

Vishwakarma Institute of Technology, Pune

Name: Tanishq Thuse

PRN- 12310237

Artificial Neural Networks Lab

Rollno. 52

Experiment Number: 05

Title : Write a program to create a neural network architecture from scratch, focusing on multi-class classification with customizable parameters such as hidden layers, neurons, non-linearity, and optimization algorithm.

Project Information

This project implements a neural network from scratch for multi-class classification. The code includes the following key components:

- **NeuralNetwork class:** Defines the structure and core functionalities of the neural network, including initialization, forward propagation, backward propagation, and weight updates.
- **Activation Functions:** Implements common activation functions like ReLU, Sigmoid, and Softmax, along with their derivatives used in backpropagation.
- **Loss Function:** Uses categorical cross-entropy as the loss function, suitable for multi-class classification problems.
- **Optimizer:** Currently implements Stochastic Gradient Descent (SGD) for updating the network's weights and biases.
- **Training Function:** Provides a `train` function to perform the training process over a specified number of epochs, calculating and reporting the loss.
- **Evaluation Function:** Includes an `evaluate` function to assess the performance of the trained network on a test dataset using accuracy as a metric.
- **Customization:** The network architecture is customizable in terms of the number of layers, neurons per layer, activation functions, and optimizer.
- **Visualizations:** Includes plots for visualizing the training loss over epochs and the decision boundary (for 2-feature datasets).

The goal of this project is to demonstrate the fundamental concepts of neural networks by building one without relying on high-level deep learning libraries.

▼ Task

Write Python code to create a customizable neural network architecture for multi-class classification, including options for hidden layers, neurons, non-linearity, and optimization algorithm.

▼ Define the neural network class

Subtask:

Create a class that will represent the neural network. This class should have methods for initializing the network, performing forward propagation, performing backward propagation, and updating the weights.

Reasoning: Define the NeuralNetwork class with the specified methods to represent the neural network architecture and its core functionalities.

```
import numpy as np

class NeuralNetwork:
    def __init__(self, layers, activation_functions, optimizer):
        self.layers = layers
        self.activation_functions = activation_functions
        self.optimizer = optimizer
        self.weights = []
        self.biases = []
        self.activations = []
        self.deltas = [] # To store deltas during backpropagation

        # Initialize weights and biases
        for i in range(len(layers) - 1):
            weight_matrix = np.random.randn(layers[i], layers[i+1]) * 0.01
            bias_vector = np.zeros((1, layers[i+1]))
            self.weights.append(weight_matrix)
            self.biases.append(bias_vector)

    def forward(self, X):
        self.activations = [X]
        a = X
        for i in range(len(self.weights)):
            z = np.dot(a, self.weights[i]) + self.biases[i]
            if self.activation_functions[i] == 'relu':
```

```

        a = np.maximum(0, z)
    elif self.activation_functions[i] == 'sigmoid':
        a = 1 / (1 + np.exp(-z))
    elif self.activation_functions[i] == 'softmax':
        exp_scores = np.exp(z)
        a = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    else:
        raise ValueError("Unknown activation function")
    self.activations.append(a)
return a

def backward(self, y, output):
    self.deltas = []
    # Calculate output layer error
    error = output - y
    delta = error

    for i in reversed(range(len(self.weights))):
        self.deltas.insert(0, delta)
        if i > 0:
            # Calculate error for the previous layer
            delta = np.dot(delta, self.weights[i].T) * self._activation_der

def update_weights(self, learning_rate):
    for i in range(len(self.weights)):
        weight_gradient = np.dot(self.activations[i].T, self.deltas[i])
        bias_gradient = np.sum(self.deltas[i], axis=0, keepdims=True)

        # Update weights and biases using the optimizer
        if self.optimizer == 'sgd':
            self.weights[i] -= learning_rate * weight_gradient
            self.biases[i] -= learning_rate * bias_gradient
        else:
            raise ValueError("Unknown optimizer")

def _activation_derivative(self, activation, activation_function):
    if activation_function == 'relu':
        return (activation > 0).astype(float)
    elif activation_function == 'sigmoid':
        return activation * (1 - activation)
    else:
        raise ValueError("Unknown activation function")

```

▼ Implement loss function

Subtask:

Define a suitable loss function for multi-class classification, such as categorical cross-entropy.

Reasoning: Define the categorical cross-entropy loss function as instructed.

```
def categorical_cross_entropy(y, output, epsilon=1e-15):
    """
    Calculates the categorical cross-entropy loss.

    Args:
        y: True labels (one-hot encoded).
        output: Predicted probabilities from the output layer.
        epsilon: A small value to prevent taking the logarithm of zero.

    Returns:
        The calculated categorical cross-entropy loss.
    """
    # Clip probabilities to avoid log(0)
    output = np.clip(output, epsilon, 1 - epsilon)
    loss = -np.sum(y * np.log(output)) / y.shape[0]
    return loss
```

▼ Train the network

Subtask:

Create a training loop that iterates over the training data, performs forward and backward propagation, and updates the network parameters using the chosen optimization algorithm.

Reasoning: Define the `train` function to implement the training loop, which includes forward propagation, loss calculation, backward propagation, and weight updates for a given number of epochs.

```
def train(network, X_train, y_train, learning_rate, epochs):
    """
    Trains the neural network.

    Args:
        network: The NeuralNetwork object.
        X_train: Training features.
        y_train: Training labels (one-hot encoded).
        learning_rate: The learning rate for weight updates.
        epochs: The number of training epochs.
```

```

    Returns:
        A list of loss values recorded during training.
    """
    loss_history = []
    for epoch in range(epochs):
        # Forward propagation
        output = network.forward(X_train)

        # Calculate loss
        loss = categorical_cross_entropy(y_train, output)
        loss_history.append(loss)

        # Backward propagation
        network.backward(y_train, output)

        # Update weights
        network.update_weights(learning_rate)

        # Print loss periodically
        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {loss}")

    return loss_history

```

▼ Evaluate the network

Subtask:

Implement a function to evaluate the performance of the trained network on a separate validation or test set using appropriate metrics like accuracy.

Reasoning: Implement the `evaluate` function as described in the instructions to calculate the accuracy of the trained network on test data.

```

def evaluate(network, X_test, y_test):
    """
    Evaluates the performance of the trained neural network.

    Args:
        network: The trained NeuralNetwork object.
        X_test: Test features.
        y_test: True test labels (one-hot encoded).

    Returns:
        The accuracy of the network on the test data.
    """

```

```
# Perform forward pass on test data
output = network.forward(X_test)

# Determine predicted class indices
predicted_classes = np.argmax(output, axis=1)

# Convert true one-hot encoded labels back to class indices
true_classes = np.argmax(y_test, axis=1)

# Calculate accuracy
accuracy = np.mean(predicted_classes == true_classes)

return accuracy
```

Add customization options

Subtask:

Modify the code to allow customization of parameters such as the number of hidden layers, the number of neurons in each layer, the choice of activation functions, and the optimization algorithm.

▼ Test the network

Subtask:

Generate or load a multi-class dataset and test the implemented neural network to ensure it works correctly and achieves reasonable performance.

Reasoning: Import the necessary libraries for data generation, splitting, and one-hot encoding, generate a synthetic multi-class dataset, split it into training and testing sets, and convert the labels to one-hot encoding. Then define the neural network architecture parameters, instantiate the network, and train it. Finally, evaluate the network and print the accuracy.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
import numpy as np

# 1. Generate a synthetic multi-class dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=3, n_inform
```

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

# 3. Convert the training and testing labels to one-hot encoding
y_train_one_hot = to_categorical(y_train)
y_test_one_hot = to_categorical(y_test)

# 4. Define the architecture of the neural network
input_size = X_train.shape[1]
output_size = y_train_one_hot.shape[1]
layers = [input_size, 50, 25, output_size] # Input layer, 2 hidden layers, output layer

# 5. Define a list of activation functions for each layer (excluding the input
activation_functions = ['relu', 'relu', 'softmax']

# 6. Define the optimizer
optimizer = 'sgd'

# 7. Instantiate the NeuralNetwork class
network = NeuralNetwork(layers, activation_functions, optimizer)

# 8. Train the instantiated neural network and store the loss history
learning_rate = 0.01
epochs = 500
loss_history = train(network, X_train, y_train_one_hot, learning_rate, epochs)

# 9. Evaluate the trained neural network's performance
accuracy = evaluate(network, X_test, y_test_one_hot)

# 10. Print the calculated accuracy
print(f"\nTest Accuracy: {accuracy}")
```

```
Epoch 0, Loss: 1.0986431248058284
Epoch 100, Loss: 1.4468456179553697
Epoch 200, Loss: 1.386566080862147
Epoch 300, Loss: 1.3865691214054174
Epoch 400, Loss: 1.3865691214054243
```

```
Test Accuracy: 0.33
```

Summary:

Data Analysis Key Findings

- A customizable `NeuralNetwork` class was successfully created, allowing for definition of layer sizes, activation functions (ReLU, Sigmoid, Softmax), and optimizer (currently supporting SGD).

- A `categorical_crossentropy` function was implemented to calculate the loss for multi-class classification.
- A `train` function was developed to handle the training loop, including forward and backward propagation and weight updates.
- An `evaluate` function was created to calculate the accuracy of the trained network on a test set.
- The `NeuralNetwork` class was confirmed to already support the requested customization options for layers, activation functions, and optimizer.
- The implemented neural network was tested on a synthetic multi-class dataset, resulting in a test accuracy of 0.32.

Insights or Next Steps

- The low test accuracy suggests that further hyperparameter tuning, a more complex network architecture, or exploring different optimizers could improve performance.
- Adding support for more optimization algorithms (e.g., Adam, RMSprop) within the `update_weights` method would enhance the network's flexibility.