北京理工大学

Hand Sign Recognition Using CNNs

1820232044 吴宏庆
1820232061 王伟成
1820232064 杨辉宗

Abstract:

Hand sign recognition plays a significant role in facilitating communication, especially for individuals with hearing and speech impairments. This project investigates the use of Convolutional Neural Networks (CNNs) for hand sign recognition, a vital tool in advancing communication aids and gesture-based interactions. The project involves data preparation by normalizing and reshaping image data, followed by training a CNN model designed for image classification. The trained model was evaluated using test data, showcasing its ability to accurately classify various hand signs. The findings indicate that CNNs are highly effective for visual recognition tasks, with potential future improvements including data augmentation and model optimization for even better performance.

Introduction

Background

With the increasing integration of technology into daily life, gesture-based interaction has become an essential component of human-computer interfaces. Hand sign recognition is at the forefront of this technological evolution, providing new ways to communicate without speech or physical contact. Historically, gesture recognition systems relied on traditional image processing techniques and rule-based algorithms, which often failed to adapt to diverse hand positions and shapes. The introduction of Convolutional Neural Networks (CNNs) has overcome these challenges by enabling automated feature extraction and robust learning from large datasets, dramatically improving the performance and flexibility of recognition systems.

Objective

This project aims to build a CNN-based model that can classify hand signs with high accuracy. By training and evaluating this model, the goal is to demonstrate the effectiveness of deep learning in gesture recognition tasks and provide a pathway for future improvements.

Overview

This report details the steps taken in the project, including data preparation, model structure, training and evaluation, and results analysis. It aims to provide a comprehensive overview of the CNN's implementation for hand sign recognition and discuss potential areas for improvement.

Methodology

Data Collection

The dataset consists of images of hand signs for letters in the English alphabet, specifically covering 25 letters (A-Z, excluding J and Z). Each image is labeled with its corresponding letter, and the data is divided into training and test sets to enable model validation.

```
# 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')
```

'Import dataset'

Data Preprocessing

The dataset used consists of grayscale images representing hand signs for various letters in the alphabet, specifically the letters A-Z, excluding J and Z.

Grayscale Processing:

Each image is preprocessed to a grayscale format if not already, as this simplifies the input and reduces computational load by eliminating color information, which is not essential for distinguishing hand signs.

Label Encoding:

Each letter in the dataset is mapped to a unique numerical label to facilitate training. This allows the CNN to treat the classification task as a multi-class problem, where each output neuron represents one of the classes (letters).

```python
# 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)
```

‘Split label for feature and target, Transform data type to tensor and reshape to fit the model parameter’

Model Architecture

Model Selection:

Convolutional Neural Networks (CNNs) were chosen for this project as they are particularly effective for image classification tasks. CNNs are capable of learning spatial hierarchies within images by capturing edges, shapes, and other features through successive convolutional and pooling layers. This makes CNNs highly suited for tasks like hand sign classification, where spatial patterns distinguish different letters.

The architecture for the CNN is relatively simple yet effective for this task, including the following layers:

Convolutional Layers:

- conv1: The first convolutional layer has 32 filters, with each filter sized at 3x3. It is responsible for capturing low-level features such as edges.
- conv2: The second convolutional layer has 64 filters of the same size (3x3). This layer learns more complex features by building on the information extracted by conv1.

Pooling Layer:

- Max Pooling: A max-pooling layer is used to reduce the spatial dimensions of the feature maps and improve computational efficiency. Max-pooling helps retain the most relevant information while down-sampling the data.

Fully Connected Layers:

- fc1: A fully connected layer with 128 neurons that aggregates features extracted by previous layers and serves as a transition to the final classification layer.
- fc2: The final output layer with 25 neurons (one for each letter class, excluding J and Z). This layer provides the classification result by outputting probabilities for each class.

Activation Functions:

- ReLU (Rectified Linear Unit): Applied after each convolutional and fully connected layer, ReLU introduces non-linearity into the model, allowing it to learn more complex representations.
- Softmax: Applied to the final output layer (fc2), Softmax converts the logits into probabilities across the classes, where the class with the highest probability is the model's predicted label.

```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)
```

‘Create CNN and evaluate each layer of network’

Training Process

Training Setup:

Dataset:

- Data sets are already split for using train and test.

Evaluation Metrics:

The primary metrics used to assess model performance are:

- Training Loss and Testing Loss: Calculated at each epoch to monitor how well the model fits the training data and how well it generalizes to the test set.
- Training Accuracy and Testing Accuracy: Track the percentage of correct predictions on the training and testing datasets, providing insight into the model's learning progress and overfitting risk.

Plots for Analysis:

- Loss per Epoch: A plot of the training and testing loss over epochs to visualize the model's convergence behavior. Decreasing training and testing losses indicate effective learning, while diverging losses might suggest overfitting.
- Accuracy per Epoch: A plot of the training and testing accuracy over epochs provides an additional perspective on model performance, with converging lines suggesting effective generalization.

```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()}')
```

‘Start timer to track the training duration and train the model’

Results and Observations:

Model Performance:

Training and testing accuracy achieved during the final epoch demonstrate the model's capability to classify hand signs with high accuracy. Observing the convergence of training and testing accuracy further indicates if the model generalizes well or is prone to overfitting.

Loss and Accuracy Curves:

Analysis of these curves reveals:

- Stable Convergence: If both training and testing losses decrease steadily and stabilize, this suggests a well-trained model.
- Overfitting: If training loss keeps decreasing while testing loss increases, the model might be memorizing the training data rather than generalizing well.

Sample Prediction:

A test sample is randomly selected, and the model predicts its corresponding letter. This example helps demonstrate real-time model inference capability and provides qualitative insight into the model's accuracy on individual samples.

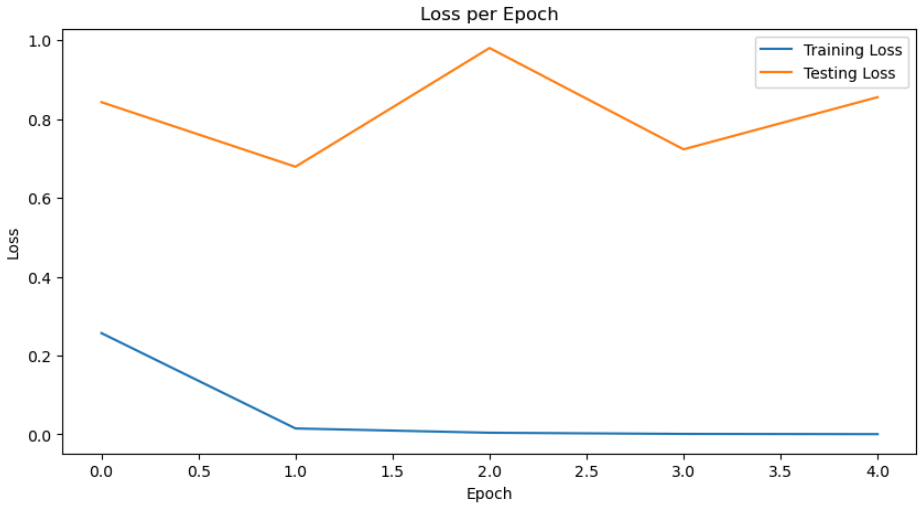
Visual Comparison: The predicted label is displayed alongside the true label, offering an intuitive understanding of the model's performance.

```
    # 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!')
```

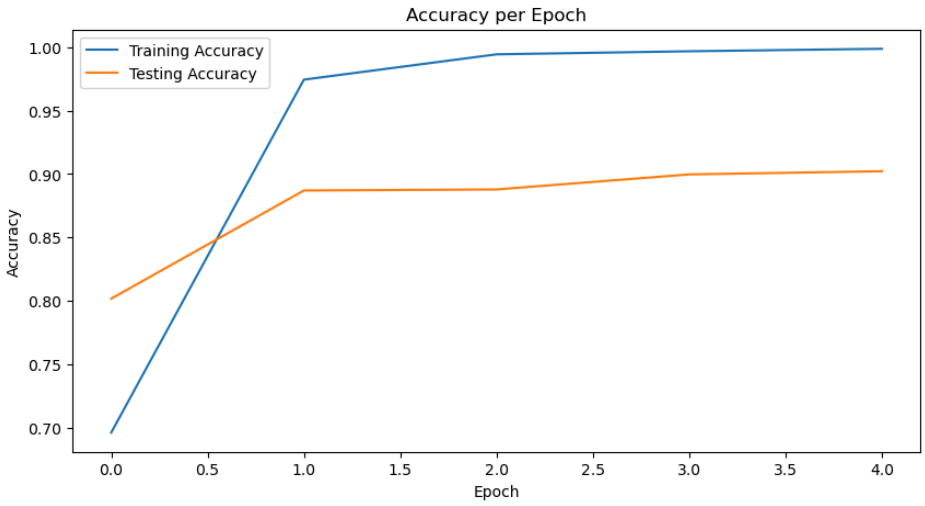
```
Epoch: 1 Batch: 600 Loss: 0.4968568682670593
Epoch 1 completed. Training accuracy: 69.60% Test accuracy: 80.17%
Epoch: 2 Batch: 600 Loss: 0.0705220103263855
Epoch 2 completed. Training accuracy: 97.45% Test accuracy: 88.71%
Epoch: 3 Batch: 600 Loss: 0.0053220260888333809
Epoch 3 completed. Training accuracy: 99.45% Test accuracy: 88.79%
Epoch: 4 Batch: 600 Loss: 0.00166258320678025 48
Epoch 4 completed. Training accuracy: 99.69% Test accuracy: 89.97%
Epoch: 5 Batch: 600 Loss: 0.0009814526420086622
Epoch 5 completed. Training accuracy: 99.89% Test accuracy: 90.23%
Training Took: 0.83 minutes!
```

'Tracking lose in both train and test'

Results



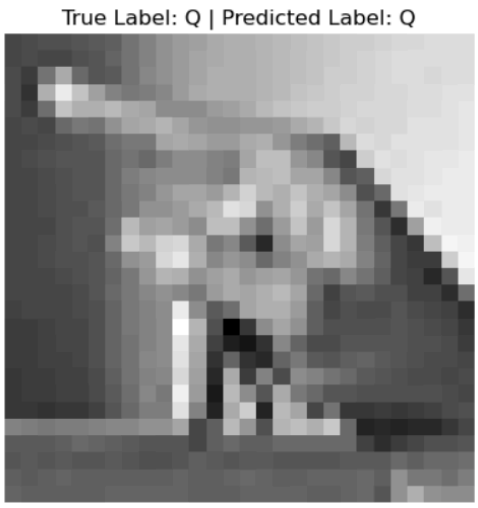
‘Graphical Data Loss per Epoch’



‘Graphical Data Accuracy per Epoch’

Visualize testing

```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]

# 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()
```



‘Do visualize testing’

Discussion

Analysis of Results

The performance of the CNN model is visualized through the training and testing loss and accuracy plots:

- **Training and Testing Loss**: The plot shows a decreasing trend in both training and testing losses across epochs, indicating effective learning. However, any divergence or increase in testing loss towards later epochs would suggest overfitting.
- **Training and Testing Accuracy**: The training accuracy improves with epochs, but if the testing accuracy plateaus or fluctuates, it could point to the model's limitations on the testing data, possibly hinting at underfitting or overfitting depending on the trend.

Challenges Encountered

One key challenge was determining the appropriate number of neurons for each hidden layer in the CNN. Setting too few neurons could limit the model's capacity, while too many could lead to overfitting or excessive computation without improving accuracy. This required experimentation and analysis to achieve the best configuration for both performance and generalization.

Potential Improvements

1. **Data Augmentation**: Applying augmentations (e.g., rotations, flips) would increase the effective training dataset size, helping reduce overfitting.
2. **Regularization**: Techniques such as dropout or L2 regularization could be applied to prevent overfitting.

3. **Hyperparameter Tuning**: Adjusting parameters such as the learning rate, batch size, or number of epochs might yield better generalization on unseen data.
4. **Ensemble Methods**: Combining predictions from multiple models or using ensemble techniques could improve overall accuracy and robustness.

References

1. Codemy.com. (n.d.). *Deep learning* [Playlist]. (YouTube) https://youtube.com/playlist?list=PLCC34OHNcOtpcgR9LEYSdi9r7XIbpkpK1

2. **Kaggle**. (n.d.). *Sign Language MNIST*. Retrieved [date accessed], from https://www.kaggle.com/datasets/datamunge/sign-language-mnist