DOCUMENTATION

1. **Output Layer Modification:**

In the original code, the output layer had a single neuron:

```
   self.fc4 = nn.Linear(hidden_size3, output_size)
```

For multi-class classification, we modify the output layer to have five neurons, each representing a class:

```
self.output = nn.Linear(hidden_layer[-1], output_size)
```

Here, 'hidden_layer[-1]' is the size of the last hidden layer, and 'output_size' is the number of classes.

2. **Loss Function:**

In the original code, the loss function was defined as binary cross-entropy loss:

```
criterion = nn.CrossEntropyLoss()
```

For multi-class classification, we continue using nn.CrossEntropyLoss, which combines the softmax operation and cross-entropy loss calculation:

```
   criterion = nn.CrossEntropyLoss()
```

This loss function is suitable for multi-class classification as it expects raw logits from the network and applies SoftMax internally.

3. **Evaluation Metrics:**

In the original code, evaluation metrics like accuracy, precision, recall, and F1-score were calculated for binary classification. For multi-class classification, these metrics are still applicable but are typically averaged across all classes. For example, precision, recall, and F1-score can be computed as macro-averages or micro-averages across all classes:

```
accuracy = accuracy_score(y_test, predicted)
   precision = precision_score(y_test, predicted, average='macro')
   recall = recall_score(y_test, predicted, average='macro')
   f1 = f1_score(y_test, predicted, average='macro')
```

Here, 'average='macro'' computes the metric for each class and then takes the unweighted mean across all classes.

4. **Confusion Matrix:**

The confusion matrix provides insights into the performance of the classifier by showing the number of correct and incorrect predictions for each class.

It is applicable to both binary and multi-class classification problems. Each row represents the instances in an actual class, while each column represents the instances in a predicted class.

```
cm = confusion_matrix(y_test, predicted)
```

We then visualize the confusion matrix using matplotlib.

Modifications for Multi-Class Classification:

1. **Output Layer Update:** The most significant modification for multi-class classification was updating the output layer of the neural network. In binary classification, a single output neuron suffices, but for multi-class problems, the output layer should have neurons equal to the number of classes. Each neuron represents the likelihood of the corresponding class.

2. **Activation Function Change**: The activation function for the output layer needed to be changed to a suitable one for multi-class classification. The SoftMax function is commonly used in this scenario as it normalizes the output into a probability distribution across multiple classes.

3. **Loss Calculation Adjustment:** Since we have multiple output neurons now, we need to modify the loss calculation accordingly. The cross-entropy loss function is widely used for multi-class classification tasks as it measures the difference between the predicted probability distribution and the true distribution of the classes.

4. **One-Hot Encoding of Labels**: For training, the ground truth labels need to be one-hot encoded to match the shape of the output layer. This ensures that each class is represented as a distinct vector with a 1 at the index corresponding to the class and 0s elsewhere.

5. **Evaluation Metrics Update:** Evaluation metrics such as accuracy, precision, recall, and F1-score need to be calculated considering the multi-class nature of the problem. These metrics provide insights into the model's performance across all classes rather than just binary outcomes.

Challenges Faced and Solutions

1. **Output Layer Activation:** Selecting the appropriate activation function for the output layer was crucial. The SoftMax function was chosen as it converts the raw scores into probabilities, ensuring that the outputs sum up to 1 across all classes.

2. **Loss Calculation:** Adapting the loss calculation to handle multi-class scenarios required careful consideration. Cross-entropy loss is well-suited for this purpose, as it penalizes the model based on the dissimilarity between predicted and true class distributions.

3. **One-Hot Encoding:** Converting the labels into one-hot encoded vectors might increase the memory requirement for large datasets. However, it's necessary for training the model effectively in multi-class classification tasks.

[One-hot encoding is a technique used to represent categorical data, such as class labels, in a binary format. In one-hot encoding, each category is represented as a binary vector, where only one bit is set to 1 (hot) and the rest are 0s (cold). This encoding ensures that each category is distinct and independent of others, allowing the model to learn the categorical relationships effectively. In the context of multi-class classification, one-hot encoding is used to represent the ground truth labels, ensuring compatibility with the output layer of the neural network, where each neuron corresponds to a class.]

Overall, by making these modifications and addressing the challenges, the neural network was successfully adapted for multi-class classification, allowing it to effectively classify inputs into one of the five distinct classes.