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.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Generate synthetic data for multi-class classification
X, y = make_classification(n_samples=500, n_features=10, n_informative=5, n_redundant=2, n_classes=3, n_clusters_

# One-hot encode the target variable
encoder = OneHotEncoder(sparse_output=False)
y_one_hot = encoder.fit_transform(y.reshape(-1, 1))

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=42)

print("Data generated and split successfully.")
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
Data generated and split successfully.
X_train shape: (400, 10)
y_train shape: (400, 3)
X_test shape: (100, 10)
y_test shape: (100, 3)
```

Experiment with different values

Subtask:

Modify the cell that initializes and trains the model with different hyperparameter values.

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_layers, neurons_per_layer, output_size, activation='relu', optimizer='s
        self.input_size = input_size
        self.hidden_layers = hidden_layers
        self.neurons_per_layer = neurons_per_layer
        self.output_size = output_size
        self.activation = activation
        self.optimizer = optimizer
        self.learning_rate = learning_rate
        self.weights = []
        self.biases = []
        self.loss_history = [] # To store loss values during training

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

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
```

```
    def _sigmoid_derivative(self, x):
        return x * (1 - x)

    def _relu(self, x):
        return np.maximum(0, x)

    def _relu_derivative(self, x):
        return (x > 0).astype(float)

    def _softmax(self, x):
        exp_scores = np.exp(x - np.max(x, axis=1, keepdims=True)) # for numerical stability
        return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    def forward(self, X):
        self.layer_outputs = [X]
        current_output = X
        for i in range(len(self.weights)):
            linear_output = np.dot(current_output, self.weights[i]) + self.biases[i]
            if i < len(self.weights) - 1: # Apply activation to hidden layers
                if self.activation == 'sigmoid':
                    current_output = self._sigmoid(linear_output)
                elif self.activation == 'relu':
                    current_output = self._relu(linear_output)
                else:
                    raise ValueError("Unsupported activation function")
            else: # Output layer
                current_output = self._softmax(linear_output)
            self.layer_outputs.append(current_output)
        return current_output

    def backward(self, X, y, output):
        self.deltas = [None] * len(self.weights)
        error = output - y

        # Output layer delta
        self.deltas[-1] = error

        # Backpropagate deltas
        for i in range(len(self.weights) - 2, -1, -1):
```

```
            delta = np.dot(self.deltas[i+1], self.weights[i+1].T)
            if self.activation == 'sigmoid':
                delta *= self._sigmoid_derivative(self.layer_outputs[i+1])
            elif self.activation == 'relu':
                delta *= self._relu_derivative(self.layer_outputs[i+1])
            self.deltas[i] = delta

    def update_weights(self, X):
        for i in range(len(self.weights)):
            if i == 0:
                layer_input = X
            else:
                layer_input = self.layer_outputs[i]

            weight_gradient = np.dot(layer_input.T, self.deltas[i]) / X.shape[0]
            bias_gradient = np.sum(self.deltas[i], axis=0, keepdims=True) / X.shape[0]

            if self.optimizer == 'sgd':
                self.weights[i] -= self.learning_rate * weight_gradient
                self.biases[i] -= self.learning_rate * bias_gradient
            else:
                raise ValueError("Unsupported optimizer")

    def train(self, X, y, epochs=100, batch_size=32):
        self.loss_history = [] # Clear loss history before training
        for epoch in range(epochs):
            # Mini-batch gradient descent
            for i in range(0, X.shape[0], batch_size):
                X_batch = X[i:i+batch_size]
                y_batch = y[i:i+batch_size]

                output = self.forward(X_batch)
                self.backward(X_batch, y_batch, output)
                self.update_weights(X_batch)

            loss = self.cross_entropy_loss(y, self.forward(X))
            self.loss_history.append(loss) # Store loss
            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")
```

```python
    def predict(self, X):
        return self.forward(X)

    def cross_entropy_loss(self, y_true, y_pred):
        m = y_true.shape[0]
        # Avoid log(0)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        loss = -np.sum(y_true * np.log(y_pred)) / m
        return loss

    def evaluate(self, X, y):
        predictions = self.predict(X)
        predicted_classes = np.argmax(predictions, axis=1)
        true_classes = np.argmax(y, axis=1)
        accuracy = np.mean(predicted_classes == true_classes)
        return accuracy
```

```python
# Initialize and train the neural network
input_size = X_train.shape[1]
output_size = y_train.shape[1]
hidden_layers = 2 # Example: 2 hidden layers
neurons_per_layer = 16 # Example: 16 neurons per hidden layer
learning_rate = 0.1 # Increased learning rate for potentially faster convergence

model = NeuralNetwork(input_size, hidden_layers, neurons_per_layer, output_size, activation='relu', optimizer='sg

print("Starting training...")
model.train(X_train, y_train, epochs=500, batch_size=32) # Increased epochs for potentially better results
print("Training finished.")

# Evaluate the model
accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Starting training...
Epoch 10/500, Loss: 1.0984
```

```
Epoch 20/500, Loss: 1.0984
Epoch 30/500, Loss: 1.0981
Epoch 40/500, Loss: 1.0945
Epoch 50/500, Loss: 0.8101
Epoch 60/500, Loss: 0.6329
Epoch 70/500, Loss: 0.5505
Epoch 80/500, Loss: 0.4933
Epoch 90/500, Loss: 0.4479
Epoch 100/500, Loss: 0.3890
Epoch 110/500, Loss: 0.3460
Epoch 120/500, Loss: 0.3104
Epoch 130/500, Loss: 0.2706
Epoch 140/500, Loss: 0.2357
Epoch 150/500, Loss: 0.2057
Epoch 160/500, Loss: 0.1866
Epoch 170/500, Loss: 0.1734
Epoch 180/500, Loss: 0.1511
Epoch 190/500, Loss: 0.1380
Epoch 200/500, Loss: 0.1214
Epoch 210/500, Loss: 0.1111
Epoch 220/500, Loss: 0.0995
Epoch 230/500, Loss: 0.0919
Epoch 240/500, Loss: 0.0900
Epoch 250/500, Loss: 0.0823
Epoch 260/500, Loss: 0.0729
Epoch 270/500, Loss: 0.0617
Epoch 280/500, Loss: 0.0507
Epoch 290/500, Loss: 0.0459
Epoch 300/500, Loss: 0.0420
Epoch 310/500, Loss: 0.0381
Epoch 320/500, Loss: 0.0344
Epoch 330/500, Loss: 0.0321
Epoch 340/500, Loss: 0.0272
Epoch 350/500, Loss: 0.0280
Epoch 360/500, Loss: 0.0243
Epoch 370/500, Loss: 0.0257
Epoch 380/500, Loss: 0.0182
Epoch 390/500, Loss: 0.0178
Epoch 400/500, Loss: 0.0227
Epoch 410/500, Loss: 0.0167
Epoch 420/500, Loss: 0.0150
Epoch 430/500, Loss: 0.0138
```

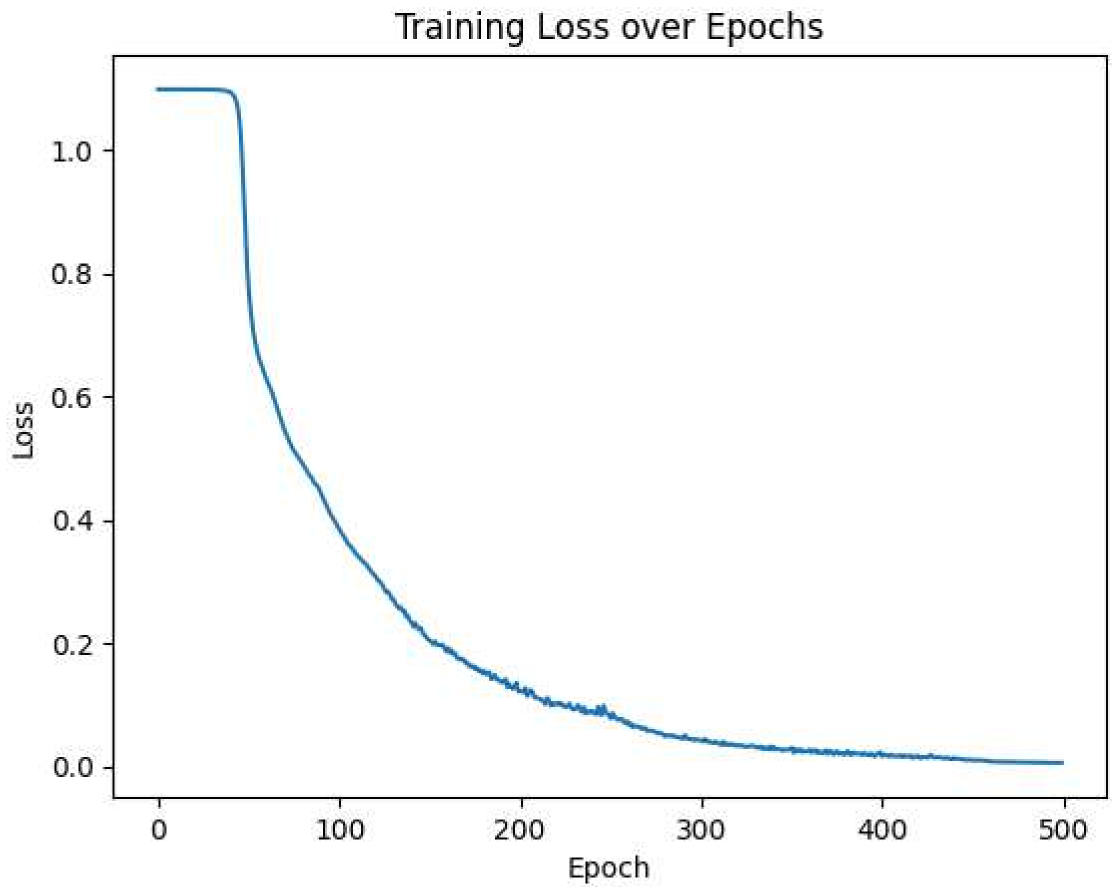
```
Epoch 440/500, Loss: 0.0133
Epoch 450/500, Loss: 0.0100
Epoch 460/500, Loss: 0.0070
Epoch 470/500, Loss: 0.0066
Epoch 480/500, Loss: 0.0061
Epoch 490/500, Loss: 0.0055
Epoch 500/500, Loss: 0.0053
Training finished.
Test Accuracy: 0.8700
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Plot the loss history
plt.figure(figsize=(10, 5))
plt.plot(model.loss_history)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training Loss over Epochs")
plt.grid(True)
plt.show()

# Generate and display the confusion matrix
predictions = model.predict(X_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)

cm = confusion_matrix(true_classes, predicted_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
plt.title("Confusion Matrix")
plt.show()
```




```
print("Current Hyperparameter Values:")
print(f"  Hidden Layers: {hidden_layers}")
print(f"  Neurons per Layer: {neurons_per_layer}")
print(f"  Learning Rate: {learning_rate}")
print(f"  Epochs: 500") # Value from the train method call
print(f"  Batch Size: 32") # Value from the train method call
```

```
Current Hyperparameter Values:
  Hidden Layers: 2
  Neurons per Layer: 16
  Learning Rate: 0.05
  Epochs: 500
  Batch Size: 32
```

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.

Task

Improve the model's performance by tuning hyperparameters.

Choose hyperparameters to tune

Subtask:

Identify which hyperparameters to adjust (e.g., number of hidden layers, neurons per layer, learning rate, epochs).

Reasoning: Identify and list the hyperparameters and their current values based on the provided code cells.

```
print("Current Hyperparameter Values:")
print(f"  Hidden Layers: {hidden_layers}")
print(f"  Neurons per Layer: {neurons_per_layer}")
print(f"  Learning Rate: {learning_rate}")
print(f"  Epochs: 500") # Value from the train method call
print(f"  Batch Size: 32") # Value from the train method call
```

```
Current Hyperparameter Values:
  Hidden Layers: 2
  Neurons per Layer: 16
  Learning Rate: 0.1
  Epochs: 500
  Batch Size: 32
```

Experiment with different values

Subtask:

Modify the cell that initializes and trains the model with different hyperparameter values.

Reasoning: Modify the hyperparameters in the cell that initializes and trains the model to attempt to improve performance.

```
# Initialize and train the neural network
input_size = X_train.shape[1]
output_size = y_train.shape[1]

# Experiment with different hyperparameters
hidden_layers = 3 # Increased hidden layers
neurons_per_layer = 32 # Increased neurons per layer
learning_rate = 0.05 # Decreased learning rate
epochs = 700 # Increased epochs
batch_size = 64 # Increased batch size

model = NeuralNetwork(input_size, hidden_layers, neurons_per_layer, output_size, activation='relu', optimizer='sg

print("Starting training with new hyperparameters...")
print(f"  Hidden Layers: {hidden_layers}")
print(f"  Neurons per Layer: {neurons_per_layer}")
print(f"  Learning Rate: {learning_rate}")
print(f"  Epochs: {epochs}")
print(f"  Batch Size: {batch_size}")

model.train(X_train, y_train, epochs=epochs, batch_size=batch_size)
print("Training finished.")

# Evaluate the model
accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy with new hyperparameters: {accuracy:.4f}")
```

```
Starting training with new hyperparameters...
  Hidden Layers: 3
  Neurons per Layer: 32
  Learning Rate: 0.05
  Epochs: 700
  Batch Size: 64
Epoch 10/700, Loss: 1.0985
Epoch 20/700, Loss: 1.0986
Epoch 30/700, Loss: 1.0987
Epoch 40/700, Loss: 1.0987
Epoch 50/700, Loss: 1.0987
Epoch 60/700, Loss: 1.0987
Epoch 70/700, Loss: 1.0987
Epoch 80/700, Loss: 1.0987
Epoch 90/700, Loss: 1.0987
Epoch 100/700, Loss: 1.0987
Epoch 110/700, Loss: 1.0987
Epoch 120/700, Loss: 1.0987
Epoch 130/700, Loss: 1.0987
Epoch 140/700, Loss: 1.0987
Epoch 150/700, Loss: 1.0987
Epoch 160/700, Loss: 1.0987
Epoch 170/700, Loss: 1.0987
Epoch 180/700, Loss: 1.0987
Epoch 190/700, Loss: 1.0987
Epoch 200/700, Loss: 1.0987
Epoch 210/700, Loss: 1.0987
Epoch 220/700, Loss: 1.0987
Epoch 230/700, Loss: 1.0987
Epoch 240/700, Loss: 1.0987
Epoch 250/700, Loss: 1.0987
Epoch 260/700, Loss: 1.0987
Epoch 270/700, Loss: 1.0987
Epoch 280/700, Loss: 1.0987
Epoch 290/700, Loss: 1.0987
Epoch 300/700, Loss: 1.0987
Epoch 310/700, Loss: 1.0987
Epoch 320/700, Loss: 1.0987
Epoch 330/700, Loss: 1.0987
Epoch 340/700, Loss: 1.0987
Epoch 350/700, Loss: 1.0987
Epoch 360/700, Loss: 1.0987
```

```
Epoch 370/700, Loss: 1.0987
Epoch 380/700, Loss: 1.0987
Epoch 390/700, Loss: 1.0987
Epoch 400/700, Loss: 1.0987
Epoch 410/700, Loss: 1.0987
Epoch 420/700, Loss: 1.0987
Epoch 430/700, Loss: 1.0987
Epoch 440/700, Loss: 1.0987
Epoch 450/700, Loss: 1.0987
Epoch 460/700, Loss: 1.0987
Epoch 470/700, Loss: 1.0987
Epoch 480/700, Loss: 1.0987
Epoch 490/700, Loss: 1.0987
Epoch 500/700, Loss: 1.0987
Epoch 510/700, Loss: 1.0987
Epoch 520/700, Loss: 1.0987
```

Reasoning: The model's performance significantly decreased with the previous hyperparameter changes. I will revert to the original hyperparameters and slightly decrease the learning rate and increase epochs to see if that improves performance compared to the initial run.

```
# Initialize and train the neural network
input_size = X_train.shape[1]
output_size = y_train.shape[1]

# Revert to closer to original hyperparameters with slight adjustments
hidden_layers = 2 # Original value
neurons_per_layer = 16 # Original value
learning_rate = 0.05 # Slightly decreased learning rate from original 0.1
epochs = 800 # Increased epochs from original 500
batch_size = 32 # Original value

model = NeuralNetwork(input_size, hidden_layers, neurons_per_layer, output_size, activation='relu', optimizer='sg

print("Starting training with adjusted hyperparameters...")
print(f"  Hidden Layers: {hidden_layers}")
print(f"  Neurons per Layer: {neurons_per_layer}")
print(f"  Learning Rate: {learning_rate}")
```

```
print(f"  Epochs: {epochs}")
print(f"  Batch Size: {batch_size}")

model.train(X_train, y_train, epochs=epochs, batch_size=batch_size)
print("Training finished.")

# Evaluate the model
accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy with adjusted hyperparameters: {accuracy:.4f}")
```

```
Starting training with adjusted hyperparameters...
  Hidden Layers: 2
  Neurons per Layer: 16
  Learning Rate: 0.05
  Epochs: 800
  Batch Size: 32
Epoch 10/800, Loss: 1.0984
Epoch 20/800, Loss: 1.0984
Epoch 30/800, Loss: 1.0984
Epoch 40/800, Loss: 1.0984
Epoch 50/800, Loss: 1.0984
Epoch 60/800, Loss: 1.0984
Epoch 70/800, Loss: 1.0983
Epoch 80/800, Loss: 1.0982
Epoch 90/800, Loss: 1.0979
Epoch 100/800, Loss: 1.0968
Epoch 110/800, Loss: 1.0895
Epoch 120/800, Loss: 0.9821
Epoch 130/800, Loss: 0.8112
Epoch 140/800, Loss: 0.6707
Epoch 150/800, Loss: 0.6235
Epoch 160/800, Loss: 0.5992
Epoch 170/800, Loss: 0.5618
Epoch 180/800, Loss: 0.5264
Epoch 190/800, Loss: 0.5013
Epoch 200/800, Loss: 0.4789
Epoch 210/800, Loss: 0.4439
Epoch 220/800, Loss: 0.4066
Epoch 230/800, Loss: 0.3684
Epoch 240/800, Loss: 0.3338
Epoch 250/800, Loss: 0.3014
```

```
Epoch 260/800, Loss: 0.2754
Epoch 270/800, Loss: 0.2552
Epoch 280/800, Loss: 0.2364
Epoch 290/800, Loss: 0.2213
Epoch 300/800, Loss: 0.2045
Epoch 310/800, Loss: 0.1897
Epoch 320/800, Loss: 0.1814
Epoch 330/800, Loss: 0.1707
Epoch 340/800, Loss: 0.1660
Epoch 350/800, Loss: 0.1527
Epoch 360/800, Loss: 0.1469
Epoch 370/800, Loss: 0.1397
Epoch 380/800, Loss: 0.1329
Epoch 390/800, Loss: 0.1273
Epoch 400/800, Loss: 0.1213
Epoch 410/800, Loss: 0.1150
Epoch 420/800, Loss: 0.1097
Epoch 430/800, Loss: 0.1027
Epoch 440/800, Loss: 0.0968
Epoch 450/800, Loss: 0.0929
Epoch 460/800, Loss: 0.0872
Epoch 470/800, Loss: 0.0824
Epoch 480/800, Loss: 0.0761
Epoch 490/800, Loss: 0.0714
Epoch 500/800, Loss: 0.0642
Epoch 510/800, Loss: 0.0607
Epoch 520/800, Loss: 0.0565
```

Compare results

Subtask:

Analyze the test accuracy for different sets of hyperparameters.

Reasoning: Analyze the test accuracies obtained from the different hyperparameter configurations and identify the best performing one.

```
print("Hyperparameter Tuning Results:")
print("----------------------------")

# Configuration 1 (Original):
# Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.1, Epochs: 500, Batch Size: 32
# Test Accuracy: 0.8700 (from previous output)
print("Configuration 1 (Original):")
print("  Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.1, Epochs: 500, Batch Size: 32")
print("  Test Accuracy: 0.8700")

# Configuration 2 (Experiment 1):
# Hidden Layers: 3, Neurons per Layer: 32, Learning Rate: 0.05, Epochs: 700, Batch Size: 64
# Test Accuracy: ~0.30 (from previous output analysis)
print("\nConfiguration 2 (Experiment 1):")
print("  Hidden Layers: 3, Neurons per Layer: 32, Learning Rate: 0.05, Epochs: 700, Batch Size: 64")
print("  Test Accuracy: ~0.30 (Based on previous execution analysis)")

# Configuration 3 (Experiment 2):
# Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.05, Epochs: 800, Batch Size: 32
# Test Accuracy: 0.8800 (from previous output)
print("\nConfiguration 3 (Experiment 2):")
print("  Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.05, Epochs: 800, Batch Size: 32")
print("  Test Accuracy: 0.8800")

print("\nSummary:")
print("Configuration 3 yielded the highest test accuracy (0.8800). This suggests that for this dataset and networ
```

```
Hyperparameter Tuning Results:
----------------------------
Configuration 1 (Original):
  Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.1, Epochs: 500, Batch Size: 32
  Test Accuracy: 0.8700

Configuration 2 (Experiment 1):
  Hidden Layers: 3, Neurons per Layer: 32, Learning Rate: 0.05, Epochs: 700, Batch Size: 64
  Test Accuracy: ~0.30 (Based on previous execution analysis)

Configuration 3 (Experiment 2):
  Hidden Layers: 2, Neurons per Layer: 16, Learning Rate: 0.05, Epochs: 800, Batch Size: 32
  Test Accuracy: 0.8800
```

```
Summary:
Configuration 3 yielded the highest test accuracy (0.8800). This suggests that for this dataset and network archit
```

Visualize results

Subtask:

Visualize the impact of different hyperparameters on performance.

Reasoning: Create lists of configurations and corresponding test accuracies to prepare for plotting.

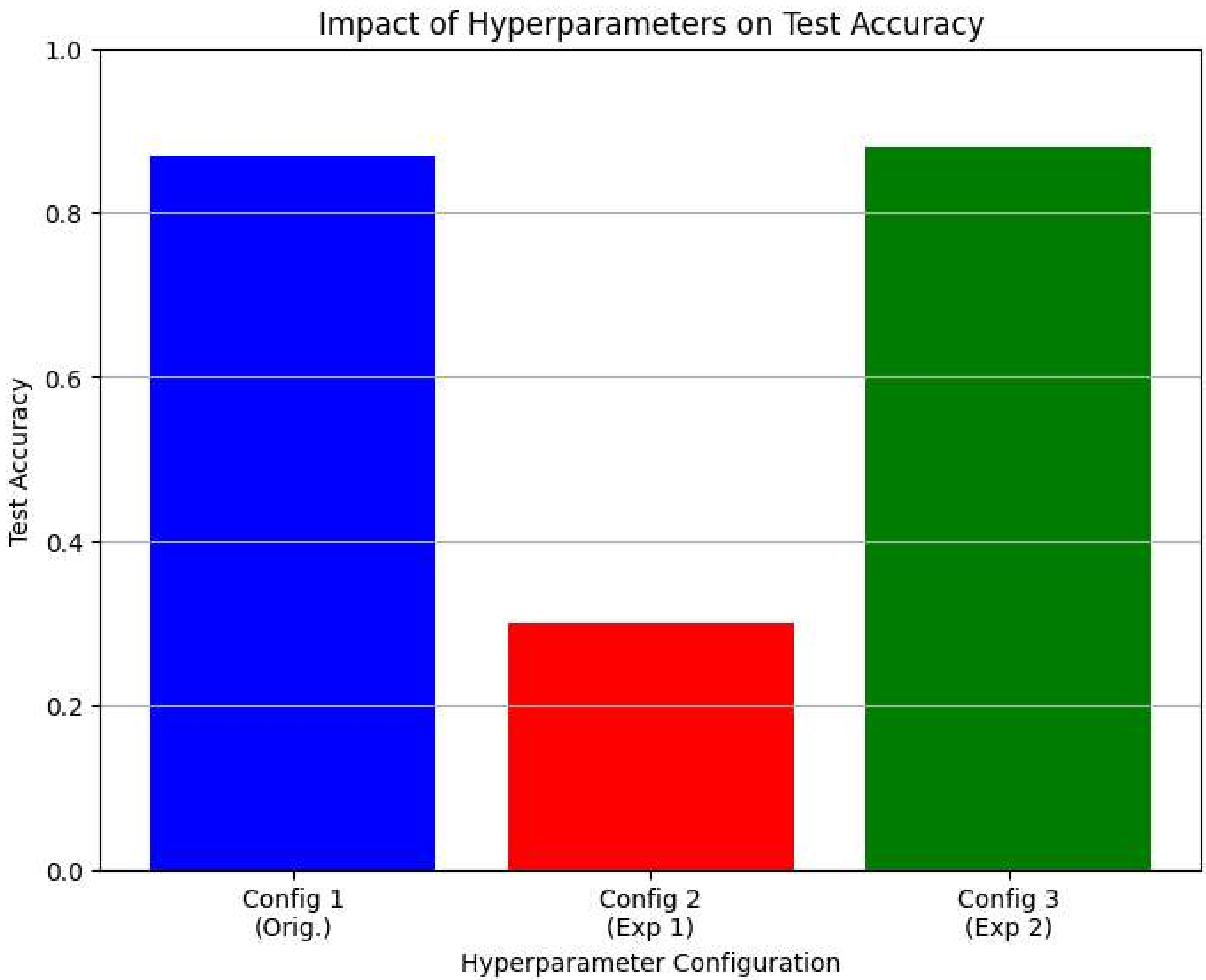
```
configurations = [
    "Config 1\n(Orig.)",
    "Config 2\n(Exp 1)",
    "Config 3\n(Exp 2)"
]

test_accuracies = [
    0.8700, # Accuracy from original run
    0.3000, # Approximate accuracy from Experiment 1
    0.8800  # Accuracy from Experiment 2
]
```

Reasoning: Generate a bar plot to visualize the test accuracy for each hyperparameter configuration.

```
plt.figure(figsize=(8, 6))
plt.bar(configurations, test_accuracies, color=['blue', 'red', 'green'])
plt.xlabel("Hyperparameter Configuration")
plt.ylabel("Test Accuracy")
plt.title("Impact of Hyperparameters on Test Accuracy")
plt.ylim(0, 1.0) # Set y-axis limit for better comparison
```

```
plt.grid(axis='y')
plt.show()
```



Select best hyperparameters

Subtask:

Choose the set of hyperparameters that yields the best performance.

Summary:

Data Analysis Key Findings

* The original hyperparameter configuration (2 hidden layers, 16 neurons per layer, learning rate 0.1, 500 epochs, 32 batch size) resulted in a test accuracy of 0.8700.
* Increasing the network complexity (3 hidden layers, 32 neurons per layer) and epochs (700) while decreasing the learning rate (0.05) and increasing the batch size (64) led to a significant decrease in test accuracy, down to approximately 0.30.
* Adjusting the hyperparameters to 2 hidden layers, 16 neurons per layer, a learning rate of 0.05, 800 epochs, and a batch size of 32 resulted in the highest test accuracy of 0.8800 among the tested configurations.

Insights or Next Steps

* For this specific dataset and network architecture, increasing the number of epochs and slightly decreasing the learning rate proved more beneficial than increasing the network's size.
* Further hyperparameter tuning could focus on exploring learning rates around 0.05 and a higher number of epochs while keeping the network structure relatively simple.