KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Deep Learning

28-10-2024

BY
ASHA

Building ANN from Scratch

Neural Networks, Activation Functions, Loss Functions, and Optimizers

How Do We Build an Artificial Neural Network (ANN)?

Step 1: Initialize the Network Architecture

A basic ANN consists of three types of layers:

Input Layer: Receives the input features (e.g., pixel values of an image, words in a text, etc.).

Hidden Layers: Consist of neurons that apply a transformation to the input data using weights and biases.

Output Layer: Produces the final prediction, typically a classification or regression output.

Each connection between neurons has a **weight**, and each neuron has a **bias**.

Step 2: Apply an Activation Function

Each neuron takes a weighted sum of its inputs, applies a bias, and passes the result through an **activation function**. The role of activation functions is to introduce **non-linearity**, allowing the network to model complex data patterns.

Step 3: Forward Propagation – Generating Predictions

Once the architecture is set, the data is fed forward through the network:

Each neuron in the hidden layers calculates the weighted sum of its inputs, adds the bias, and applies the activation function.

The final layer outputs a prediction, often using activation functions like **Softmax** for multi-class classification or **Sigmoid** for binary classification.

Step 4: Define a Loss Function

The **loss function** quantifies how far off the network's predictions are from the actual values. Common loss functions include:

Mean Squared Error (MSE) for regression tasks.

Cross-Entropy Loss for classification tasks. The goal is to minimize this loss by adjusting the network's weights and biases.

Step 5: Backpropagation – Learning from Errors

After generating predictions, the network needs to learn from its mistakes. **Backpropagation** is the process of calculating the **gradient** of the loss function with respect to each weight in the network, using the chain rule.

This gradient indicates how each weight should be adjusted to reduce the overall loss.

Step 6: Use an Optimizer to Update Weights

The **optimizer** is responsible for adjusting the weights based on the gradients from backpropagation. Popular optimizers include:

Stochastic Gradient Descent (SGD): Updates weights using small, random batches of data.

Adam: Combines the advantages of SGD and other adaptive algorithms, allowing for faster and more efficient convergence.

Optimizers are key to ensuring that the model converges to an optimal solution efficiently.

Step 7: Train the Network

The training process consists of feeding the model with batches of data (called **epochs**), performing forward and backward passes, and updating weights to reduce the loss function. After several iterations, the model learns to make accurate predictions.

How the Core Components Affect Performance

- **Activation Functions**: The right activation function helps capture non-linearities in data and ensures that gradients flow properly through the network during backpropagation.
- **Loss Functions**: Selecting an appropriate loss function helps the model understand how to measure errors. For classification problems, **Cross-Entropy Loss** is effective, while for regression tasks, **Mean Squared Error** is commonly used.
- **Optimizers**: Optimizers like **Adam** adjust weights efficiently, improving convergence speed and ensuring the model finds a good local or global minimum in the loss function landscape. The choice of optimizer influences how quickly and how well the model learns.

1. Activation Functions: Shaping the Output

Activation functions introduce non-linearity into the model, allowing it to learn complex patterns. Here are some commonly used ones:

- **Sigmoid**: Maps inputs between 0 and 1, used often in binary classification tasks. However, it suffers from **vanishing gradients** for large inputs.
- **Tanh**: Outputs values between -1 and 1, making it zero-centered. It's an improvement over sigmoid but still faces the vanishing gradient issue for deep networks.
- **ReLU (Rectified Linear Unit)**: Outputs the input directly if it's positive; otherwise, it returns 0. It solves the vanishing gradient problem but may suffer from the **dying ReLU** issue (neurons stop updating for negative inputs).
- **Softmax**: Converts outputs into probabilities, commonly used in the final layer for multi-class classification problems.

2. Loss Functions: Quantifying Error

- Loss functions measure how far the predicted outputs are from the actual values, guiding the learning process:
- **Mean Squared Error (MSE)**: Used in regression tasks, it penalizes larger errors more heavily due to squaring.
- **Binary Cross-Entropy**: Common in binary classification, this loss function measures the performance of the model when predicting between two classes.
- **Categorical Cross-Entropy**: Extends binary cross-entropy for multi-class classification tasks.

3. Optimizers: Fine-Tuning the Learning Process

Optimizers adjust the model's weights based on the gradient of the loss function. They control how the model learns during training:

- **Gradient Descent**: The basic approach where weights are updated in the direction that minimizes the loss. Variants include:
  - **Batch Gradient Descent**: Uses the entire dataset for each update. Slow for large datasets.
  - **Stochastic Gradient Descent (SGD)**: Updates weights for each training sample, making it faster but noisier.
  - **Mini-Batch Gradient Descent**: A balance between batch and stochastic gradient descent, where updates are made on mini-batches.
- **Adam (Adaptive Moment Estimation)**: Combines the benefits of both RMSprop and momentum, adjusting the learning rate based on first and second moments of the gradient. It's one of the most widely used optimizers for deep learning.

Exercise 1: Regression on Insurance Dataset (Basic ANN with Gradient Descent)

Objective:

We focused on building a simple neural network for regression tasks. Using an insurance dataset, the task was to predict continuous outputs such as insurance charges based on input features like age, BMI, and region.

Key Concepts Covered

1. **Basic Structure of an ANN:**
    1. **Input Layer, Hidden Layer, Output Layer:** We designed a network with these layers to model the data.
    2. **Activation Function:** We applied the ReLU activation function for hidden layers and a linear activation function for the output layer, suited for regression problems.
2. **Forward Propagation:**
    1. We used forward propagation to pass inputs through the layers and predict outputs based on learned weights.
    2. The **Mean Squared Error (MSE)** loss function was applied to measure the difference between predicted and actual values.
3. **Backward Propagation & Gradient Descent:**
    1. Using gradient descent, we adjusted the weights to minimize the error by computing gradients during backward propagation.
    2. The importance of the learning rate was highlighted in controlling the speed of training and convergence.

Outcomes

- A solid understanding of ANN architecture and how data flows through it.
- Explored the role of forward and backward propagation in learning.
- Worked with loss functions and gradient descent to train the model.

Exercise 2: Classification on Churn Modeling (Binary Classification using Cross-Entropy and Gradient Descent)

Objective:

The focus was on applying ANNs to a binary classification task, predicting customer churn. We developed a model to determine if a customer would leave the service.

Key Concepts Covered

1.Binary Classification:

1. We transitioned from regression to classification, implementing a **sigmoid activation** in the output layer to output probabilities for binary outcomes.

2.Loss Function:

1. We used **Binary Cross-Entropy** as the loss function to measure the difference between the predicted probabilities and the actual class labels.

3.Gradient Descent in Classification:

1. We utilized gradient descent to adjust weights based on binary cross-entropy loss and sigmoid activation.

Takeaways

- Developed an understanding of how ANNs handle binary classification tasks.
- Worked with binary cross-entropy for classification loss.
- Learned to evaluate classification models using accuracy and other metrics.

Exercise 3: Regression on Insurance Dataset with Different Optimizers

Objective:

We revisited the insurance dataset to demonstrate how different optimizers influence model training. By changing the optimization algorithm, we examined the effects on convergence speed and accuracy.

Key Concepts Covered

1.Optimization Algorithms:

1. We explored advanced optimizers like **Adam**, **RMSprop**, and **Momentum**, comparing them to simple gradient descent. Each optimizer's impact on learning efficiency and speed was observed.

2.Challenges with Gradient Descent:

1. We addressed common issues like slow convergence or local minima, seeing how advanced optimizers like Adam and RMSprop help overcome these challenges.

3.Implementation of Adam Optimizer:

1. Adam optimizers was tested, and their effect on training and convergence rate was observed.

Takeaways

- Gained insight into how different optimizers impact the training process of neural networks.
- Compared gradient descent with more advanced optimizers to understand their practical implications.

Exercise 4: Multilayer ANN for Classification on Iris Dataset

Objective:

We will extend the neural network by adding multiple hidden layers to solve a multi-class classification problem using the Iris dataset. The goal is to classify different species of iris flowers based on input features like sepal length, sepal width, petal length, and petal width.

1.Multilayer Perceptron (MLP) for Classification:

1. We will build a deeper neural network by adding multiple hidden layers. This will allow the model to capture more complex patterns in the data for multi-class classification.

2.Softmax Activation for Multi-Class Classification:

1. In the output layer, we will apply the **softmax activation function**. This will help the network output probabilities for each class (setosa, versicolor, virginica), making it suitable for multi-class classification.

3.Loss Function - Categorical Cross-Entropy:

1. We will use **categorical cross-entropy** as the loss function, which is optimal for multi-class problems like this one.

Steps in the Code for Building a Multilayer ANN from Scratch for the Iris Dataset

Step 1: Import Libraries

Description: This step imports the necessary libraries and modules for data manipulation, dataset loading, splitting, standardization, and accuracy measurement.

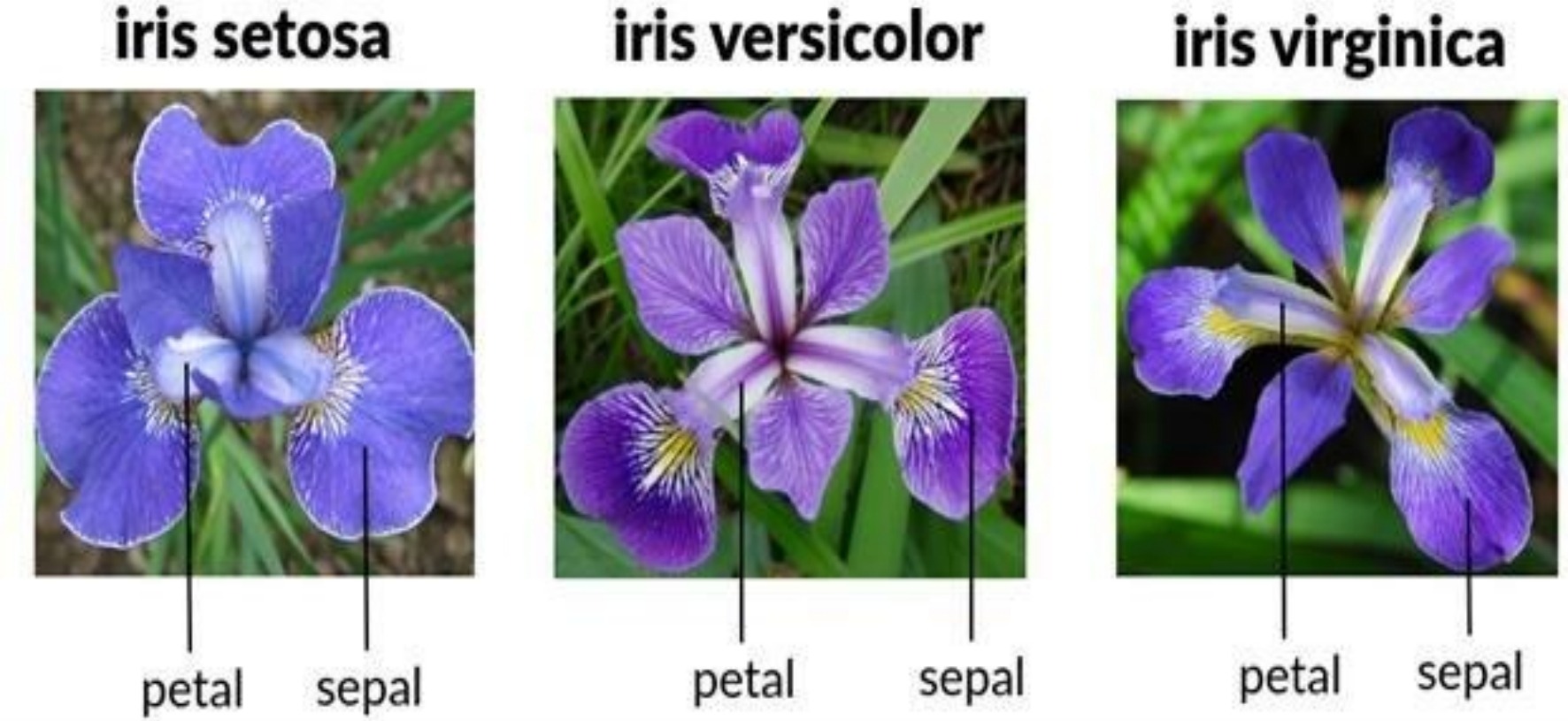
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

Step 2: Load the Iris Dataset

Description: load the Iris dataset and extracts the features and target labels:

- X: Contains the feature data (sepal length, sepal width, petal length, petal width).
- y: Contains the target labels (species of the iris).

```
data = load_iris()
X = data.data
y = data.target
```

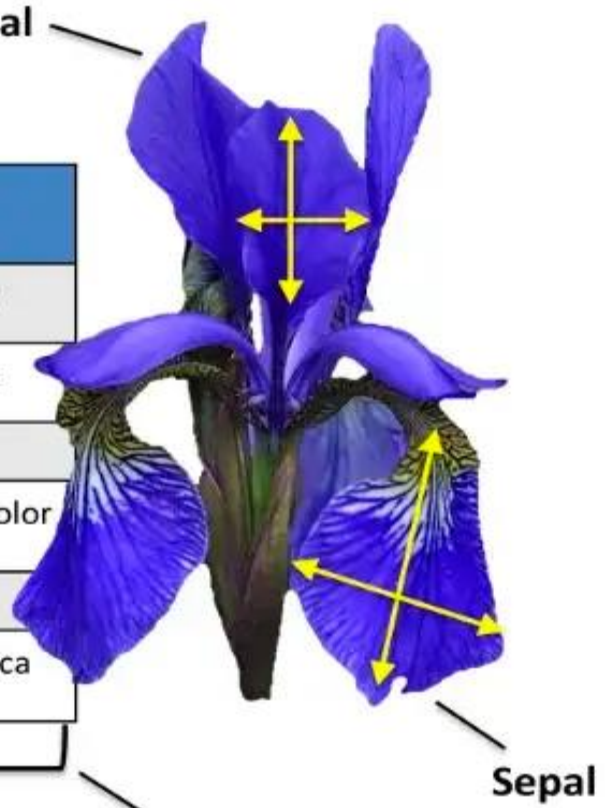

iris setosa
iris versicolor
iris virginica
petal
sepal
petal
sepal
petal
sepal

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Features
(attributes, measurements, dimensions)

Class labels
(targets)



Step3 : Convert Labels to One-Hot Encoding

Description: This step converts the target variable into a one-hot encoded format:

- **Function Definition**: The one_hot_encoding function creates a one-hot encoded representation of the target labels using NumPy's eye function, which generates a 2D identity matrix. For example, if a sample belongs to class 1, it will be represented as [0, 1, 0].
- num_classes: Specifies the number of unique classes (3 for the Iris dataset).
- y_one_hot: The resulting one-hot encoded labels, which are used for multi-class classification.

One-Hot Encoding: An Overview

One-hot encoding is a technique used in machine learning and data processing to convert categorical variables into a numerical format.

Example: One-Hot Encoding on a Sample Dataset

Sample Dataset

Consider a simple dataset with a feature "Color":

ID	Color
1	Red
2	Green
3	Blue
4	Green
5	Red

The transformed dataset after one-hot encoding would look like this:

ID	Color	Color_Red	Color_Green	Color_Blue
1	Red	1	0	0
2	Green	0	1	0
3	Blue	0	0	1
4	Green	0	1	0
5	Red	1	0	0

```python
# Convert labels to one-hot encoding
def one_hot_encoding(y, num_classes):
    return np.eye(num_classes)[y]

y_one_hot = one_hot_encoding(y, num_classes=3)
```

- **y**: The array of target labels (class indices).
- **num_classes**: The total number of unique classes present in the target variable.
- **np.eye(num_classes)**: This function from the NumPy library creates an identity matrix of size num_classes x num_classes. An identity matrix has ones on the diagonal and zeros elsewhere.

For instance, if num_classes is 3, the output will look like this:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

•**[y]**: This indexing operation selects rows from the identity matrix. Each entry in the target array y acts as an index to the identity matrix, effectively selecting the corresponding one-hot encoded vector for each class label.

•For example, if y is [0, 1, 2, 1], the function will map:

- •0 → [1, 0, 0] (Setosa)
- •1 → [0, 1, 0] (Versicolor)
- •2 → [0, 0, 1] (Virginica)
- •1 → [0, 1, 0] (Versicolor)

•**Return Value**: The function returns a 2D array where each row corresponds to the one-hot encoded vector for the respective class label in y.

- Here, we call the one_hot_encoding function with y and num_classes set to 3 (since there are three classes in the Iris dataset).
- The result is stored in y_one_hot, which will be a 2D array with shape (n_samples, n_classes) where n_samples is the number of samples in the dataset and n_classes is the number of unique classes (3 in this case).

Step 4 : Split the Dataset and Standardize Features

Description:

- **Split Dataset**: This part divides the dataset into training and testing sets:
  - train_test_split: A utility function that randomly splits the data.
  - X_train, y_train: The training data and corresponding one-hot encoded labels.
  - X_test, y_test: The testing data and corresponding labels.
  - test_size=0.2: Indicates that 20% of the data will be reserved for testing.
  - random_state=42: Ensures that the random splitting is reproducible.
- **Standardize Features**: This part standardizes the feature values:
  - StandardScaler: Used to standardize the features to have a mean of 0 and a standard deviation of 1.
  - fit_transform(): Computes the mean and standard deviation from the training data and applies the transformation.
  - transform(): Standardizes the testing data using the same statistics from the training data.

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Step 5: Initialize Weights and Biases

Description: This step initializes weights and biases for each layer:

- Weights are initialized using a normal distribution to introduce randomness.
- Biases are initialized to zeros, which is a common practice.

```python
def initialize_parameters(input_size, hidden_size1, hidden_size2, output_size):
    np.random.seed(42)  # For reproducibility

    # Weights and biases for input to first hidden layer
    weights_input_hidden1 = np.random.randn(input_size, hidden_size1) * 0.01
    bias_hidden1 = np.zeros((1, hidden_size1))

    # Weights and biases for first hidden layer to second hidden layer
    weights_hidden1_hidden2 = np.random.randn(hidden_size1, hidden_size2) * 0.01
    bias_hidden2 = np.zeros((1, hidden_size2))

    # Weights and biases for second hidden layer to output layer
    weights_hidden2_output = np.random.randn(hidden_size2, output_size) * 0.01
    bias_output = np.zeros((1, output_size))

    return {
        "weights_input_hidden1": weights_input_hidden1,
        "bias_hidden1": bias_hidden1,
        "weights_hidden1_hidden2": weights_hidden1_hidden2,
        "bias_hidden2": bias_hidden2,
        "weights_hidden2_output": weights_hidden2_output,
        "bias_output": bias_output
    }
```

Step 6: Define Activation Functions and compute loss function (CCE)

Description: This step defines the activation functions used in the network:

- **Sigmoid**: Used for the hidden layers, it introduces non-linearity.
- **Softmax**: Used for the output layer, it converts logits into probabilities for multi-class classification.

Kmit

```python
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    s = sigmoid(z)
    return s * (1 - s)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

$$\text{CCE} = -\frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{K} y_{ij} \cdot \log(\hat{y}_{ij} + \epsilon)$$

where:

- N is the number of samples in the batch.
- K is the number of classes.
- y_{ij} is the true label for class i of sample j (either 0 or 1 in a one-hot encoded format).
- $\hat{y}_{ij}$ is the predicted probability for class i of sample j.
- ϵ is a small constant (e.g., 1×10^{-8}) added to avoid taking the logarithm of zero.

```python
def compute_loss(y_true, y_pred):
    """Calculate the cross-entropy loss."""
    loss = -np.mean(np.sum(y_true * np.log(y_pred + 1e-8), axis=1))
    return loss
```

step7: Forward Propagation

Define a function to perform forward propagation:

Compute the hidden layer activation by applying the sigmoid function. Compute the output layer activation using the softmax function.

Implement the function forward():

(X, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2, bias_hidden2, weights_hidden2_output, bias_output)

```python
def forward(X, weights_input_hidden1, bias_hidden1, weights_hidden1_hidden2, bias_hidden2, weights_hidden2_output, bias_output):
    # First hidden layer
    z_hidden1 = np.dot(X, weights_input_hidden1) + bias_hidden1
    a_hidden1 = sigmoid(z_hidden1)

    # Second hidden layer
    z_hidden2 = np.dot(a_hidden1, weights_hidden1_hidden2) + bias_hidden2
    a_hidden2 = sigmoid(z_hidden2)

    # Output layer
    z_output = np.dot(a_hidden2, weights_hidden2_output) + bias_output
    a_output = softmax(z_output)

    return a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output
```

step8: Backward Propagation

Define a function to perform backward propagation:

Compute gradients of the loss function with respect to weights and biases using the chain rule. Update gradients for the weights and biases of both layers. Implement the function backward():

(X, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, weights_hidden1_hidden2, weights_hidden2_output)

```
def backward(X, y, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, weights_hidden1_hidden2, weights_hidden2_output):
    m = y.shape[0]

    # Output layer gradients
    dz_output = a_output - y
    dw_hidden2_output = np.dot(a_hidden2.T, dz_output) / m
    db_output = np.sum(dz_output, axis=0, keepdims=True) / m

    # Second hidden layer gradients
    dz_hidden2 = np.dot(dz_output, weights_hidden2_output.T) * sigmoid_derivative(z_hidden2)
    dw_hidden1_hidden2 = np.dot(a_hidden1.T, dz_hidden2) / m
    db_hidden2 = np.sum(dz_hidden2, axis=0, keepdims=True) / m

    # First hidden layer gradients
    dz_hidden1 = np.dot(dz_hidden2, weights_hidden1_hidden2.T) * sigmoid_derivative(z_hidden1)
    dw_input_hidden1 = np.dot(X.T, dz_hidden1) / m
    db_hidden1 = np.sum(dz_hidden1, axis=0, keepdims=True) / m

    return dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output
```

step9: Train the Neural Network

Define a function to train the network:

Use the forward and backward propagation functions to update weights and biases over multiple epochs. Print the loss value at regular intervals (e.g., every 100 epochs) to monitor training progress. Implement the function train():

Description: This step begins the training loop for the network:

- **Forward Pass**: Calculate activations for each layer using the weights and biases.
- **Compute Loss**: Calculate the loss using the categorical cross-entropy function.
- **Backward Pass**: Calculate gradients to update weights and biases (not fully shown here).

```python
def train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate=0.01, epochs=1000):
    # Initialize parameters with two hidden layers
    parameters = initialize_parameters(input_size, hidden_size1, hidden_size2, output_size)

    for epoch in range(epochs):
        # Forward pass
        a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2, z_output = forward(
            X_train,
            parameters["weights_input_hidden1"], parameters["bias_hidden1"],
            parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
            parameters["weights_hidden2_output"], parameters["bias_output"]
        )

        # Compute loss
        loss = compute_loss(y_train, a_output)

        # Backward pass
        dw_input_hidden1, db_hidden1, dw_hidden1_hidden2, db_hidden2, dw_hidden2_output, db_output = backward(
            X_train, y_train, a_hidden1, a_hidden2, a_output, z_hidden1, z_hidden2,
            parameters["weights_hidden1_hidden2"], parameters["weights_hidden2_output"]
        )

        # Update weights and biases
        parameters["weights_input_hidden1"] -= learning_rate * dw_input_hidden1
        parameters["bias_hidden1"] -= learning_rate * db_hidden1
        parameters["weights_hidden1_hidden2"] -= learning_rate * dw_hidden1_hidden2
        parameters["bias_hidden2"] -= learning_rate * db_hidden2
        parameters["weights_hidden2_output"] -= learning_rate * dw_hidden2_output
        parameters["bias_output"] -= learning_rate * db_output

        # Print loss every 100 epochs
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Loss: {loss}')

    return parameters
```

step10: make Predictions and Evaluate the Model

Define a function to make predictions:

Use the trained network to compute probabilities for the test set and predict the class with the highest probability. Define a function to evaluate the model:

Calculate the accuracy of the predictions by comparing them with the true labels of the test set.

Implement the functions predict() and evaluation code:

```python
def predict(X, parameters):
    """Make predictions using the trained weights."""
    a_hidden1, a_hidden2, a_output, _, _, _ = forward(
        X,
        parameters["weights_input_hidden1"], parameters["bias_hidden1"],
        parameters["weights_hidden1_hidden2"], parameters["bias_hidden2"],
        parameters["weights_hidden2_output"], parameters["bias_output"]
    )
    return np.argmax(a_output, axis=1)

# Train the neural network with two hidden layers
input_size = X_train.shape[1]
hidden_size1 = 10  # Size of the first hidden layer
hidden_size2 = 10  # Size of the second hidden layer
output_size = y_train.shape[1]
learning_rate = 0.01
epochs = 1000

# Train the model
parameters = train(X_train, y_train, input_size, hidden_size1, hidden_size2, output_size, learning_rate, epochs)

# Make predictions and evaluate
y_pred = predict(X_test, parameters)
y_true = np.argmax(y_test, axis=1)
accuracy = accuracy_score(y_true, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```