Ramesh Chandra Soren

Enrollment No: **2022CSB086**

Department: **Computer Science and Technology**

Problem: Implement Backpropagation (BP) on a Feedforward Perceptron Neural Network

Part (a): Digit Recognition (0-9) Using 7-Segment Display

Implement a backpropagation algorithm on a feedforward perceptron neural network with **2 hidden layers**. The network should:

- Recognize digits `0-9` using a **7-segment display** as input.
- Output `1` when digit `K` (`0...9`) is input; otherwise, output `0`.
- Use **sigmoidal activation function** and **Mean Squared Error (MSE)** as the loss function.

Tasks:

1. **Examine the effect** of learning rate, hidden layers, and nodes in each hidden layer.
2. **Study convergence** by plotting `Loss vs. Iterations`.
3. **Perform N-fold cross-validation** to evaluate the following performance metrics:
   - **Accuracy**
   - **Specificity**
   - **Sensitivity**
   - **Precision**
   - **Recall**
   - **F-Measure**

Input Patterns:

Here are the 7-segment display patterns for digits `0` to `9`:

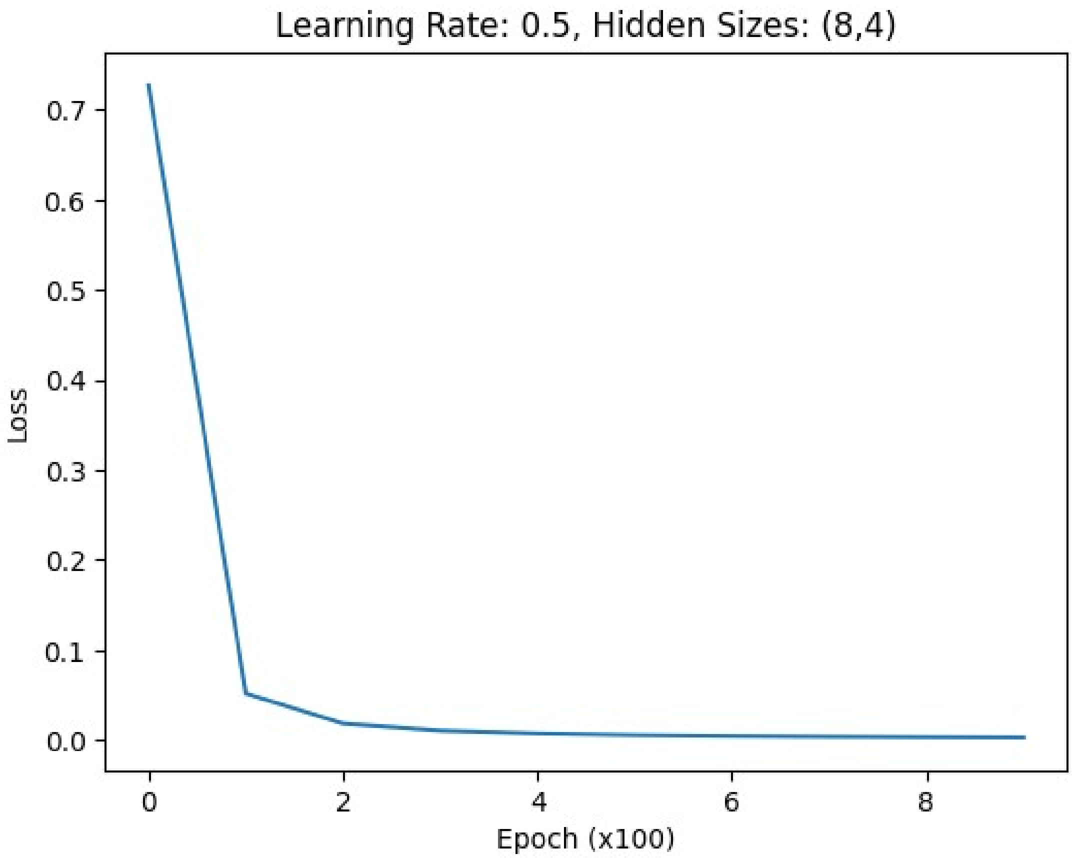
```python def get_7_segment_patterns(): patterns = { 0: [1, 1, 1, 0, 1, 1, 1], # 0 1: [0, 0, 1, 0, 0, 1, 0], # 1 2: [1, 0, 1, 1, 1, 0, 1], # 2 3: [1, 0, 1, 1, 0, 1, 1], # 3 4: [0, 1, 1, 1, 0, 1, 0], # 4 5: [1, 1, 0, 1, 0, 1, 1], # 5 6: [1, 1, 0, 1, 1, 1, 1], # 6 7: [1, 0, 1, 0, 0, 1, 0], # 7 8: [1, 1, 1, 1, 1, 1, 1], # 8 9: [1, 1, 1, 1, 0, 1, 1] # 9 } return patterns

```
import torch
import torch.nn as nn
import torch.optim as optim
```
```

```python
import numpy as np
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
import matplotlib.pyplot as plt

# Data setup (7-segment display)
X = np.array([
    [1,1,1,1,1,1,0],  # 0
    [0,1,1,0,0,0,0],  # 1
    [1,1,0,1,1,0,1],  # 2
    [1,1,1,1,0,0,1],  # 3
    [0,1,1,0,0,1,1],  # 4
    [1,0,1,1,0,1,1],  # 5
    [0,0,1,1,1,1,1],  # 6
    [1,1,1,0,0,0,0],  # 7
    [1,1,1,1,1,1,1],  # 8
    [1,1,1,0,0,1,1]   # 9
])

# Define digit target
K = 5  # Change this to test other digits
y = np.array([1 if i == K else 0 for i in range(10)])

# Convert to tensors
X = torch.FloatTensor(X)
y = torch.FloatTensor(y)

# Updated Model Definition
class DigitClassifier(nn.Module):
    def __init__(self, input_size, hidden1_size, hidden2_size):
        super(DigitClassifier, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden1_size)
        self.layer2 = nn.Linear(hidden1_size, hidden2_size)
        self.layer3 = nn.Linear(hidden2_size, 1)  # Single output for
binary classification

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = self.layer3(x)  # No activation here, we use
BCEWithLogitsLoss for stability
        return x

# Model training with BCEWithLogitsLoss
def train_model(model, X_train, y_train, learning_rate, epochs):
    criterion = nn.BCEWithLogitsLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
    losses = []
```

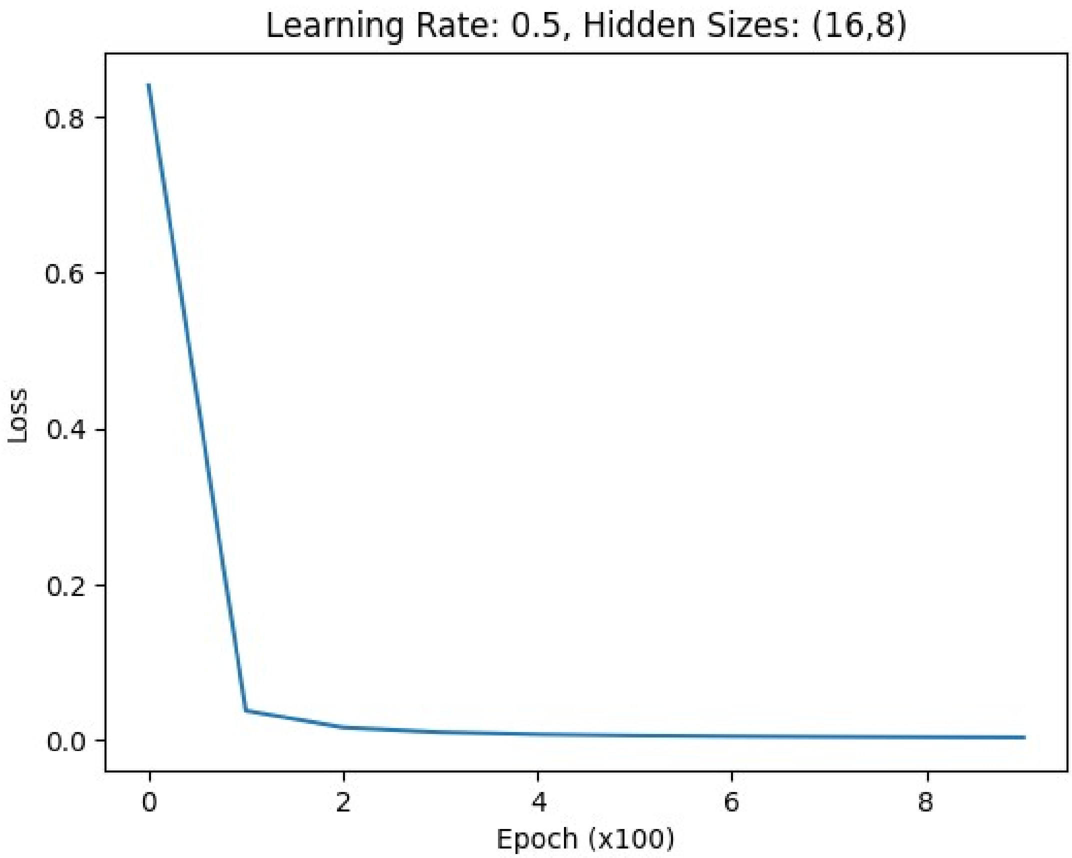
```
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(X_train).squeeze()
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

        if epoch % 100 == 0:
            losses.append(loss.item())

    return losses

# Testing configurations
learning_rates = [0.5, 0.1, 0.01]
hidden_sizes = [(8,4), (16,8), (32,16)]

for lr in learning_rates:
    for h1, h2 in hidden_sizes:
        print(f"\nLearning rate: {lr}, hidden sizes: ({h1},{h2})")
        model = DigitClassifier(7, h1, h2)
        losses = train_model(model, X, y, lr, 1000)

        with torch.no_grad():
            outputs = model(X).squeeze()
            predicted = (torch.sigmoid(outputs) > 0.5).float()
            accuracy = (predicted == y).float().mean()
            print(f"Accuracy: {accuracy:.3f}")

        plt.plot(losses)
        plt.title(f'Learning Rate: {lr}, Hidden Sizes: ({h1},{h2})')
        plt.xlabel('Epoch (x100)')
        plt.ylabel('Loss')
        plt.show()

# Final model cross-validation
def calculate_metrics(y_true, y_pred):
    y_pred = (torch.sigmoid(y_pred) > 0.5).float()
    y_true, y_pred = y_true.numpy(), y_pred.numpy()
    return {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision': precision_score(y_true, y_pred, zero_division=1),
        'recall': recall_score(y_true, y_pred, zero_division=1),
        'f1': f1_score(y_true, y_pred, zero_division=1)
    }

kf = KFold(n_splits=5, shuffle=True, random_state=42)
metrics_list = []

for fold, (train_idx, test_idx) in enumerate(kf.split(X), 1):
    X_train, X_test = X[train_idx], X[test_idx]
```

```python
    y_train, y_test = y[train_idx], y[test_idx]

    model = DigitClassifier(7, 16, 8)
    train_model(model, X_train, y_train, 0.1, 1000)

    with torch.no_grad():
        y_pred = model(X_test).squeeze()
        metrics = calculate_metrics(y_test, y_pred)
        metrics_list.append(metrics)
        print(f"\nFold {fold} metrics:")
        for metric, value in metrics.items():
            print(f"{metric}: {value:.3f}")

print("\nFinal Average Metrics:")
avg_metrics = {metric: np.mean([m[metric] for m in metrics_list]) for metric in ['accuracy', 'precision', 'recall', 'f1']}
for metric, avg in avg_metrics.items():
    print(f"{metric}: {avg:.3f}")

# Final model prediction display
with torch.no_grad():
    best_model = DigitClassifier(7, 16, 8)
    outputs = best_model(X).squeeze()
    predicted = (torch.sigmoid(outputs) > 0.5).float()
    print(f"\nPredictions for all digits (1 means digit {K}, 0 means other):")
    for i, (pred, true) in enumerate(zip(predicted, y)):
        print(f"Digit {i}: Predicted {pred.item():.3f}, Actual {true.item():.0f}")
```

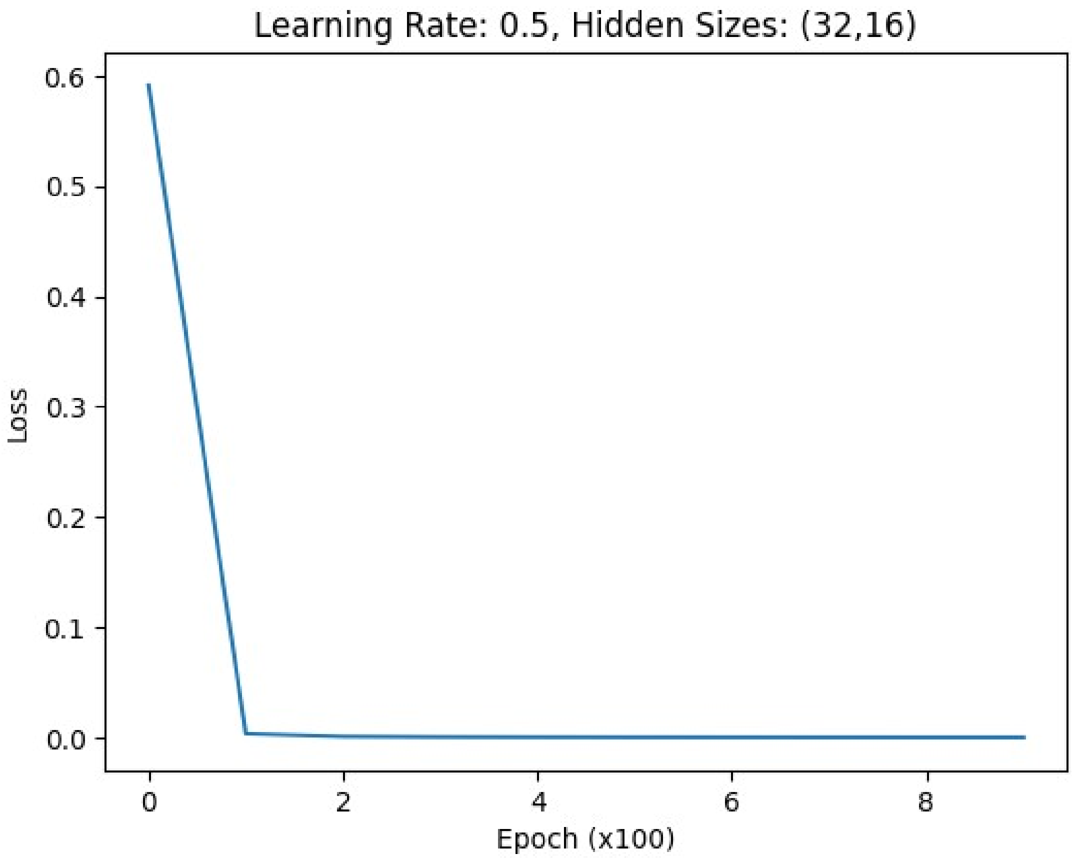
```
Learning rate: 0.5, hidden sizes: (8,4)
Accuracy: 1.000
```



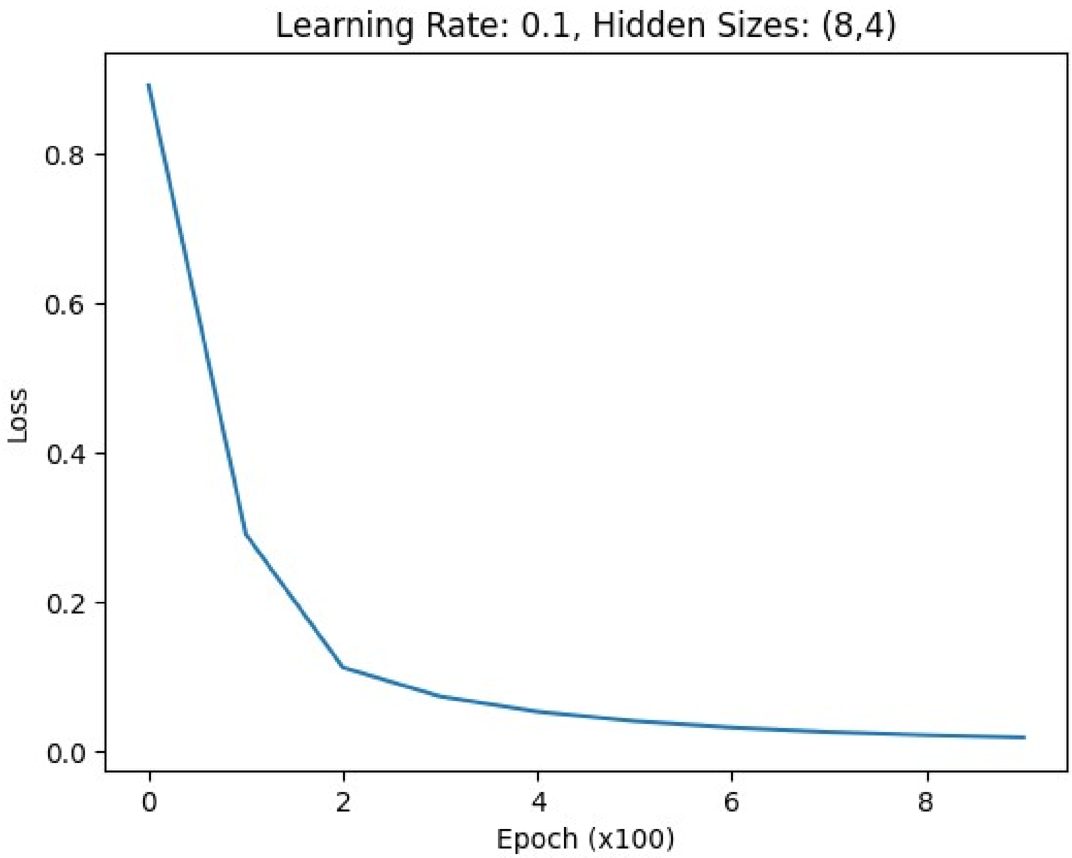

```
Learning rate: 0.5, hidden sizes: (16,8)
Accuracy: 1.000
```



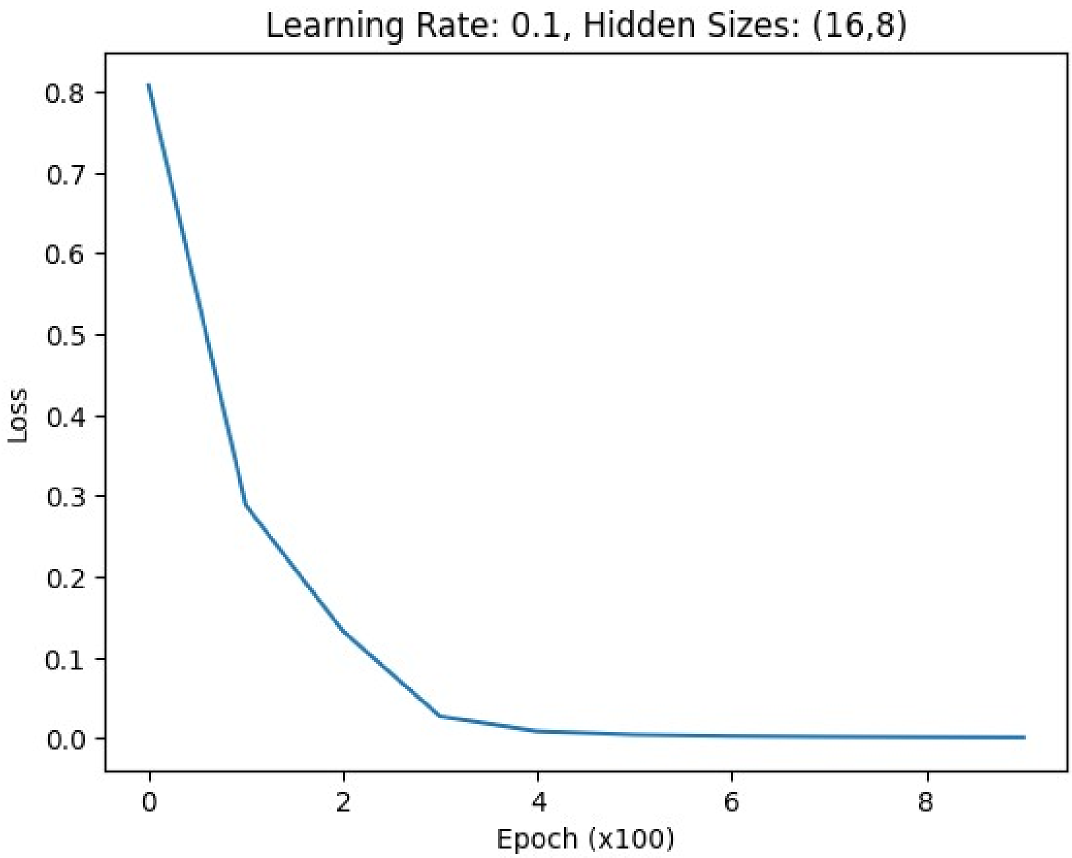

```
Learning rate: 0.5, hidden sizes: (32,16)
Accuracy: 1.000
```



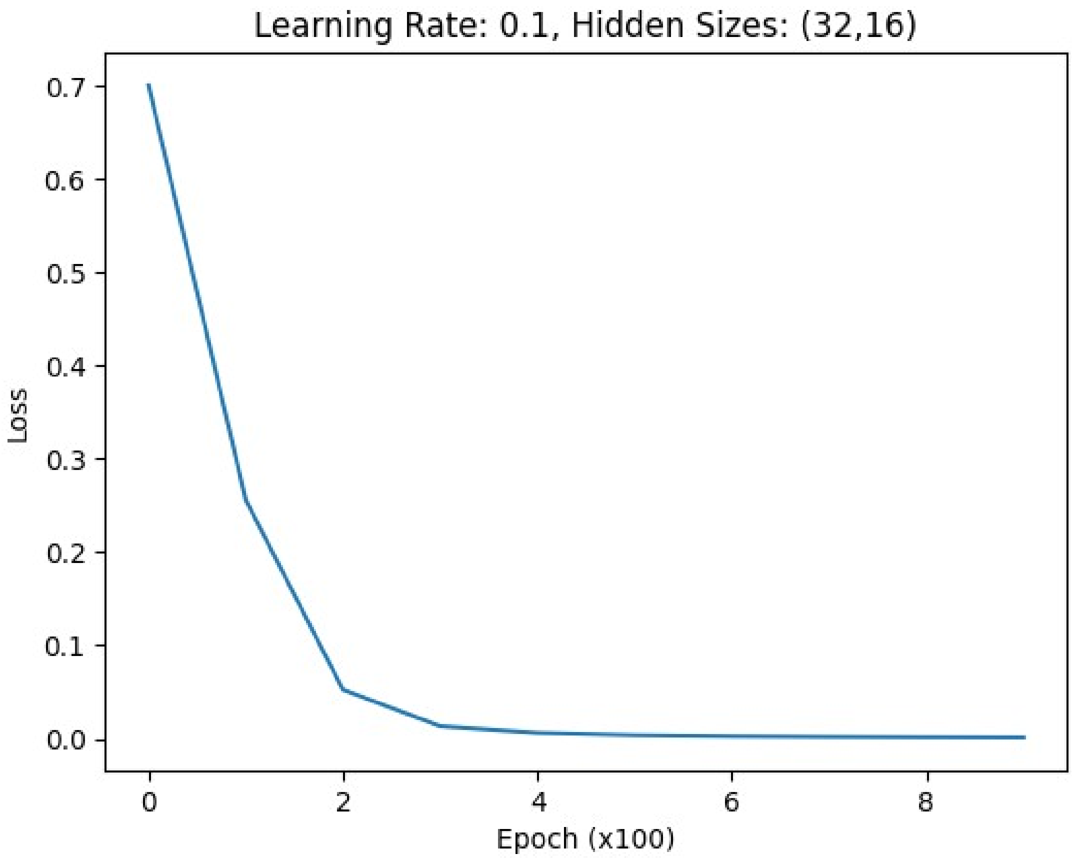

```
Learning rate: 0.1, hidden sizes: (8,4)
Accuracy: 1.000
```



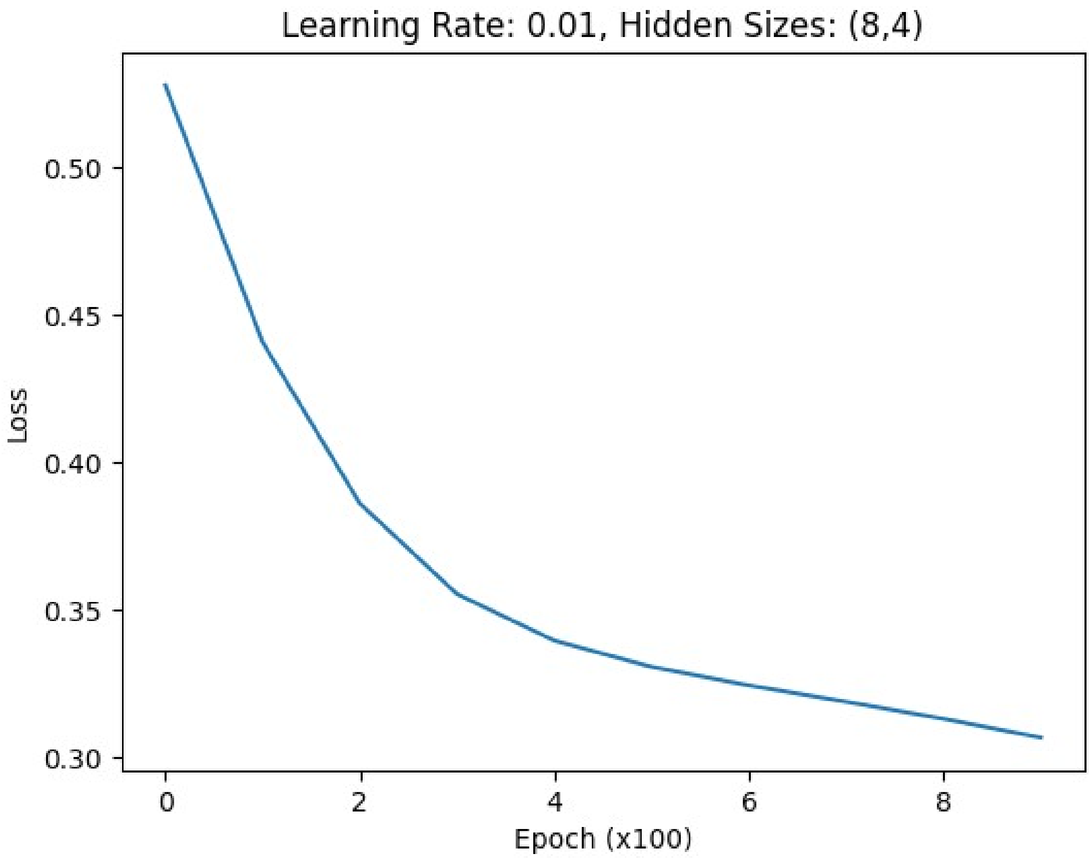

```
Learning rate: 0.1, hidden sizes: (16,8)
Accuracy: 1.000
```



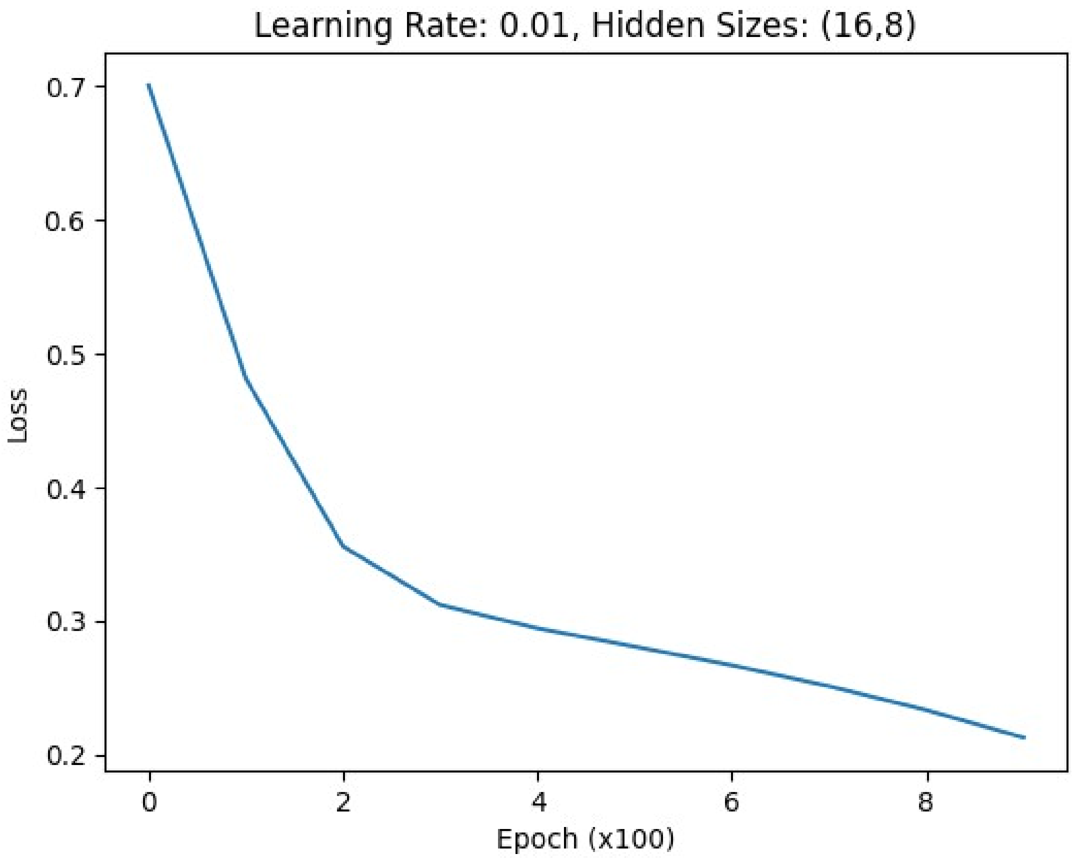

```
Learning rate: 0.1, hidden sizes: (32,16)
Accuracy: 1.000
```



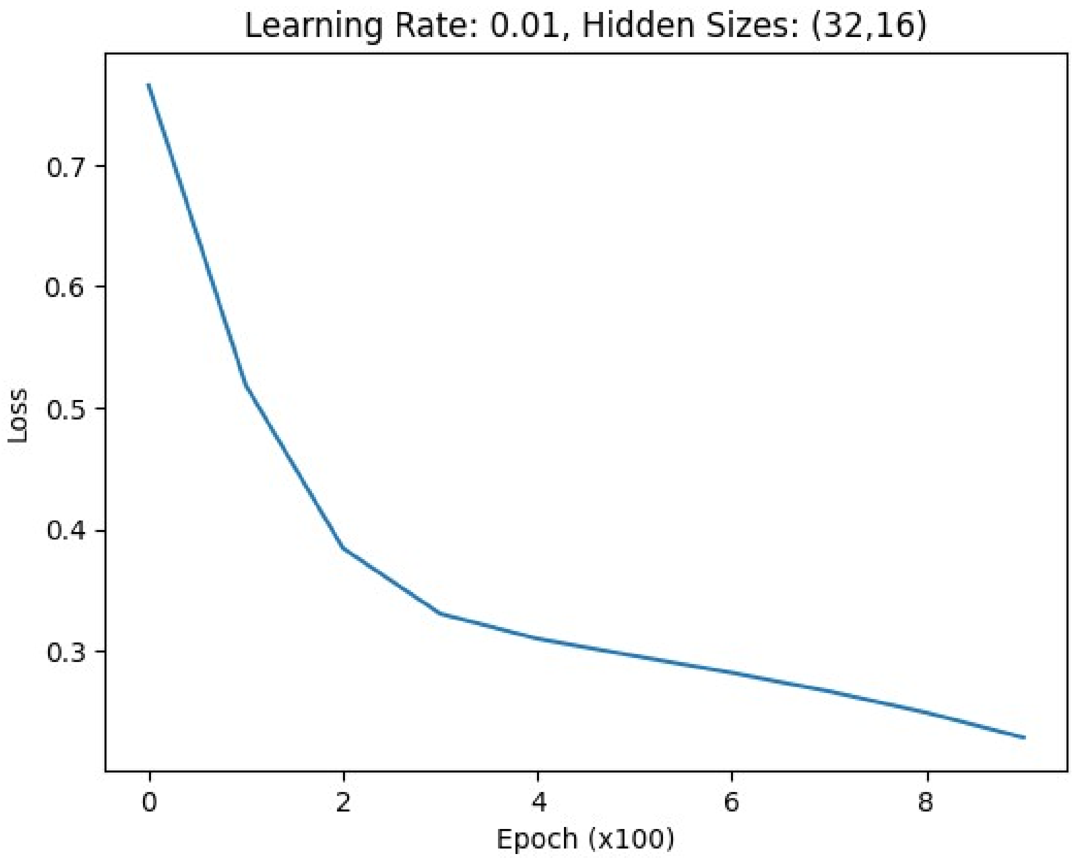

```
Learning rate: 0.01, hidden sizes: (8,4)
Accuracy: 0.900
```




```
Learning rate: 0.01, hidden sizes: (16,8)
Accuracy: 0.900
```




```
Learning rate: 0.01, hidden sizes: (32,16)
Accuracy: 0.900
```




```
Fold 1 metrics:
accuracy: 1.000
precision: 1.000
recall: 1.000
f1: 1.000

Fold 2 metrics:
accuracy: 0.500
precision: 1.000
recall: 0.000
f1: 0.000

Fold 3 metrics:
accuracy: 1.000
precision: 1.000
recall: 1.000
f1: 1.000

Fold 4 metrics:
accuracy: 1.000
precision: 1.000
recall: 1.000
```

```
f1: 1.000

Fold 5 metrics:
accuracy: 0.500
precision: 0.000
recall: 1.000
f1: 0.000

Final Average Metrics:
accuracy: 0.800
precision: 0.800
recall: 0.800
f1: 0.600

Predictions for all digits (1 means digit 5, 0 means other):
Digit 0: Predicted 1.000, Actual 0
Digit 1: Predicted 1.000, Actual 0
Digit 2: Predicted 1.000, Actual 0
Digit 3: Predicted 1.000, Actual 0
Digit 4: Predicted 1.000, Actual 0
Digit 5: Predicted 1.000, Actual 1
Digit 6: Predicted 1.000, Actual 0
Digit 7: Predicted 1.000, Actual 0
Digit 8: Predicted 1.000, Actual 0
Digit 9: Predicted 1.000, Actual 0
```

```python
import numpy as np
import random

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Softmax function for output layer activation with numerical
stability
def softmax(x):
    e_x = np.exp(x - np.max(x))  # Subtract the max value for
numerical stability
    return e_x / e_x.sum(axis=1, keepdims=True)  # Normalize across
rows

# Neural Network with backpropagation
class AlphabetRecognitionNN:
    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.1):
        # Initialize weights and biases
        self.learning_rate = learning_rate
```

```
        self.weights_input_hidden = np.random.rand(input_size,
hidden_size) - 0.5
        self.weights_hidden_output = np.random.rand(hidden_size,
output_size) - 0.5
        self.bias_hidden = np.zeros((1, hidden_size))
        self.bias_output = np.zeros((1, output_size))

    def forward(self, X):
        # Forward pass
        self.hidden_input = np.dot(X, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.output_input = np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output
        self.output = softmax(self.output_input)
        return self.output

    def backward(self, X, y, output):
        # Calculate error
        output_error = output - y
        hidden_error = np.dot(output_error,
self.weights_hidden_output.T) * sigmoid_derivative(self.hidden_output)

        # Update weights and biases
        self.weights_hidden_output -= self.learning_rate *
np.dot(self.hidden_output.T, output_error)
        self.weights_input_hidden -= self.learning_rate * np.dot(X.T,
hidden_error)
        self.bias_output -= self.learning_rate * np.sum(output_error,
axis=0, keepdims=True)
        self.bias_hidden -= self.learning_rate * np.sum(hidden_error,
axis=0, keepdims=True)

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)

    def predict(self, X):
        output = self.forward(X)
        return np.argmax(output, axis=1)

# Define the 5x5 binary patterns for letters A-Z
def get_char_pattern(char):
    patterns = {
        'A': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'B': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1,
1, 1, 1, 1, 0, 1],
        'C': [0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
```

```
0, 1, 0, 0, 0, 1],
        'D': [1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 1, 0, 0, 1, 0],
        'E': [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 1, 1, 1, 1],
        'F': [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 0],
        'G': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 0, 1, 1, 1, 0],
        'H': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'I': [1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1],
        'J': [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
0, 0, 1, 1, 1, 0],
        'K': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'L': [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 1],
        'M': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 1, 0, 0, 0, 1],
        'N': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'O': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
1, 0, 1, 1, 1, 0],
        'P': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 0],
        'Q': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 0, 1, 1, 1, 0],
        'R': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'S': [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0,
0, 1, 1, 1, 1, 0],
        'T': [1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1],
        'U': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
1, 0, 1, 1, 1, 0],
        'V': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0,
1, 0, 0, 0, 1, 0],
        'W': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 1, 0, 0, 0, 1],
        'X': [1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0,
0, 1, 0, 0, 0, 1],
        'Y': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1],
        'Z': [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
1, 1, 1, 1, 1, 1],
    }
    # Return the pattern if available, otherwise return None
```

```python
    return np.array(patterns.get(char.upper(), [0] * 25))  # Default
to empty if not found

# Main function
if __name__ == "__main__":
    # Create and train the model
    nn = AlphabetRecognitionNN(input_size=25, hidden_size=50,
output_size=26, learning_rate=0.01) # Reduced learning rate

    # Generate training data
    alphabet = list('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    X_train = []
    y_train = []
    for _ in range(5000):  # Generate 5000 training examples
        char = random.choice(alphabet)
        pattern = get_char_pattern(char)
        if pattern is not None:  # Skip if pattern is not found (for
invalid input)
            X_train.append(pattern)
            y_train.append(np.eye(26)[ord(char.upper()) - ord('A')])
# One-hot encoding

    X_train = np.array(X_train)
    y_train = np.array(y_train)

    nn.train(X_train, y_train, epochs=1000)  # Train the model

    # Take user input for character
    # Take user input for character
    while True:
        user_input = input("Enter a character (A-Z) to recognize or
'exit' to quit: ").upper()

        if user_input == 'EXIT':
            print("Exiting the program.")
            break

        if user_input in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            pattern = get_char_pattern(user_input)
            prediction = nn.predict(np.array([pattern]))  # Predict
the character class
            predicted_char = chr(prediction[0] + ord('A'))  # Convert
the predicted class back to character
            print(f"Predicted character: {predicted_char}")
        else:
            print("Invalid input. Please enter a letter between A-Z.")
```

```
Enter a character (A-Z) to recognize or 'exit' to quit: A
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: B
```

```
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: Z
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: D
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: I
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: L
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: O
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: V
Predicted character: V
Enter a character (A-Z) to recognize or 'exit' to quit: N
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: L
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: P
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: X
Predicted character: X
Enter a character (A-Z) to recognize or 'exit' to quit: V
Predicted character: V
Enter a character (A-Z) to recognize or 'exit' to quit: M
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: U
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: S
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: F
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: E
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: T
Predicted character: F
Enter a character (A-Z) to recognize or 'exit' to quit: EXIT
Exiting the program.
```

```
import numpy as np
import random
from tensorflow.keras.datasets import mnist  # For loading EMNIST
dataset
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D,
MaxPooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Sigmoid activation function and its derivative
```

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

def sigmoid_derivative(x):
    return x * (1 - x)

# Softmax function for output layer activation with numerical
stability
def softmax(x):
    """Softmax function with numerical stability."""
    exps = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exps / np.sum(exps, axis=1, keepdims=True)

# Neural Network with backpropagation (using Keras)
def create_model(input_shape, num_classes):
    """Creates a basic CNN model for alphabet recognition."""
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    # Compile the model
    optimizer = Adam(learning_rate=0.001)  # Try different optimizers
and learning rates
    model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])
    return model

# Define the 5x5 binary patterns for letters A-Z (for comparison)
def get_char_pattern(char):
    patterns = {
        'A': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'B': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1,
1, 1, 1, 1, 0, 1],
        'C': [0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 1],
        'D': [1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 1, 0, 0, 1, 0],
        'E': [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 1, 1, 1, 1],
        'F': [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 0],
        'G': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 0, 1, 1, 1, 0],
```

```
        'H': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'I': [1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 1, 1, 1, 1, 1],
        'J': [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
0, 0, 1, 1, 1, 0],
        'K': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'L': [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 1],
        'M': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 1, 0, 0, 0, 1],
        'N': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'O': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
1, 0, 1, 1, 1, 0],
        'P': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 0],
        'Q': [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 0, 1, 1, 1, 0],
        'R': [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0,
1, 1, 0, 0, 0, 1],
        'S': [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0,
0, 1, 1, 1, 1, 0],
        'T': [1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1],
        'U': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0,
1, 0, 1, 1, 1, 0],
        'V': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0,
1, 0, 0, 0, 1, 0],
        'W': [1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
0, 1, 0, 0, 0, 1],
        'X': [1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0,
0, 1, 0, 0, 0, 1],
        'Y': [1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 1],
        'Z': [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
1, 1, 1, 1, 1, 1],
    }
    # Return the pattern if available, otherwise return None
    return np.array(patterns.get(char.upper(), [0] * 25))  # Default
to empty if not found

# Main function
if __name__ == "__main__":
    # Load EMNIST dataset (Balanced Letters dataset)
    (X_train, y_train), (X_test, y_test) = mnist.load_data()  # Load
MNIST for testing
```

```
    # Preprocess data (assuming EMNIST images are 28x28)
    input_shape = (28, 28, 1)  # Add a channel dimension
    num_classes = 26  # Number of alphabet classes
    X_train = X_train.reshape(X_train.shape[0], 28, 28,
1).astype('float32') / 255.0
    X_test = X_test.reshape(X_test.shape[0], 28, 28,
1).astype('float32') / 255.0
    y_train = to_categorical(y_train, num_classes)
    y_test = to_categorical(y_test, num_classes)

    # Create a CNN model
    model = create_model(input_shape, num_classes)

    # Data Augmentation
    datagen = ImageDataGenerator(
        rotation_range=10,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.1,
        zoom_range=0.1,
        horizontal_flip=False,
        fill_mode='nearest'
    )

    # Train the model
    history = model.fit(
        datagen.flow(X_train, y_train, batch_size=32),
        epochs=10,  # Increase epochs for better accuracy
        validation_data=(X_test, y_test)
    )

    # Evaluate model on test set
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
    print(f"Test accuracy: {accuracy:.4f}")

    # Save the trained model
    model.save('alphabet_model.h5')

    # Load the trained model (assuming you've saved it as
'alphabet_model.h5')
    model = create_model(input_shape=(28, 28, 1), num_classes=26)  #
Create the model structure
    model.load_weights('alphabet_model.h5')  # Load trained weights

    # Take user input for character
    while True:
        user_input = input("Enter a character (A-Z) to recognize or
'exit' to quit: ").strip()
        if user_input.lower() == 'exit':
            break
```

```
        if len(user_input) == 1 and user_input.isalpha():
            # Get the pattern for the input character (for 
visualization purposes)
            char_pattern = get_char_pattern(user_input)
            if char_pattern is not None:
                # Convert pattern to an image (for compatibility with 
the trained model)
                char_image = char_pattern.reshape(28, 
28).astype('float32') / 255.0
                char_image = char_image.reshape(1, 28, 28, 1)  # Add 
channel dimension

                # Predict using the trained model
                prediction = model.predict(char_image)
                predicted_char = chr(np.argmax(prediction[0]) + 
ord('A'))
                print(f"Predicted character: {predicted_char}")
            else:
                print(f"Character '{user_input}' not recognized.")
        else:
            print("Invalid input! Please enter a character from A-Z or 
'exit' to quit.")

Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 73s 38ms/step - accuracy: 0.8137 - 
loss: 0.5867 - val_accuracy: 0.9865 - val_loss: 0.0421
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 81s 38ms/step - accuracy: 0.9689 - 
loss: 0.1032 - val_accuracy: 0.9842 - val_loss: 0.0456
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 73s 39ms/step - accuracy: 0.9766 - 
loss: 0.0756 - val_accuracy: 0.9904 - val_loss: 0.0285
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 79s 37ms/step - accuracy: 0.9819 - 
loss: 0.0604 - val_accuracy: 0.9923 - val_loss: 0.0241
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 73s 39ms/step - accuracy: 0.9839 - 
loss: 0.0511 - val_accuracy: 0.9933 - val_loss: 0.0234
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 82s 39ms/step - accuracy: 0.9859 - 
loss: 0.0458 - val_accuracy: 0.9904 - val_loss: 0.0326
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 72s 38ms/step - accuracy: 0.9879 - 
loss: 0.0403 - val_accuracy: 0.9898 - val_loss: 0.0296
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 80s 38ms/step - accuracy: 0.9889 - 
loss: 0.0361 - val_accuracy: 0.9910 - val_loss: 0.0255
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 70s 37ms/step - accuracy: 0.9891 - 
```

```
loss: 0.0360 - val_accuracy: 0.9928 - val_loss: 0.0206
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 82s 37ms/step - accuracy: 0.9892 -
loss: 0.0327 - val_accuracy: 0.9924 - val_loss: 0.0259

WARNING:absl:You are saving your model as an HDF5 file via
`model.save()` or `keras.saving.save_model(model)`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')` or
`keras.saving.save_model(model, 'my_model.keras')`.

Test accuracy: 0.9924
Enter a character (A-Z) to recognize or 'exit' to quit: A

----------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
<ipython-input-7-26756c79b498> in <cell line: 74>()
    127             if char_pattern is not None:
    128                 # Convert pattern to an image (for
compatibility with the trained model)
--> 129                 char_image = char_pattern.reshape(28,
28).astype('float32') / 255.0
    130                 char_image = char_image.reshape(1, 28, 28, 1)
# Add channel dimension
    131

ValueError: cannot reshape array of size 25 into shape (28,28)
```