In this module, we delve into the foundational principles of classification problems in machine learning. Classification entails predicting whether an element belongs to a specific category. There are three primary types of classification problems: binary classification, multi-class classification, and multi-label classification.

Neural Network Architecture for Classification Before diving into coding, let's explore the general architecture of a neural network for classification tasks. Here are some adjustable hyperparameters and key components depending on the classification type:

- **Binary Classification:**
  - Input layer shape (in_features): Number of features in the data.
  - Hidden layers: Number and configuration of hidden layers.
  - Neurons per hidden layer: Specific to the problem, typically between 10 and 512.
  - Output layer shape (out_features): 1 (representing one of two classes).
  - Hidden layer activation: Commonly ReLU or other activation functions.
  - Output activation: Sigmoid (torch.sigmoid in PyTorch).
  - Loss function: Binary crossentropy (torch.nn.BCELoss in PyTorch).
  - Optimizer: SGD (stochastic gradient descent), Adam, etc.
- **Multiclass Classification:**
  - Similar input layer shape, hidden layers, neurons per hidden layer, and hidden layer activation as in binary classification.
  - Output layer shape (out_features): 1 per class (e.g., 3 for food, person, or dog categories).
  - Output activation: Softmax (torch.softmax in PyTorch).
  - Loss function: Cross entropy (torch.nn.CrossEntropyLoss in PyTorch).
  - Similar optimizer as in binary classification.

1. Classification Data Preparation

- **1.1 Creating and Organizing Classification Data**
  - Classification data is generated using Scikit-Learn's make_circles() method, creating two circles with differently colored points, comprising two features (X1 and X2) and labels (y) containing values 0 or 1.
- **1.2 Data Visualization**
  - The data is displayed in a Pandas DataFrame and visualized using a scatter plot to understand the distribution and patterns.
- **1.3 Adjusting Input and Output**
  - The data's shape is explored for consistency, transformed into tensors for PyTorch compatibility, and a sample is examined to understand its structure.
- **1.4 Data Conversion and Splitting**
  - Data is converted into PyTorch tensors and split into training (80%) and testing (20%) sets using train_test_split, resulting in 800 training samples and 200 testing samples.

2. Model Building

- **2.1 Device-Agnostic Code Preparation**
    - Ensures the model runs on CPU or GPU, if available, ensuring device compatibility.
- **2.2 Model Construction**
    - A class inheriting nn.Module is created with two nn.Linear layers to accommodate the input and output data shapes.
- **2.3 Forward Function and Model Instance**
    - The forward() function outlines the model's forward computation. The model is instantiated and sent to the target device (CPU or GPU).
- **2.4 Building Model with nn.Sequential (Optional)**
    - Introduces simpler model construction using nn.Sequential, though custom nn.Module is often needed for more complex cases.
- **2.5 Model Prediction**
    - The model is tested by making predictions on test data, with results evaluated for appropriate output.
- **2.6 Loss Function and Optimizer Setup**
    - nn.BCEWithLogitsLoss() chosen for binary classification problems. SGD optimizer with a learning rate of 0.1 is used.
- **2.7 Evaluation Function (Optional)**
    - An accuracy_fn() function is created to measure model accuracy during training and evaluation.

3. Model Training

- **3.1 Model Output Conversion to Predicted Labels**
    - Before training, it's confirmed that the model produces outputs consistent with prediction labels. Model's raw output (logits) is converted into prediction probabilities and rounded to obtain labels.
- **3.2 Training and Testing Loop Construction**
    - Training and testing steps are built into loops, with model updates and progress printed every 10 epochs.
- **3.3 Initial Model Evaluation**
    - Despite training, performance is unsatisfactory. A stagnant accuracy around 50% suggests the model is no better than random decision-making. Further evaluation and adjustments might be needed.

4. Predictions and Model Evaluation

- **4.1 Initial Assessment Through Metrics**
    - The model appears to make random predictions, indicated by the stagnant accuracy. A deeper understanding requires visual analysis.
- **4.2 Decision Boundary Visualization and Analysis**
    - Visualizing the decision boundary shows the model tries to separate data with a straight line, which doesn't fit the circular data distribution. This explains the low performance.
- **4.3 Identifying Underfitting**

    - This condition, where the model fails to capture existing patterns in data, is termed underfitting.
- **4.4 Model Improvement**
    - To enhance performance, adjustments are needed for the model to capture more complex patterns, possibly by adding complexity or applying techniques like kernel tricks in Support Vector Machine (SVM).

5. Model Enhancement (From a Model Perspective)

- **5.1 Model Enhancement Approaches**
    - Focus on improving the model itself by adding layers or units, increasing epochs, altering activation functions, adjusting learning rates, or changing loss functions.
- **5.2 Model Enhancement Implementation**
    - Attempts to improve the model by adding layers, increasing hidden units, and extending training duration.
- **5.3 Exploring Issues**
    - Why is our model still failing? Testing is done with linear data to see if it can model effectively.
- **5.4 Linear Data Model Adjustments**
    - A model designed for linear data shows better results, indicating an issue with modeling circular data.

6. The Missing Puzzle: Non-linearity

- **6.1 Recreating Non-linear Data (Red and Blue Circles)**
    - The model can draw straight lines but needs capacity for non-linear lines.
- **6.2 Building a Model with Non-linearity**
    - Enhancing the model's capacity by adding non-linear activation functions like ReLU between hidden layers.
- **6.3 Training Model with Non-linearity**
    - The model, with ReLU added, is trained on new data (red and blue circles) while monitoring loss and accuracy.
- **6.4 Evaluating Non-linear Activation Function-Trained Model**
    - Post-training evaluation on test data and comparing decision-making with the linear-only model.

7. Duplicating Non-linear Activation Functions

- **7.1 Replicating ReLU Function**
    - Examining how non-linear activation functions, like ReLU, aid in modeling non-linear data.
- **7.2 Implementing ReLU**
    - Replicating ReLU using PyTorch, which changes negative values to 0 while keeping positive values unchanged.
- **7.3 Implementing Sigmoid**

    - Implementing sigmoid, used for generating values between 0 and 1, interpreted as probabilities.
- **7.4 Visualizing Results**
    - Visualizing the effects of these non-linear activation functions.

8. Combining Everything in a PyTorch Multi-Class Model

- **8.1 Creating Multi-Class Data**
    - Transitioning from binary to multi-class classification. Data is created using Scikit-Learn's make_blobs() method.
- **8.2 Constructing a Multi-Class Classification Model in PyTorch**
    - Building a model similar to previous ones but capable of handling multi-class data, including a BlobModel class with specific hyperparameters.
- **8.3 Setting Up Loss Function and Optimizer for Multi-Class Model**
    - Using nn.CrossEntropyLoss() for multi-class problems and continuing with SGD for optimization.
- **8.4 Obtaining Prediction Probabilities for Multi-Class Model**
    - Performing a forward pass to check functionality. Converting logits to prediction probabilities using softmax.
- **8.5 Training and Testing Loop for Multi-Class Model**
    - Writing loops for training and testing, adjusting steps to convert model output into prediction probabilities and labels.
- **8.6 Making and Evaluating Predictions with Multi-Class Model**
    - Evaluating the trained model's accuracy and visualizing predictions using plot_decision_boundary(), ensuring data is moved from GPU to CPU for use with matplotlib.

9. More Classification Evaluation Metrics Up to this point, we have discussed a few methods for evaluating classification models, such as accuracy, loss, and prediction visualization. These are common methods you'll encounter and serve as a good starting point. However, you might want to assess your classification models using additional metrics, as listed below:

- **Accuracy:**
    - Measures how many predictions are correct out of 100. For example, 95% accuracy means the model correctly predicts 95 out of 100 times.
- **Precision:**
    - The proportion of true positives relative to the total number of samples predicted positive. Higher precision leads to fewer false positives (predicting 1 when it should be 0).
- **Recall:**
    - The proportion of true positives relative to the total number of true positives and false negatives (predicting 0 when it should be 1). Higher recall leads to fewer false negatives.
- **F1-Score:**
    - Combines precision and recall into a single metric. A score of 1 is the best, while 0 is the worst.

- **Confusion Matrix:**
  - Compares predicted values with actual values in a tabular format. If 100% correct, all values in the matrix would be on the diagonal line from the top left to the bottom right.
- **Classification Report:**
  - A collection of several key classification metrics such as precision, recall, and f1-score.

Scikit-Learn, a popular and world-class machine learning library, provides many implementations of the above metrics. If you're looking for a PyTorch-compatible version, check out TorchMetrics, particularly the classification section. This library offers a suite of metrics specifically tailored for use with PyTorch, allowing for seamless integration and evaluation within PyTorch workflows.