Section 1: Project Implementation

To implement this project, consider using the Hidden Markov Model.

1- To prepare the input data, select the necessary preprocessing steps, describe each of them, and apply them to the images.

1- Import the required libraries for the final project implementation:

```
import torch
import torch.nn as nn
import numpy as np
from torchvision.datasets import FashionMNIST
from torch.utils.data import DataLoader
from torchvision import transforms
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import tqdm

from hmmlearn import hmm
```

We import the required libraries for the final project implementation as follows:

• Torch: We use this library to create the network structure, receive the dataset, perform preprocessing on the images, and train and evaluate the model.

• Numpy: We use this library to work with input images as input matrices and perform mathematical operations on them.

• Sklearn: We use this library to compute evaluation metrics.

• Matplotlib.pyplot: We use this library to display graphs.

• Tqdm: We use this library to display the length of the training process.

• Hmmlearn: We use this library to train and evaluate the Hidden Markov Model.

2- Define the necessary preprocessing steps:

```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=(-10, 10)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Necessary preprocessing steps for the training data: Since the project's solution structure involves initially extracting a feature vector from the set of input images, it is better to use methods that increase the diversity between the images so that similar categories can be distinguished from each other to a good extent:

• transforms.RandomHorizontalFlip(p=0.5): This image transformation flips the image horizontally with a probability of p = 0.5.

• transforms.RandomRotation(degrees=(-10, 10)): This image transformation randomly rotates the image in the rotation direction between 10 and -10 degrees.

• transforms.ToTensor(): This transformation converts the image to a PyTorch tensor. A PyTorch tensor contains numerical values and can be processed by artificial neural network models.

• transforms.Normalize((0.5,), (0.5,)): This function is used to normalize the training data so that the mean and standard deviation of the input values are equal to 0.5. In fact, it puts the input values between -1 and 1. This can help improve the model's learning.

Necessary preprocessing steps for the evaluation data:

•transforms.Normalize((0.5,), (0.5,)): This function is used to normalize the evaluation data so that the mean and standard deviation of the input values are equal to 0.5. In fact, it puts the input values between -1 and 1. This can help improve the model's learning.

3- Receiving the dataset and applying the defined preprocessing steps:

```
# Download the Fashion-MNIST dataset and labels
trainset = FashionMNIST(root='./data', train=True, download=True,
transform=train_transform)
testset = FashionMNIST(root='./data', train=False, download=True,
transform=test_transform)

# Define the data loaders
batch_size = 256
trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
testloader = DataLoader(testset, batch_size=batch_size, shuffle=True)
```

In this section, we use the Fashion_MNIST dataset library to receive the dataset in two parts related to training and evaluation data. When receiving the dataset, we specify the preprocessing steps for both training and evaluation data to be applied to the data.

Then we use the DataLoader library to prepare the data for entering the model training and evaluation process. We set the batch size of data to 256.

2- Model structure according to the task (classification) and input data type (image):

First, let's review the ultimate goal of the project. In this project, we aim to classify the images of the Fashion_MNIST dataset using the concept of Hidden Markov Model, and we expect the accuracy to reach over 85%. Since the input data type is an image, and the Hidden Markov Model takes a set of observations as input and adjusts the probabilities during the training process, by entering the matrix related to all images, the length of the model's observations will be very large, which makes the training and modeling process difficult, and as a result, we may not achieve the desired accuracy. Therefore, we design and adjust the structure of solving this project as follows:

- Receive the dataset and apply the mentioned preprocessing steps to better distinguish the categories and increase the diversity of images.

- Use a model to extract features from input images to reduce the dimensions of the images from their actual size to a 10-dimensional feature vector to reduce the number of Hidden Markov Model observations and make the training process easier.

- Since we have 10 different categories in the Fashion_MNIST dataset, we train 10 Hidden Markov Models with the training data and labels for each category.

- Then, by entering each evaluation data, we compare the probabilities generated by each Hidden Markov Model corresponding to each category, and determine the evaluation data category based on the highest probability generated by one of the Hidden Markov Models.

- Finally, we calculate the Accuracy, F1-score, and AUC-ROC metrics and display them.

3- Considering the above steps, design a Hidden Markov Model as a classifier and try to achieve 85% accuracy.

4- Defining a CNN model with attention blocks for feature extraction:

```python
class SelfAttention(nn.Module):
    def __init__(self, in_dim):
        super(SelfAttention, self).__init__()
        self.query_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim//8, kernel_size=1)
        self.key_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim//8, kernel_size=1)
        self.value_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim, kernel_size=1)
        self.softmax = nn.Softmax(dim=-2)
        self.gamma = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        m_batchsize, C, width, height = x.size()
        proj_query = self.query_conv(x).view(m_batchsize, -1, width*height).permute(0, 2, 1)
        proj_key = self.key_conv(x).view(m_batchsize, -1, width*height)
        energy = torch.bmm(proj_query, proj_key)
        attention = self.softmax(energy)
        proj_value = self.value_conv(x).view(m_batchsize, -1, width*height)

        out = torch.bmm(proj_value, attention.permute(0, 2, 1))
        out = out.view(m_batchsize, C, width, height)
        out = self.gamma*out + x
        return out, attention

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
        self.att1 = SelfAttention(16)
        self.dropout1 = nn.Dropout(p=0.5)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.att2 = SelfAttention(32)
        self.dropout2 = nn.Dropout(p=0.5)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.att3 = SelfAttention(64)

        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.dropout3 = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x, p1 = self.att1(x)
        x = self.dropout1(x)
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x, p2 = self.att2(x)
        x = self.dropout2(x)
        x = self.pool(nn.functional.relu(self.conv3(x)))
        x, p3 = self.att3(x)

        x = x.view(-1, 64 * 7 * 7)
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout3(x)
        x = self.fc2(x)
        return x
```

This section includes defining two classes, SelfAttention and CNN, which are used to implement a convolutional neural network using self-attention layers for feature extraction from input images. The reason for using attention blocks is that the goal is to produce a feature vector of only 1x10 dimensions from a 28x28 image, and some of the categories in the dataset are very similar and close to each other. Using attention blocks helps the network to pay more attention to the important and influential parts of the image for feature extraction, so that the generated feature vectors can be better and more accurately classified by Hidden Markov Models, ultimately leading to better accuracy.

SelfAttention class: This class defines a self-attention layer. This layer calculates an energy matrix for each pixel of the image using three convolutional layers (one for query mapping, one for key mapping, and one for value mapping).

In other words, instead of using fixed weights to calculate image features, three convolutional mappings are applied to Query, Key, and Value.

Query is a convolutional mapping of the image, which aims to produce a smaller-sized vector that represents points in the image that need attention. Key is similar to Query, a convolutional mapping of the image that aims to identify the relationship between query points and other points in the image. Value is another convolutional mapping of the image that aims to produce a representation of the image that shows attention to each point of the image based on the points specified in Query and Key. By combining these three mappings, an energy matrix is calculated for each pixel of the image. This energy matrix represents the relationship of each point in the image with all other points in the image. Then, using the Softmax function, a weighted value is assigned to each pixel of the image. Then, by multiplying the weighted matrix with the key mapping weighted matrix, an image with features based on the original image is obtained. Finally, this image with attention features is added to the original features with a trainable parameter (gamma). Using the self-attention layer, the network can pay attention to important points in the image and use them better for image classification. This method is used in many applications of image processing and natural language processing.

The CNN class defines a convolutional neural network using self-attention layers. This network consists of three convolutional layers with kernel size of 3x3 and filters of 16, 32, and 64, respectively. After each convolutional layer, a self-attention layer is applied. Then, using a Max Pooling layer with size of 2x2, the image size is halved. Then, the images are connected to a fully connected layer with 128 neurons and then to a fully connected layer with 10 neurons. ReLU activation function is used in each convolutional and fully connected layer, and Dropout layer is also used to prevent overfitting in the network.

Overall, this neural network receives an image with dimensions of 28x28 as input and returns a 10-dimensional feature vector as output.

5- Training the feature extraction network:

```
# Train the CNN model on the Fashion-MNIST training data
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
CNN_model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(CNN_model.parameters(), lr=0.001)

num_epochs = 20
train_loss = []
train_acc = []

for epoch in range(num_epochs):
    running_loss = 0.0
    correct = 0
    total = 0

    for i, data in tqdm.tqdm(enumerate(trainloader), total=len(trainloader)):
        inputs, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = CNN_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Calculate accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        running_loss += loss.item() * inputs.size(0)

    epoch_loss = running_loss / len(trainset)
    epoch_acc = correct / total
    train_loss.append(epoch_loss)
    train_acc.append(epoch_acc)

    print(f"Epoch {epoch+1}/{num_epochs}, loss: {epoch_loss:.4f}, accuracy: {epoch_acc:.4f}")
```

In this section, we train a convolutional neural network with self-attention layers on the Fashion-MNIST training data.

1. In the first line, we use the torch.device function to determine whether the network should run on the GPU or CPU. If a GPU is available, we perform the training operations with it.

2. In the second line, we create a CNN model and transfer it to the desired device or GPU.

3. In the third line, we define the nn.CrossEntropyLoss() function as the cost function for the network to extract features. This cost function is used for multi-class classification. In the fourth line, we define the Adam optimization algorithm to update the network's weights.

4. In the fifth line, we define the number of training epochs for the network.

5. In the sixth line, we define two lists to store the network's error and accuracy for each training epoch.

6. In the outer loop, the network is trained on the training data. In each training epoch, using the inner loop, all the training data is fed into the network, and the network's weights are updated using the Adam optimization algorithm.

7. In the ninth line, we use the torch.max function to compare the predicted label of the network with the actual label, and calculate the number of correct predictions for each training epoch.

8. In this part of the code, the network's accuracy is calculated for each training epoch. To do this, first using the torch.max(outputs.data, 1) function, the label predicted by the network for each data is calculated for each available data. The torch.max function returns the largest element in an array. Here, by calling torch.max(outputs.data, 1), the largest element in each row of the outputs.data matrix (which represents the network's predictions for the input data) is calculated. Here, the torch.max function returns two values: the value of the largest element in each row and the column number of that element. However, we do not need the column value here, so we use the underscore symbol to

represent the first value returned by the torch.max function, which points to the largest element in each row of the outputs.data matrix. The second value returned by the torch.max function, which is the predicted label of the network for each data, is assigned to the predicted variable. In the second line, the total number of training data is calculated using labels.size(0) and added to the total variable. In the third line, for each data, the predicted label is compared with its actual label using predicted == labels. Then, using the sum() function, the number of correct predictions made by the network for all the training data in each training epoch is calculated and added to the correct variable.

9. In the tenth and eleventh lines, the error calculated by the cost function is weighted, and using the loss.backward() function, the gradient of the error with respect to the network's weights is calculated and the weights are updated using optimizer.step().

10. In the thirteenth line, the error and accuracy of each training epoch are stored in the respective lists and printed in the output.

6- Displaying training process charts:

```
# Plot the training loss and accuracy
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
ax[0].plot(train_loss)
ax[0].set_title("Training Loss")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[1].plot(train_acc)
ax[1].set_title("Training Accuracy")
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Accuracy")
plt.show()
```

Using this code snippet, we plot the accuracy and error charts obtained for each iteration during the training process.

7- Feature Extraction from Input Images:

```
# Load the trained model
CNN_model.load_state_dict(torch.load('models/CNN_model.pth'))

# Extract feature vectors and labels for the Fashion-MNIST training data
CNN_model.eval()
train_features = []
train_labels = []
with torch.no_grad():
    for data in trainloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        features = CNN_model(inputs).cpu().numpy()
        train_features.append(features)
        train_labels.append(labels.cpu().numpy())
train_features = np.concatenate(train_features)
train_labels = np.concatenate(train_labels)

# Extract feature vectors and labels for the Fashion-MNIST test data
test_features = []
test_labels = []
with torch.no_grad():
    for data in testloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        features = CNN_model(inputs).cpu().numpy()
        test_features.append(features)
        test_labels.append(labels.cpu().numpy())
test_features = np.concatenate(test_features)
test_labels = np.concatenate(test_labels)
```

1. At first, we load the best trained model.

2. Then, we put the neural network in evaluation mode using the ()eval function. In evaluation mode, the neural network is taken out of the training state and there is no need to update its weights and biases.

3. Using a for loop, the feature vectors and labels of the training and evaluation data are extracted and stored. In each iteration of the for loop, the training or evaluation data is taken and passed through the neural network computations, and the extracted features are placed in the train_features or test_features list.

4. The labels of the training or test data are also stored in the train_labels or test_labels list. The feature and label values are concatenated into a single unit using the np.concatenate function in the train_features, test_features, train_labels, and test_labels arrays.

5. Finally, these extracted feature vectors from the training and evaluation images are used for classification in the Hidden Markov Model (HMM) model.

8- Training Hidden Markov Models (HMMs) for Classification:

```
# Train 10 HMMs for each class
num_states = 2 # number of hidden states for each HMM
num_classes = 10  # number of classes in the dataset
models = []  # list to store the HMM models

for c in range(num_classes):
    # Extract feature vectors for the current class
    train_features_c = train_features[train_labels == c]
    # Create an instance of the HMM model
    model_c = hmm.GaussianHMM(n_components=num_states, covariance_type='diag')
    # Train the HMM model using training data until convergence
    model_c.fit(train_features_c)
    while not model_c.monitor_.converged:
        model_c.fit(train_features_c, init_params='', verbose=False)
    models.append(model_c)
```

In this part of the code, 10 HMM models are created for each category in the data and trained using the extracted features from the training data. Each category has an HMM model, which can be used to classify the evaluation data.

1. The number of hidden states for each HMM model is determined by the num_states value. In this example, two hidden states have been chosen for each class.

2. The number of classes is determined by the num_classes value. In this example, the number of classes for the Fashion-MNIST dataset, which includes 10 classes, has been selected.

3. An empty list named models is created to store the HMM models.

4. A for loop is defined for each class of the training data. In each iteration of the loop, the feature vectors of the training data related to the current class (c) are extracted from the train_features array using the train_labels == c condition and placed in the train_features_c variable.

5. Then, an HMM model is created using the GaussianHMM function. The reason for using this function is to test, evaluate, and measure the accuracy of the model results. In this function, n_components is set to determine the number of hidden states, and covariance_type is set to determine the type of covariance matrix used in the model. In this example, the diag type has been chosen for covariance_type, which considers the covariance matrix as a diagonal matrix.

6. Then, the HMM model is trained using the fit function on the feature vectors of the training data related to the current class. During training, the model reaches convergence with its parameters.

7. After training the model, using the while loop, if the model has not yet converged, the model continues to be trained until convergence is reached.

8. The trained HMM model is added to the models list. After the for loop ends, the models list contains 10 HMM models for each data class. Finally, using these 10 HMM models, the test data can be classified, and the classification accuracy can be measured.

4 -In model evaluation, in addition to the above metric, F1-score and AUC-ROC evaluation metrics are used, and the results are reported and analyzed.

9- Model Evaluation on Data:

```
# Initialize a list to store predicted labels and true labels
predicted_labels = []
true_labels = []

# Initialize a list of thresholds for each class
thresholds = []

for test_feature, test_label in zip(test_features, test_labels):
    # Calculate log-likelihood scores for each class
    log_likelihoods = []
    for model in models:
        log_likelihood = model.score([test_feature]) # note that input to score should be 2D array
        log_likelihoods.append(log_likelihood)

    predicted_label = np.argmax(log_likelihoods)
    predicted_labels.append(predicted_label)
    true_labels.append(test_label)

    # Calculate probability of each test image belonging to each class
    proba = np.exp(log_likelihoods) / np.sum(np.exp(log_likelihoods))

    thresholds.append(proba[1])

# Convert boolean array to integer array
true_labels = np.array(true_labels, dtype=int)

# Calculate ROC curve and AUC score for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve((true_labels == i).astype(int), thresholds, pos_label=1)
    roc_auc[i] = auc(fpr[i], tpr[i])
```

In this part of the code, for each test image evaluation, the logarithm of the likelihood probability of the test image is calculated for each data class in the HMM model. Then, using these likelihoods, the test image is assigned to a class that has a higher likelihood.

1. Initially, two empty lists named predicted_labels and true_labels are created to store the predicted and actual labels of the test images.

2. An empty list named thresholds is created to store the threshold value of each class.

3. The for loop is executed for each test image. In each iteration of the loop, the test image feature and its actual label are placed in the test_feature and test_label variables, respectively. In lines 4 to 8, using another for loop, the logarithm of the likelihood probability of the test image is calculated for each HMM model associated with each category in the data. For this purpose, the score function is executed on each HMM model with input test_feature (which is a one-dimensional array and needs to be converted to a two-dimensional array). In each iteration of the loop, the likelihoods of the test image for each data class are stored in the log_likelihoods list.

4. In line 9, using the argmax function, the data class number with the highest likelihood is determined as the predicted label for the test image, and it is stored in the predicted_label variable.

5. Then, this predicted label and the actual label of the test image are placed in the predicted_labels and true_labels lists. Then, using the logarithm of the likelihood probabilities of the test image, the probability of the test image for each class is calculated. For this purpose, the log_likelihoods list is used, and it is normalized (exponentiated to the power of e) and then divided by the sum of probabilities for all classes.

6. The positive threshold for each category or HMM model with label 1 (which is specified in the proba list with index 1) is stored in the thresholds list.

7. Using the astype function, the actual label array of all test images is converted to an array of integers.

8. In lines 22 to 26, for each class, the ROC curve and its area (AUC) are calculated. For this purpose, the roc_curve function is executed with the actual label array, the list of thresholds, and the positive label, and the FPR and TPR values for each threshold are stored in the fpr and tpr lists. Then, using the auc function, the area under the ROC curve is obtained for each class and stored in the roc_auc list.

10- Displaying AUC-ROC Curve:

```
# Plot AUC-ROC for each class
plt.figure(figsize=(8,6))
for i in range(num_classes):
    plt.plot(fpr[i], tpr[i], lw=2, label='ROC curve of class %d (AUC = %0.4f)' % (i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
# plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('AUC-ROC for Each Class')
plt.legend(loc="lower right")
plt.show()
```

In this part, using the FPR and TPR values calculated for each category, the ROC curve and its area (AUC) are separately plotted for each category.

1. Using the plt.figure function, a new figure with a specified size (8 by 6 inches) is created.

2. Then, using the for loop and the range function, for each category, the ROC curve and its area (AUC) are plotted using the plt.plot function.

3. Here, using the lw attribute, we choose the line thickness, and using the label attribute, we put the curve label in the figure.

4. Then, using the plt.plot function, we draw an auxiliary line with a dashed pattern from point (0, 0) to point (1, 1) with a line thickness of 2.

5. Finally, using the plt.xlim and plt.ylim functions, we determine the range of the x and y axes on the figure.

6. Using the plt.ylabel and plt.title functions, we label the y-axis and title of the figure.

7. Finally, using the plt.legend function, we create a legend containing the labels of the plotted curves.

11- Calculation of Accuracy and F1-score Metrics:

```
# Calculate accuracy and f1 score
accuracy = accuracy_score(test_labels, predicted_labels)
f1 = f1_score(test_labels, predicted_labels, average='macro')

# Print accuracy, F1 score, and AUC score for each class
for i in range(num_classes):
    print(f"AUC-ROC for class {i}: {roc_auc[i]:.4f}")
print(f"Accuracy: {accuracy:.4f}")
print(f"F1 score: {f1:.4f}")
```

In this part, using the library and functions related to accuracy and F1-score metrics, we calculate them. Finally, we print the AUC area for each category along with the final accuracy and F1-score of the model in the output.

You can see the obtained results in the second section or results analysis section.

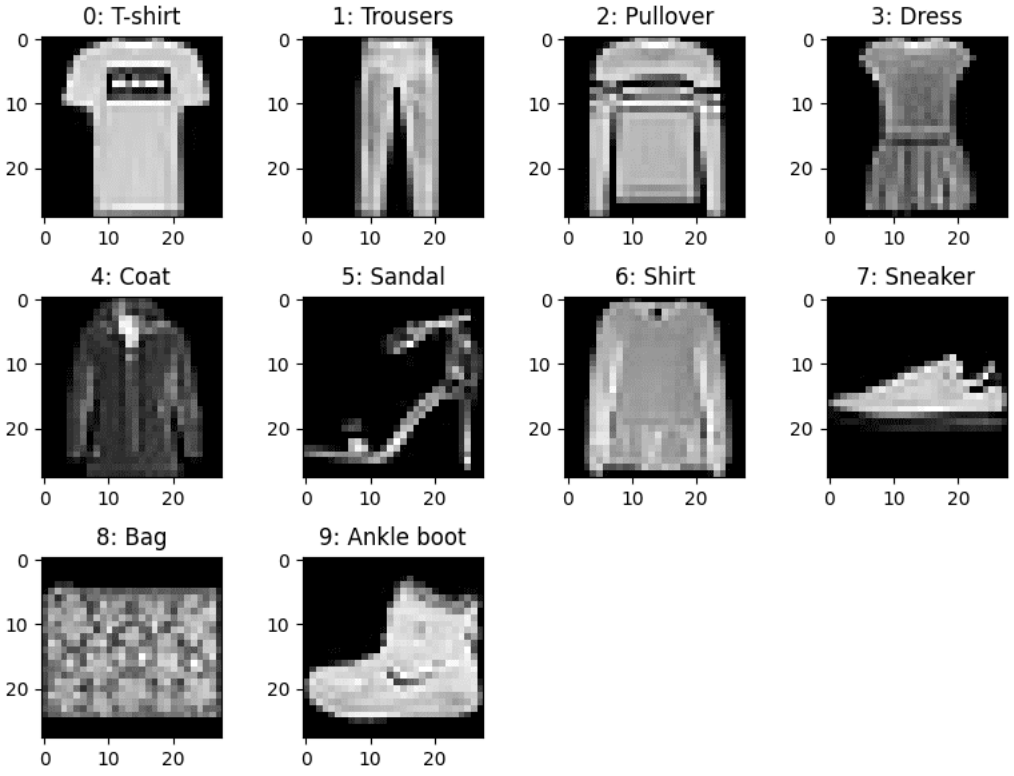
Part 2: Results Analysis

In this report, we first reviewed the concepts and solved the descriptive questions related to the hidden Markov model, and then we discussed the implementation details. Now we move on to the results and discussion section, where we present and analyze the results of the implemented models.

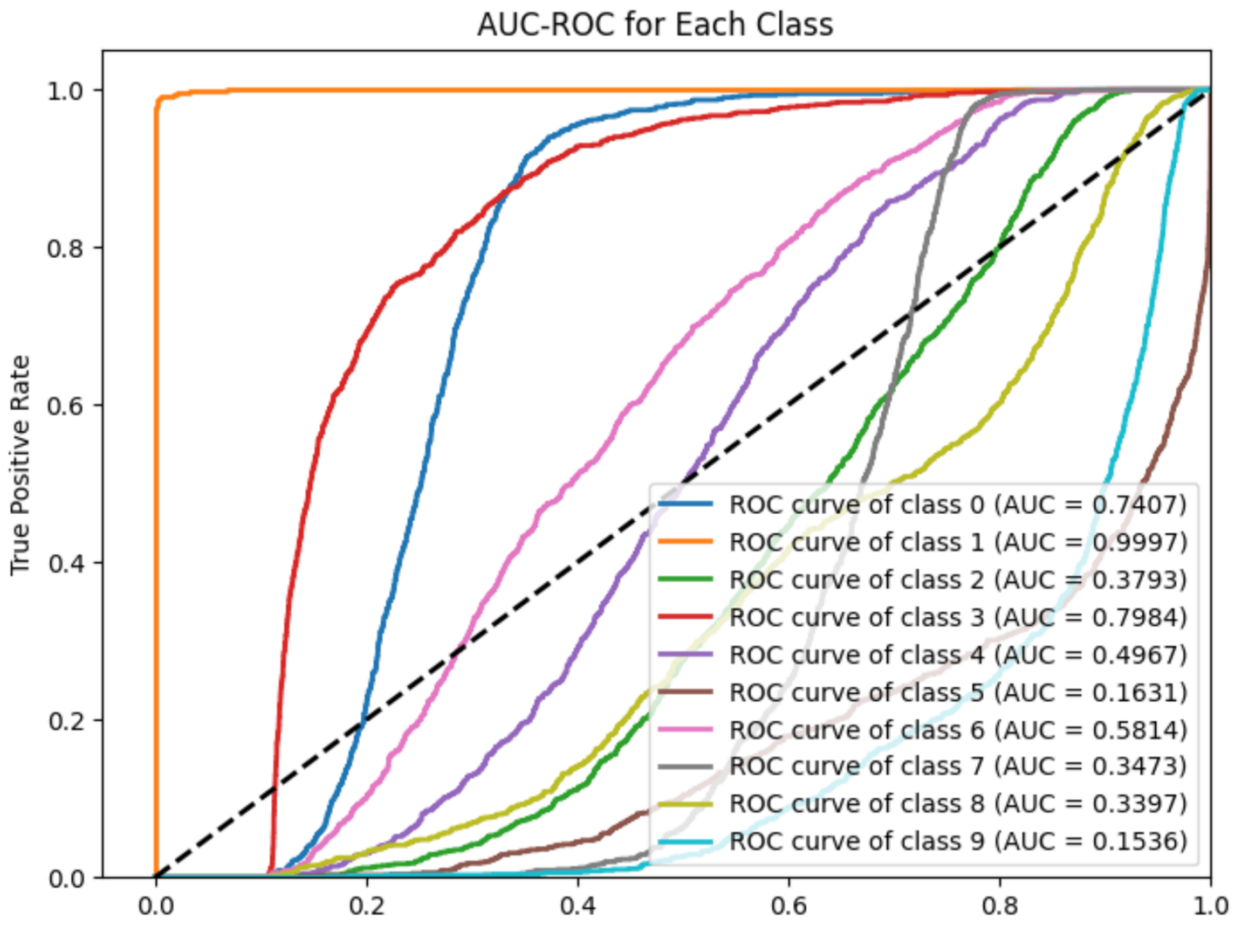
In this section, we will recapitulate the questions asked and examine the results obtained from the implementation of each section.

In evaluating the model, in addition to the high criterion, we used F1-score and AUC-ROC evaluation metrics, and report and analyze our results.

First, it is good to have a look at the categories available in the Fashion_MNIST dataset:

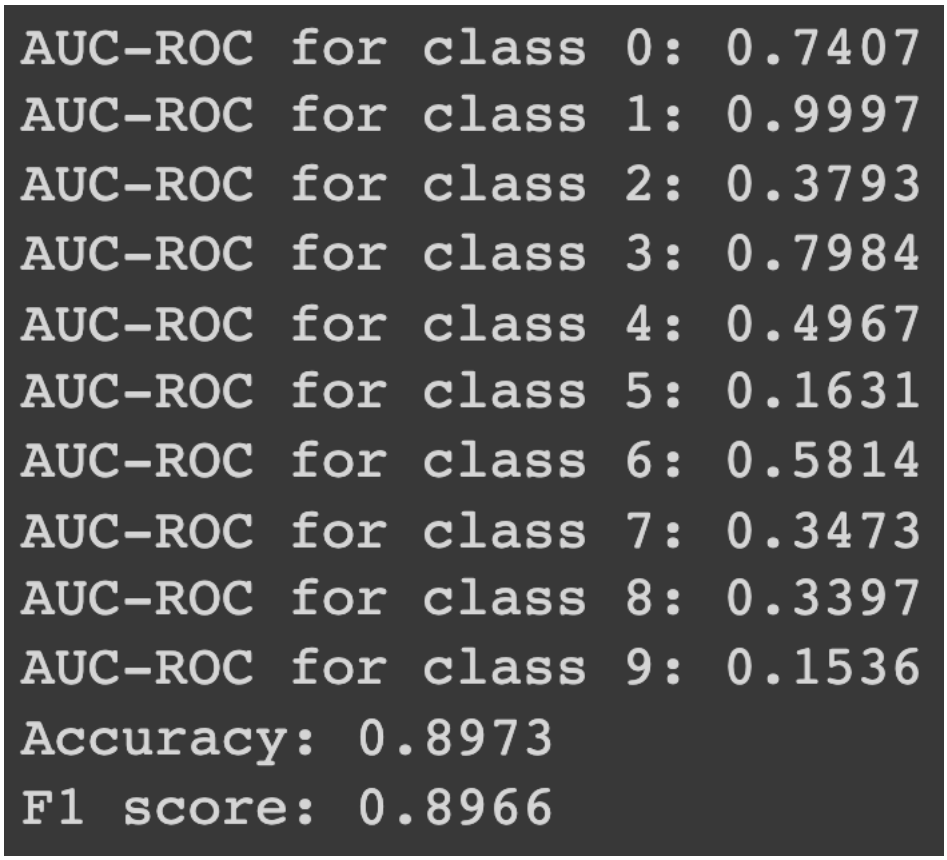


You can see the final AUC-ROC curve plotted for each category in the figure below:



As you can see in the above figure, the areas under the curve for categories 1, 3, and 0 were larger than the other categories, respectively. If we take a closer look at the images of these categories, we will notice that after applying the mentioned preprocessing steps and applying attention layers in the feature extraction network, they had better discrimination capability than the images corresponding to other categories in this dataset. Therefore, their extracted feature vectors were more distinguishable than other categories and hence, were classified better than other categories and achieved higher accuracy. However, on the other hand, categories that were very close to each other in terms of image pixels and were indistinguishable obtained lower scores in this evaluation metric. By increasing the number of data or making the feature extraction method more complex, we can improve these obtained accuracies to a certain extent.

You can see the AUC-ROC area for each category in the image below:



Also, as you can see in the above image, the model achieved an accuracy of 89% and an F1-score of 0.8966.