activation:

$$A_2 = \sigma(Z_2)$$

Step 3: Output Layer

$$Z_3 = A_2 \cdot W_3 + b_3$$

Where:

- Z_3 is the final logits (shape $[n \times 10]$), representing the raw class scores for each of the 10 classes.
- No activation function is applied at this step, so the output is a raw vector of logits.

Question 3 of 6

Core Questions (3)

Below is a longer block of code for writing a simple perceptron from scratch.

Continue your discussion from the basic questions on this network. Describe the network that's being created. Be sure to include:

What is it learning to do? How many layers does it have? How many units in the layers? What are the units activation functions? How are the units connected to each other (i.e., how are the weights set initially?) What is the output? What do the functions predict() and train_weights() do?

Perceptron Network: Discussion and Breakdown

1. What is it learning to do?

The Perceptron model is learning to perform **binary classification**. It is applied to the **Sonar dataset**, which contains sonar signals reflected from two types of objects (rocks or mines). The model aims to classify whether the sonar signals come from a **rock** or a **mine**.

2. How many layers does it have?

This is a **single-layer Perceptron**:

- It has an **input layer** and an **output layer** but **no hidden layers**.

- The input layer connects directly to the output layer.

3. How many units in the layers?

- **Input Layer**: There are **60 units** (since the dataset has 60 features per data instance).
- **Output Layer**: There is **1 unit** because this is a **binary classification** task.

4. What are the units' activation functions?

The activation function in a Perceptron is a **step function**: \begin{cases} 1 & \text{if } x \geq 0 \ 0 & \text{if } x < 0 \end{cases}

The step function converts the weighted sum of inputs into a binary output.

5. How are the units connected to each other (i.e., how are the weights set initially)?

Each input unit is connected to the output unit through a **weight**. Initially, the weights are set to **zero**, and they are updated during training via **stochastic gradient descent**.

The weight update rule is:

$$ w_i = w_i + \\eta \\times \\text{error} \\times x_i $$

Where:

- w_i is the weight of the i-th input.
- $\\eta$ is the learning rate.
- **error** is the difference between the true output and the predicted output.
- x_i is the value of the i-th input.

6. What is the output?

The output is a **binary classification**:

- **1** indicates a **mine**.
- **0** indicates a **rock**.

This binary output is determined by applying the step activation function to the weighted sum of the inputs.

7. What do the functions `predict()` and `train_weights()` do?

Function: `predict()`

The `predict()` function:

- Takes a row of input data and the current weights.
- Computes the **weighted sum** (activation) of the inputs and weights, adding a bias term.

- Applies the **step function** to the weighted sum to make a prediction (either 0 or 1).

Function: `train_weights()`

The `train_weights()` function:

- **Trains the model** using **stochastic gradient descent**.
- For each training instance, it:
  a. Calls `predict()` to make a prediction.
  b. Computes the **error** (actual class minus predicted class).
  c. Updates the weights based on the error, learning rate, and input values.
- This process is repeated for a specified number of **epochs**, allowing the model to refine the weights.

8. How should I interpret the scores?

The scores you get from the output of the code represent the accuracy of the Perceptron model on different folds of the dataset, as part of cross-validation. Here's how you can interpret these scores:

1. Each Score Represents Fold Accuracy The dataset is split into a specified number of folds (in your case, n_folds = 3). For each fold, the Perceptron model is trained on the other two folds (the training set) and tested on the remaining fold (the test set). The accuracy for each fold is computed by comparing the predicted labels to the actual labels of the test fold. So, for example:

Score 1 might be the accuracy when Fold 1 was used as the test set, and Folds 2 and 3 were used for training. Score 2 might be the accuracy when Fold 2 was used as the test set, and Folds 1 and 3 were used for training. Score 3 might be the accuracy when Fold 3 was used as the test set, and Folds 1 and 2 were used for training.

1. Mean Accuracy The Mean Accuracy represents the average of the accuracy scores across all the folds. This gives you a general idea of how well the model is performing across the entire dataset, rather than just on a single train-test split.
2. Interpretation of the Scores High scores (e.g., above 85-90%) generally indicate that the model is performing well on the binary classification task. Lower scores (e.g., below 60-70%) may suggest that the model struggles with the classification task, either due to insufficient training data, overly simple model architecture (like the Perceptron), or inadequate hyperparameters (such as the learning rate or number of epochs).
3. Variation Between Scores If the scores vary significantly between folds (e.g., one fold has 90% accuracy and another has 60%), this could indicate that the model's performance is inconsistent, possibly due to overfitting or due to different data distributions in the folds. If the scores are consistent across the folds (e.g., 85%, 84%, 86%), this indicates that the model is performing consistently and generalizing well to unseen data. Next Steps: If the mean accuracy is acceptable for your use case, you can proceed with this model. If the mean accuracy is too low, you might consider: Increasing the number of epochs to allow the model more time to learn. Improving feature representation by using more complex techniques or models (since a simple Perceptron may not capture complex

patterns well). Tuning hyperparameters like the learning rate or adjusting the dataset size.

9. How to epoch and learning rate affect the model?

Both epochs and learning rate are crucial hyperparameters in training a machine learning model like a Perceptron. They directly impact how the model learns from the data. Here's a breakdown of their effects:

1. Epochs An epoch refers to one complete pass through the entire training dataset. In a multi-epoch training process, the model is exposed to the same data multiple times.

How Epochs Affect the Model: Too Few Epochs: If the number of epochs is too small, the model may not have enough opportunities to learn the patterns in the training data. This is called underfitting. Symptoms of too few epochs include low accuracy on both the training and test sets because the model hasn't had enough time to learn. Optimal Number of Epochs: With the right number of epochs, the model will reach a point where it has learned the patterns in the training data and generalizes well to unseen data. The performance on the test set (validation data) will plateau or improve slowly. Too Many Epochs: If you use too many epochs, the model will continue learning from the training data even after it has learned the main patterns. It may start learning noise and random fluctuations, leading to overfitting. Symptoms of too many epochs include high accuracy on the training set but low accuracy on the test set, meaning the model has learned too much about the specific data and performs poorly on unseen data. Guideline: Start with a moderate number of epochs (e.g., 100-200 for simpler models). Track the training and validation accuracy. If the training accuracy continues improving while validation accuracy stagnates or decreases, it's a sign of overfitting, and you should stop training.

1. Learning Rate The learning rate controls how much the model's parameters (weights) are adjusted with respect to the error at each step of training. It determines how fast the model learns.

How Learning Rate Affects the Model: Too Low Learning Rate:

A very low learning rate causes the model to learn very slowly. It may take many epochs for the model to converge (reach a stable, minimal error). The model might get stuck in local minima of the loss function and fail to learn optimal weights. Symptoms: The training process is slow, and the model may not improve even after many epochs. The learning curve (accuracy or loss over epochs) appears flat. Optimal Learning Rate:

An optimal learning rate balances speed of convergence and stability. The model quickly reaches a good solution without overshooting the optimal weights. Symptoms: The training accuracy improves steadily over epochs, and the validation accuracy tracks similarly, with both converging at a satisfactory point. Too High Learning Rate:

If the learning rate is too high, the model will adjust weights too drastically, which can cause it to overshoot the optimal weight values. The model may never converge to a good solution and may oscillate or diverge (where the training loss increases or fluctuates wildly). Symptoms: The training accuracy may fluctuate heavily, and validation accuracy will likely stay low or degrade. The learning curve appears chaotic or inconsistent. Guideline: Start with a moderate learning rate (e.g., 0.01 or 0.001) and adjust based on how the training process goes. If the model is learning too slowly, increase the learning rate. If it's unstable, reduce the learning rate. Effects of

Epochs and Learning Rate Combined: The number of epochs and the learning rate are interrelated in their effects on model performance:

Few Epochs + High Learning Rate: The model will adjust weights quickly but may not have enough time (epochs) to converge to an optimal solution. Few Epochs + Low Learning Rate: The model learns very slowly and might stop training before it reaches a good solution. Many Epochs + High Learning Rate: The model may diverge or oscillate, never reaching a stable solution. Many Epochs + Low Learning Rate: The model may take a long time to converge, but it may eventually reach a stable solution (though possibly overfitting if too many epochs are used).

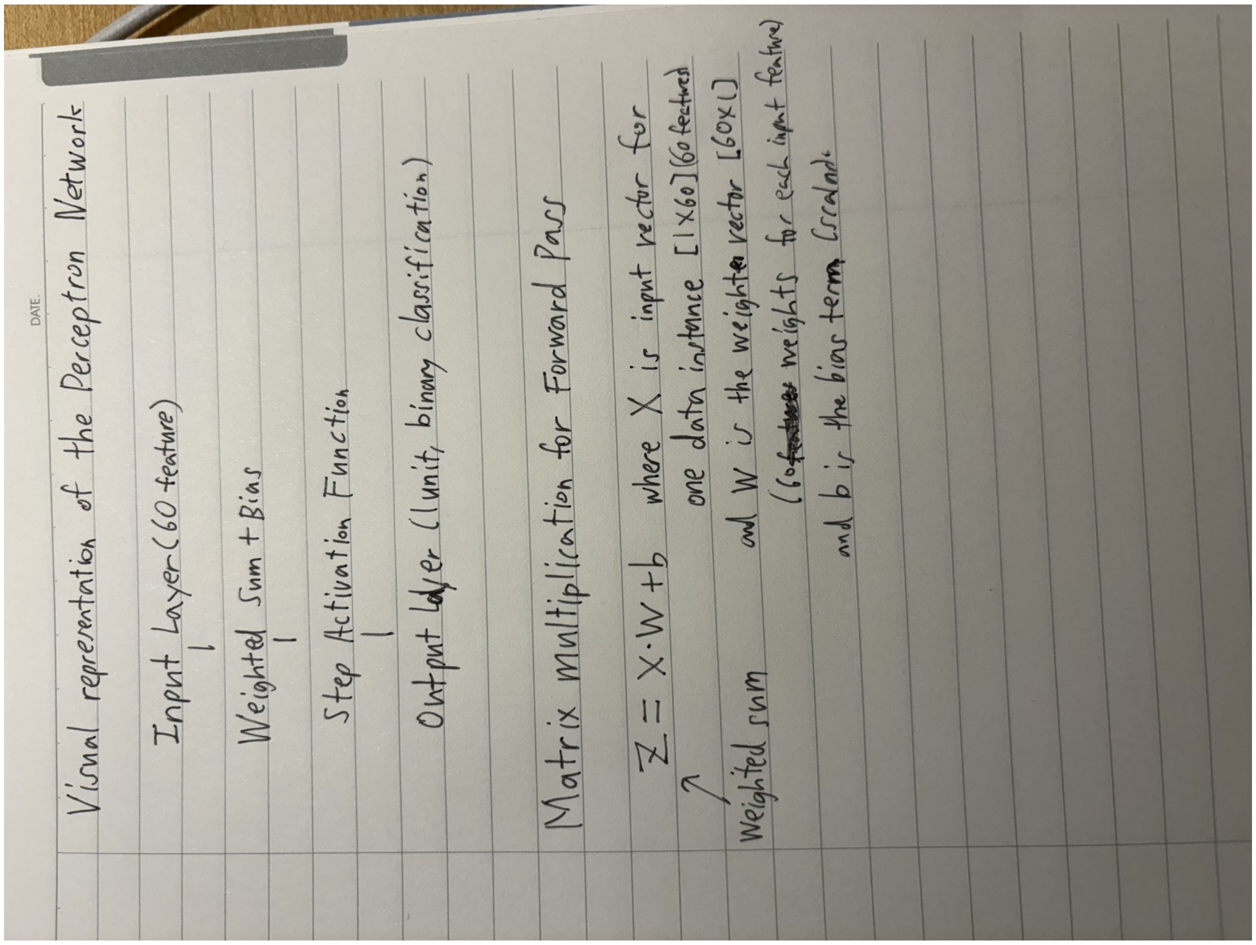
10. What are loss functions, and how do they affect model learning?

Question 4 of 6

Draw the network (you can draw each layer as a rectangle, you don't need to draw each unit), clearly label the connection, and write out the matrix multiplication that corresponds to doing a forward pass through the network. Upload this drawing in the file upload cell below.

```python
from IPython.display import Image, display

# Display the image
display(Image(filename='session7-2.jpeg'))
```



DATE.

Visual representation of the Perceptron Network

Input Layer (60 feature)
↓
Weighted Sum + Bias
↓
Step Activation Function
↓
Output layer (1 unit, binary classification)

Matrix multiplication for Forward Pass

$Z = X \cdot W + b$ where X is input vector for one data instance [1×60] (60 features)

Weighted sum

and W is the weighted vector [60×1] (60 ~~features~~ weights for each input feature)

and b is the bias term (scalar).

```python
import numpy as np
import tensorflow as tf
from random import seed, randrange
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

# Load Fashion MNIST dataset
def load_fashion_mnist():
    (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.fashion_mnist.load_data()
    # Flatten the 28x28 images into 784-dimensional vectors
    x_train = x_train.reshape((x_train.shape[0], 28 *
28)).astype('float32')
    x_test = x_test.reshape((x_test.shape[0], 28 *
28)).astype('float32')
    # Normalize pixel values (0-255) to (0-1)
    x_train /= 255
    x_test /= 255
    return x_train, y_train, x_test, y_test
```

```python
# Apply PCA to reduce dimensionality
def apply_pca(x_train, x_test, n_components=50):
    pca = PCA(n_components=n_components)
    x_train_pca = pca.fit_transform(x_train)
    x_test_pca = pca.transform(x_test)
    return x_train_pca, x_test_pca

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross-validation split
def evaluate_algorithm(dataset, labels, algorithm, n_folds, *args):
    dataset = list(zip(dataset, labels))  # Combine dataset and labels
    folds = cross_validation_split(dataset, n_folds)
    scores = list()

    for i, fold in enumerate(folds):
        # Split the data into training and test sets
        train_set = [item for j, sublist in enumerate(folds) if j != i
for item in sublist]
        test_set = fold

        # Train and test the model
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[1] for row in fold]  # Extract actual labels
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

# Make a prediction with weights
def predict(row, weights):
    activation = weights[0]
    for i in range(len(row)):
        activation += weights[i + 1] * row[i]
    return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
    weights = [0.0 for i in range(len(train[0][0]) + 1)]  # Initialize
weights with bias term
    for epoch in range(n_epoch):
        for row in train:
            inputs = row[0]  # Extract input features
            prediction = predict(inputs, weights)
            error = row[1] - prediction
            weights[0] += l_rate * error  # Update bias term
            for i in range(len(inputs)):
                weights[i + 1] += l_rate * error * inputs[i]  # Update
weights
    return weights

# Perceptron Algorithm With Stochastic Gradient Descent
def perceptron(train, test, l_rate, n_epoch):
    predictions = list()
    weights = train_weights(train, l_rate, n_epoch)
    for row in test:
        prediction = predict(row[0], weights)  # row[0] contains
features
        predictions.append(prediction)
    return predictions

# Main function to run the Perceptron algorithm on Fashion MNIST
seed(1)

# Load Fashion MNIST dataset
x_train, y_train, x_test, y_test = load_fashion_mnist()

# Reduce the dataset size for faster testing (e.g., only use 1000
samples)
x_train, y_train = x_train[:1000], y_train[:1000]
x_test, y_test = x_test[:1000], y_test[:1000]

# Apply PCA to reduce dimensionality to 50 principal components
x_train_pca, x_test_pca = apply_pca(x_train, x_test, n_components=50)

# We will perform binary classification by keeping only two classes
(e.g., 0 = T-shirt, 1 = Trouser)
binary_train_indices = np.where((y_train == 0) | (y_train == 1))
binary_test_indices = np.where((y_test == 0) | (y_test == 1))
x_train_pca, y_train = x_train_pca[binary_train_indices],
y_train[binary_train_indices]
x_test_pca, y_test = x_test_pca[binary_test_indices],
y_test[binary_test_indices]

# Normalize the labels to 0 and 1 (for binary classification)
y_train = np.where(y_train == 0, 0, 1)
y_test = np.where(y_test == 0, 0, 1)

# Evaluate algorithm
n_folds = 3
l_rate = 0.01
n_epoch = 100  # Reduce the number of epochs to 100 for faster
performance
scores = evaluate_algorithm(x_train_pca, y_train, perceptron, n_folds,
l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

```
Scores: [95.71428571428572, 94.28571428571428, 97.14285714285714]
Mean Accuracy: 95.714%
```

This model is a Perceptron-based algorithm applied to the Fashion MNIST dataset for binary classification. The Fashion MNIST dataset contains images of clothing items, and in this case, the model is tasked with distinguishing between two classes (e.g., T-shirt vs. Trouser). Here's an overview of how the model works:

1. Dataset Loading and Preprocessing Fashion MNIST dataset is loaded, which consists of 28x28 grayscale images of clothing items. These images are flattened into 1D arrays of 784 pixels (since 28x28 = 784), and the pixel values are normalized between 0 and 1 to help the model learn more effectively. To reduce the number of features and speed up computation, Principal Component Analysis (PCA) is applied, reducing the feature size to 50 components while retaining the most important information from the original 784-pixel images.
2. Binary Classification Setup The model is designed to classify between two classes: T-shirts (label 0) and Trousers (label 1). The dataset is filtered to keep only these two categories. The labels are normalized to 0 and 1, with 0 representing T-shirts and 1 representing Trousers.
3. Cross-Validation The data is split into 3 folds using cross-validation. This ensures that the model is tested on different subsets of the data, which helps assess how well it generalizes to new, unseen data. In each iteration of cross-validation, two folds are used for training the model, and one fold is used for testing.
4. Perceptron Algorithm The Perceptron is a simple linear classifier that learns by adjusting its weights based on the errors it makes during prediction. It uses stochastic gradient descent to update the weights. For each training example, the Perceptron calculates the prediction, compares it to the actual label, and updates the weights based on the error. The model runs for 100 epochs (iterations over the entire training set) with a learning rate of 0.01, which determines the step size of weight updates.
5. Evaluation The model's performance is measured using accuracy, which is the percentage of correct predictions made on the test data. After cross-validation, the model outputs the accuracy scores for each fold and calculates the mean accuracy to give an overall sense of its performance.