31/10/2024

HAND SIGN RECOGNITION USING CNN

1820232044 吴宏庆
1820232061 王伟成
1820232064 杨辉宗

INTRODUCTION

This project utilizes Convolutional Neural Networks (CNNs) to classify hand signs, contributing to advancements in gesture recognition for applications like sign language translation and human-computer interaction. By leveraging CNNs' capabilities with image data, we aim to accurately identify different hand gestures. The project involves preprocessing images, normalizing data, and training a CNN model to learn gesture patterns. In this presentation, we'll explore the model's structure, training process, and evaluation, showcasing how deep learning can make communication more intuitive and accessible.

DATA PREPARATION

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torchvision import transforms

import numpy as np
import pandas as pd
```

Import Libraries

```python
# Step 1: Load the train and test CSV files
train_data = pd.read_csv('sign_mnist_train.csv')
test_data = pd.read_csv('sign_mnist_test.csv')

# Step 2: Prepare the training data
# Separate the labels and pixel data for the training set
X_train = train_data.drop(columns=['label']).values  # Pixel values (features)
y_train = train_data['label'].values  # Labels (target)

# Normalize the pixel values (0-255 -> 0-1)
X_train = X_train / 255.0

# Reshape X_train to (-1, 1, 28, 28) to match the format for CNN input (batch_size, channels, height, width)
X_train = X_train.reshape(-1, 1, 28, 28)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)

# Step 3: Create the DataLoader for the training data
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, drop_last=True)

# Step 4: Prepare the test data
# Separate the labels and pixel data for the test set
X_test = test_data.drop(columns=['label']).values  # Pixel values (features)
y_test = test_data['label'].values  # Labels (target)

# Normalize the pixel values (0-255 -> 0-1)
X_test = X_test / 255.0

# Reshape X_test to (-1, 1, 28, 28) to match the format for CNN input
X_test = X_test.reshape(-1, 1, 28, 28)

# Convert to PyTorch tensors
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create the DataLoader for the test data
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, drop_last=True)
```

Import and Transform Data

```python
# Step 5: Define the corrected CNN model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)  # First convolutional layer
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # Second convolutional layer
        self.pool = nn.MaxPool2d(2, 2)  # Max pooling
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # Adjusted fully connected layer
        self.fc2 = nn.Linear(128, 25)  # Output layer (25 classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # First conv -> relu -> maxpool
        x = self.pool(F.relu(self.conv2(x)))  # Second conv -> relu -> maxpool
        x = x.view(-1, 64 * 7 * 7)  # Correctly flatten the feature maps
        x = F.relu(self.fc1(x))  # Fully connected layer with ReLU activation
        x = self.fc2(x)  # Output layer
        return x

# Step 6: Initialize the model, criterion, and optimizer
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

MODEL STRUCTURE

Outlines the architecture of the Convolutional Neural Network (CNN) model used in this project. The model includes two convolutional layers, each followed by ReLU activation and max pooling, which helps capture important features from the input images. The convolutional layers are followed by a fully connected layer that flattens the features and prepares them for classification. The final output layer predicts the class of the hand gesture. The model is optimized using the Adam optimizer with a learning rate of 0.001, and cross-entropy loss is used as the loss function.

```python
import time
start_time = time.time()

# Create variables to track progress
epochs = 5
train_losses = []
test_losses = []
train_correct = []
test_correct = []

for epoch in range(epochs):
    trn_corr = 0
    tst_corr = 0

    # Training loop
    model.train()
    for batch_idx, (X_train_batch, y_train_batch) in enumerate(train_loader):
        batch_idx += 1  # Start batches at 1

        # Forward pass
        y_pred = model(X_train_batch)
        loss = criterion(y_pred, y_train_batch)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate training accuracy
        predicted = torch.max(y_pred.data, 1)[1]
        batch_corr = (predicted == y_train_batch).sum().item()
        trn_corr += batch_corr

        # Print loss every 600 batches
        if batch_idx % 600 == 0:
            print(f'Epoch: {epoch+1} Batch: {batch_idx} Loss: {loss.item()}')
```

TRAINING PROCESS

The code iterates through each epoch, performing a forward pass to compute predictions and calculate loss. It then performs a backward pass to adjust the model weights. Training and testing accuracy are recorded at each epoch, allowing us to monitor improvement over time. The printout at the bottom shows a summary of training accuracy and loss after each epoch, confirming that the model's performance improves steadily with training.

```
Epoch: 1 Batch: 600 Loss: 0.37831735610961914
Epoch 1 completed. Training accuracy: 69.12% Test accuracy: 84.08%
Epoch: 2 Batch: 600 Loss: 0.025819314643740654
Epoch 2 completed. Training accuracy: 97.57% Test accuracy: 88.19%
Epoch: 3 Batch: 600 Loss: 0.0018747454741969705
Epoch 3 completed. Training accuracy: 99.55% Test accuracy: 87.05%
Epoch: 4 Batch: 600 Loss: 0.0016683146823197603
Epoch 4 completed. Training accuracy: 99.57% Test accuracy: 90.70%
Epoch: 5 Batch: 600 Loss: 0.21189694106578827
Epoch 5 completed. Training accuracy: 99.36% Test accuracy: 90.69%
Training Took: 0.80 minutes!
```

TESTING AND RESULTS

```
    # Record training loss and accuracy
    train_losses.append(loss.item())
    train_correct.append(trn_corr)

    # Testing loop
    model.eval()
    with torch.no_grad():
        for X_test_batch, y_test_batch in test_loader:
            y_val = model(X_test_batch)
            loss = criterion(y_val, y_test_batch)

            # Calculate test accuracy
            predicted = torch.max(y_val.data, 1)[1]
            tst_corr += (predicted == y_test_batch).sum().item()

    # Record test loss and accuracy
    test_losses.append(loss.item())
    test_correct.append(tst_corr)

    # Print epoch summary
    print(f'Epoch {epoch+1} completed. Training accuracy: {trn_corr/len(train_dataset)*100:.2f}% Test accuracy: {tst_corr/len(test_dataset)*100:.2f}%')

total_time = time.time() - start_time
print(f'Training Took: {total_time/60:.2f} minutes!')
```

Code for evaluating the model on training and test data, tracking accuracy and loss per epoch to ensure effective generalization and performance on unseen data.

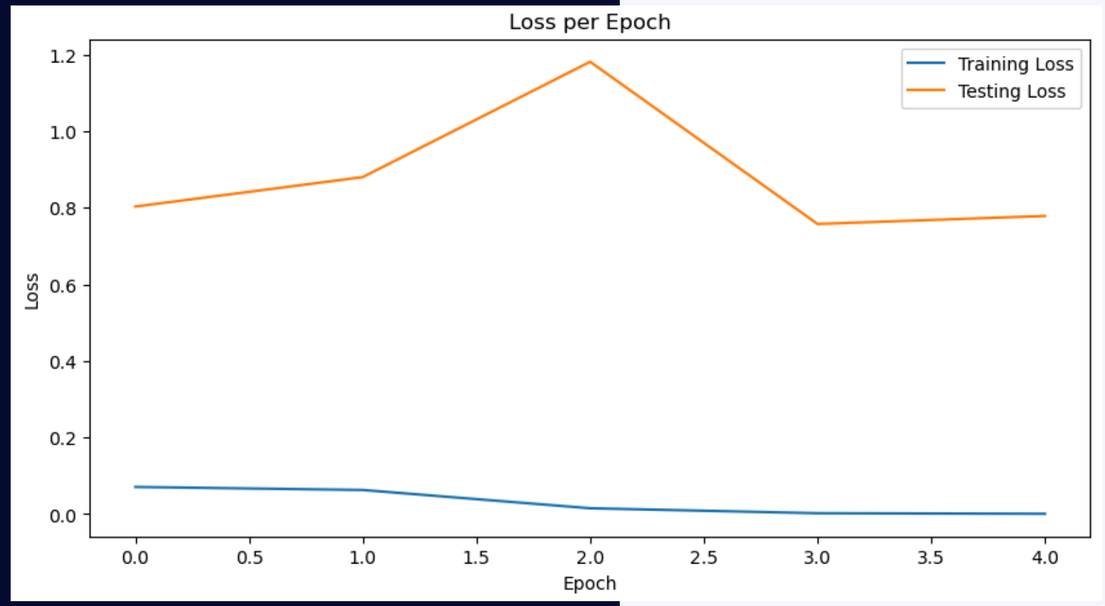
PLOTTING

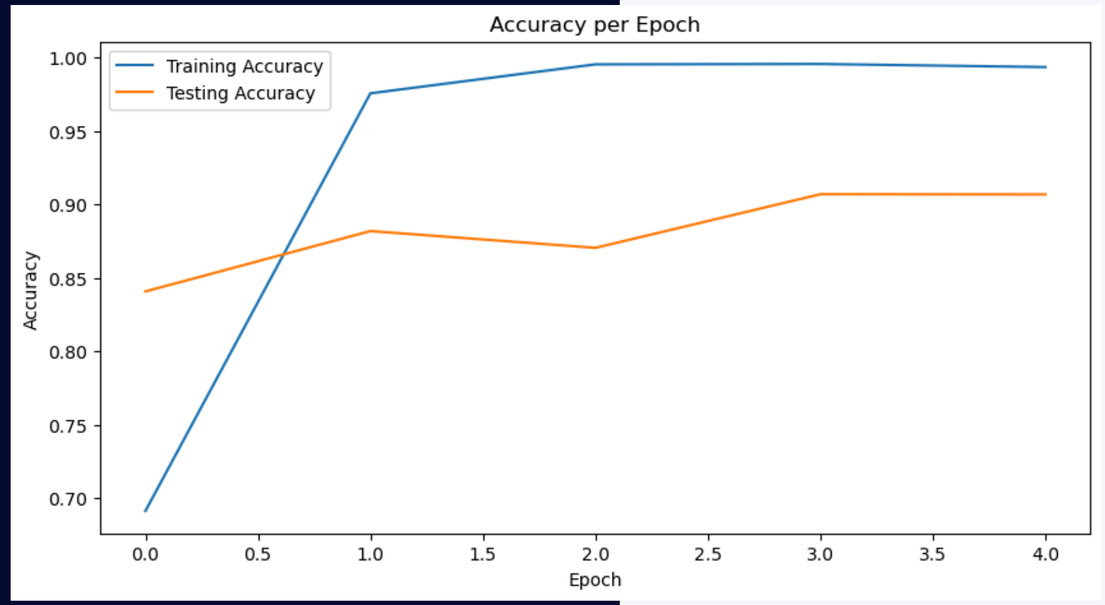
The code used to visualize the model's performance. Two plots are created: one for Loss per Epoch and another for Accuracy per Epoch. The loss plot shows how training and testing loss evolve over time, helping to identify potential overfitting. The accuracy plot tracks the model's accuracy on both training and testing datasets, indicating how well the model learns and generalizes. These visualizations are essential for understanding model behavior and making adjustments if needed.

```python
import matplotlib.pyplot as plt

# Plot losses
plt.figure(figsize=(10,5))
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.legend()
plt.title('Loss per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

# Plot accuracies
plt.figure(figsize=(10,5))
plt.plot([t/len(train_dataset) for t in train_correct], label='Training Accuracy')
plt.plot([t/len(test_dataset) for t in test_correct], label='Testing Accuracy')
plt.legend()
plt.title('Accuracy per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```

MODEL PERFORMANCE OVERVIEW

- Training loss decreases consistently, while testing loss fluctuates, indicating potential overfitting.
- Training accuracy reaches nearly 100%, but testing accuracy stabilizes around 85-90%.
- Results suggest the model learns well





TEST SAMPLE

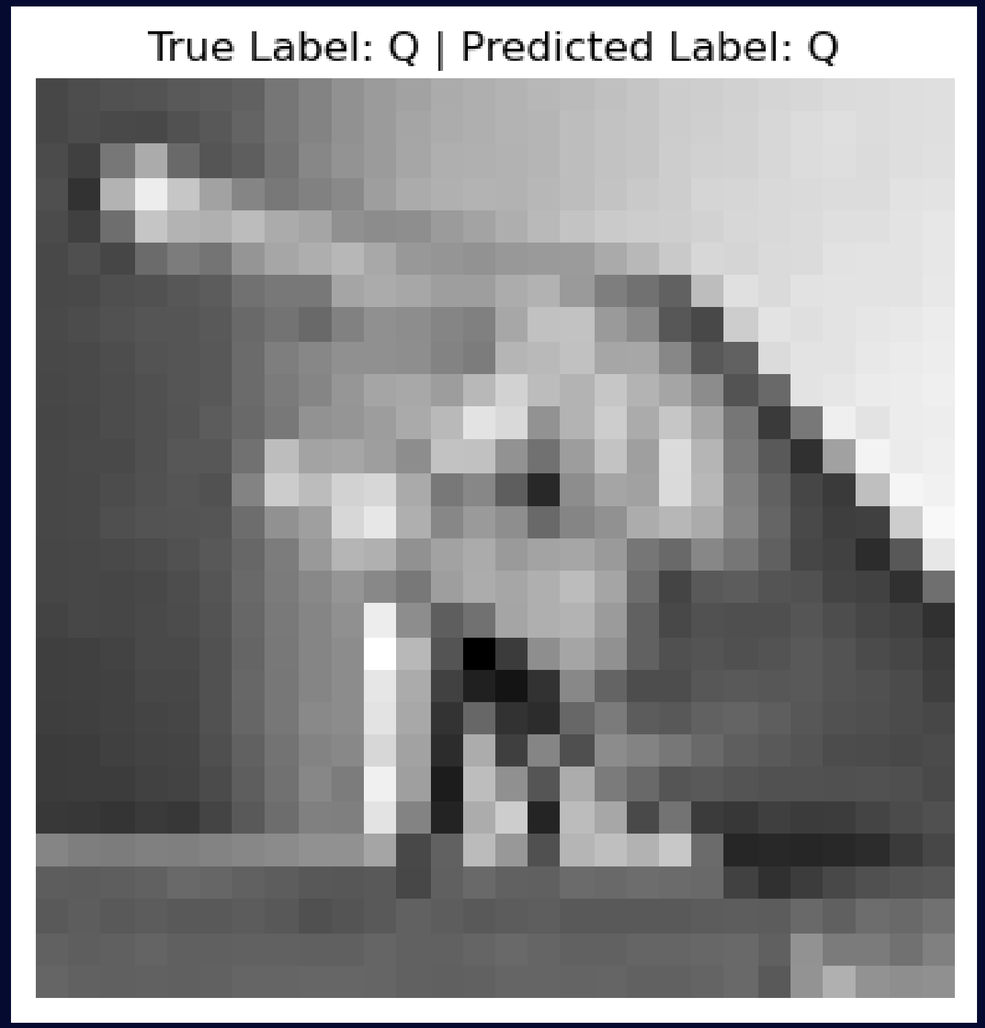
Randomly selects a test sample for model evaluation, displays the true and predicted labels, and visually checks the model's performance on unseen data.

```python
import random
import numpy as np
import torch
import matplotlib.pyplot as plt
import string

# Set the random seed
seed_value = 42
random.seed(seed_value)
np.random.seed(seed_value)
torch.manual_seed(seed_value)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Create a mapping from labels to letters
letters = [letter for letter in string.ascii_uppercase if letter not in ('J', 'Z')]
label_to_letter = {i: letter for i, letter in enumerate(letters)}

# Randomly select an image from the test dataset
total_test_samples = len(test_dataset)
random_idx = random.randint(0, total_test_samples - 1)
image_tensor, true_label = test_dataset[random_idx]
image_tensor = image_tensor.unsqueeze(0)  # Add batch dimension

# Perform inference
model.eval()
with torch.no_grad():
    output = model(image_tensor)
    _, predicted_label = torch.max(output, 1)
    predicted_label = predicted_label.item()
    true_label = true_label.item()

# Map labels to letters
true_letter = label_to_letter[true_label]
predicted_letter = label_to_letter[predicted_label]
```

```python
# Display the image and prediction
image_np = image_tensor.squeeze().numpy()

plt.imshow(image_np, cmap='gray')
plt.title(f'True Label: {true_letter} | Predicted Label: {predicted_letter}')
plt.axis('off')
plt.show()
```

RESULTS

This sample result shows the model correctly identifying the letter "Q," illustrating its effectiveness in gesture recognition.



CONCLUSION

This project demonstrates the effectiveness of CNNs in recognizing American Sign Language gestures, showing the potential of deep learning to enhance communication. Through data preprocessing, model training, and evaluation, we achieved a reliable gesture recognition model. The results highlight AI’s role in making communication more accessible and inclusive.

THANK YOU

We hope you found the information insightful and engaging.