CSE-625-Term Project Report
Misclassified Image Analysis in Neural Networks

Vivek Inturi
Shruthin Sainapuram
Venkata Sai Deepak Reddy
Rajasekhar Dwarakapally
Hemanth Ambati

12/10/2024

Project Statement
The project focuses on automating the recognition of handwritten English letters by leveraging deep learning techniques. Using the EMNIST Letters Dataset, the goal is to classify input images into one of 26 classes (A-Z), enabling applications like handwriting digitization and automated text recognition.

Objective
The objective of this project is to build and train a robust convolutional neural network (CNN) to accurately classify handwritten lowercase letters (A-Z) from the EMNIST Letters Dataset. The project aims to enhance model generalization through advanced data augmentation techniques, evaluate the model's performance using accuracy metrics and confusion matrices, analyze misclassified samples to identify patterns and areas for improvement, and demonstrate the use of PyTorch for implementing and training deep learning models effectively.

About the Dataset
Dataset Name: EMNIST (Extended MNIST)
The **EMNIST dataset** is a comprehensive collection designed for different machine learning tasks related to handwritten digits and letters. It is an extension of the popular MNIST dataset. Whether it recognizes numbers depends on the specific split you use.
EMNIST Overview:

- **Total Dataset**: Contains 814,255 images across different splits.
- **Image Size**: Each image is 28×28, grayscale.
- **Classes**: Depending on the split, classes can range from digits (0-9) to uppercase or lowercase letters (A-Z).

Splits in EMNIST:
EMNIST is divided into multiple splits, each designed for specific tasks:

1. **By Class**: Includes both digits (0-9) and letters (A-Z), 62 classes in total.
2. **By Merge**: Merges similar uppercase and lowercase letters (e.g., 'C' and 'c'), with 47 classes.
3. **Balanced**: Balanced dataset with digits and letters, 47 classes.
4. **Digits**: Only contains handwritten digits (0-9), 10 classes.
5. **Letters**: Only contains lowercase handwritten letters, 26 classes.
6. **MNIST**: A replica of the original MNIST dataset with handwritten digits (0-9), 10 classes.

General description of the Approach

Performed the following steps to classify handwritten lowercase letters from the EMNIST Letters dataset using a convolutional neural network (CNN)

1. **Device Configuration for Model Training**

   We initially train the model on a CPU, followed by training on a GPU. This approach allows us to compare the runtime optimization and evaluate the impact on accuracy between the two devices. By utilizing the GPU, we also leverage its capabilities for parallel computation, enhancing the efficiency and speed of our training process.

2. **Data Loading and Preprocessing**

   Loaded the EMNIST Letters dataset using PyTorch's "datasets.EMNIST".

   Divided the dataset into:

   - Training Set: 124,800 samples.
   - Testing Set: 20,800 samples.
   - Selected the 'letters' split, which contains 26 classes representing the letters A-Z.

   Data Augmentation:

   - Applied various data augmentation techniques to the training data to enhance model robustness and generalization:
     i. **Random Rotation**: Rotated images randomly within ±15 degrees to simulate varied handwriting angles.
     ii. **Random Horizontal Flip**: Flipped images horizontally to account for writing variations.
     iii. **Color Jitter**: Adjusted brightness, contrast, saturation, and hue slightly.
     iv. **Random Affine Transform**: Applied random translations (±10% of image dimensions) to simulate shifts.
   - **Normalization**: Normalized images to have a mean of 0.5 and a standard deviation of 0.5 to standardize input data.

   Data Loaders:

   - Created training and testing data loaders with a batch size of 64 for efficient data handling during training and evaluation.

3. **Model Design**

   Convolutional Neural Network (CNN) Architecture:

   - Implemented a deep CNN named AdvancedNet with the following components:
   - **Convolutional Layers**: Multiple layers with increasing filter sizes to capture hierarchical features.
   - **Batch Normalization**: Applied after convolutional layers to normalize activations and accelerate training.
   - **Activation Functions**: Used ReLU (Rectified Linear Unit) to introduce non-linearity.
   - **Pooling Layers**: Employed MaxPooling to reduce spatial dimensions and extract dominant features.
   - **Adaptive Average Pooling**: Aggregated features to a fixed size before the fully connected layer.

- **Fully Connected Layer**: Mapped extracted features to the 26 output classes corresponding to letters A-Z.
- **Dropout Layer**: Included to prevent overfitting by randomly disabling neurons during training.

4. **Training the Model**

- **Loss Function and Optimizer**:
  - Used **CrossEntropyLoss** suitable for multi-class classification problems.
  - Employed the **Adam optimizer** with a learning rate of 0.001 for efficient gradient descent optimization.
- **Training Loop**:
  - Trained the model over **10 epochs**, iterating over the training data in batches.
  - For each batch:
    - Forward propagated inputs through the network to obtain predictions.
    - Calculated the loss by comparing predictions with true labels.
    - Backpropagated the loss to compute gradients.
    - Updated model parameters using the optimizer.
  - Monitored and printed training loss and accuracy after each epoch to track progress.

5. **Evaluating the Model**

Testing:

- Evaluated the trained model on the test dataset to assess its generalization ability.
- Calculated the overall test accuracy, achieving approximately 87.38%.

Misclassification Analysis:

- Identified misclassified images during evaluation.
- Saved misclassified images to a directory (./misclassified_images) for further inspection.
- Reviewed the saved misclassified images to understand common errors.
- Analyzed patterns such as confusion between visually similar letters.

6. **Saving the Model**

Model Persistence:

- Saved the trained model's state dictionary to a file (model_state.pth).
- Allows for future loading and inference without retraining

Implementation details (including important source code snippets)

1. **Data Loading and Preprocessing**

The EMNIST Letters dataset was loaded using PyTorch's torchvision.datasets. Data augmentation techniques were applied to enhance generalization, and normalization ensured consistent input scaling.

```
# Ensure the directory for misclassified images exists
misclassified_dir = './misclassified_images'
os.makedirs(misclassified_dir, exist_ok=True)

# Define transformations with enhanced data augmentation
transform = transforms.Compose([
    transforms.RandomRotation(15),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
# Load the EMNIST dataset (letters split)
train_data = datasets.EMNIST(root='./data', split='letters', train=True, download=True, transform=transform)
test_data = datasets.EMNIST(root='./data', split='letters', train=False, download=True, transform=transform)

print(f'Train Data Size: {len(train_data)}')
print(f'Test Data Size: {len(test_data)}')
```

```
Train Data Size: 124800
Test Data Size: 20800
```

```
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

2. Model Architecture

A convolutional neural network (CNN) named AdvancedNet was implemented with multiple layers for feature extraction, batch normalization for stable training, and dropout to prevent overfitting.

```
# Define a complex neural network with dropout and batch normalization
class AdvancedNet(nn.Module):
    def __init__(self):  # Corrected from _init_ to __init__
        super(AdvancedNet, self).__init__()  # Corrected from _init_ to __init__
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.AdaptiveAvgPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(128, 27)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

3. Training the Model

The model was trained using CrossEntropyLoss and the Adam optimizer. The training loop handled forward propagation, loss computation, backpropagation, and parameter updates for 10 epochs.

```python
def train_model(epochs=10):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        total = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        print(f'Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(train_loader):.4f}, Accuracy: {accuracy:.2f}%')

    torch.save(model.state_dict(), 'model_state.pth')
```

4. **Evaluating the Model**
   The model's accuracy was calculated on the test set by comparing predicted labels with ground-truth labels.

```python
def evaluate_model():
    model.load_state_dict(torch.load('model_state.pth', map_location=device))
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Test Accuracy: {accuracy:.2f}%')
```

5. **Saving and Analyzing Misclassified Images**
   Misclassified images were identified and saved for further analysis.

```python
def save_and_show_misclassified_images():
    misclassified = []
    model.eval()
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            misclass_mask = predicted != labels
            misclassified_images = images[misclass_mask]
            misclassified_labels = labels[misclass_mask]
            misclassified_preds = predicted[misclass_mask]
            for img, true, pred in zip(misclassified_images, misclassified_labels, misclassified_preds):
                img_path = os.path.join(misclassified_dir, f'{true}_{pred}_{len(misclassified)}.png')
                plt.imsave(img_path, img.cpu().squeeze(), cmap='gray')
                misclassified.append((img.cpu(), true.item(), pred.item()))
    print(f'Saved {len(misclassified)} misclassified images to {misclassified_dir}')
```

Results and Contributions

Results

CPU Output

1. **Performance:**
   - Achieved a test accuracy of 87.38% after training for 10 epochs.
   - Training Loss decreased consistently, reaching 0.4806 by the 10th epoch.

- Accuracy improved steadily from 64.31% (Epoch 1) to 84.28% (Epoch 10).

2. **Misclassified Images:**
    - Identified 2,541 misclassified images, saved to ./misclassified_images.
3. **Execution Time:**
    - Training on the CPU took 60 minutes to complete.

GPU Output

1. **Performance**:
    - Achieved a **test accuracy of 87.91%**, slightly higher than the CPU.
    - Training Loss decreased consistently, reaching **0.4763** by the 10th epoch.
    - Accuracy improved from **63.56%** (Epoch 1) to **84.39%** (Epoch 10).
2. **Misclassified Images**:
    - Identified **2,485 misclassified images**, saved to ./misclassified_images.
3. **Execution Time**:
    - Training on the GPU took only **25 minutes**, significantly faster than the CPU.

Comparison: CPU vs. GPU

Aspect	CPU	GPU
Test Accuracy	87.38%	87.91%
Training Loss (Epoch 10)	0.4806	0.4763
Training Time	60 minutes	25 minutes
Misclassified Images	2,541	2,485

Contributions

1. **Model Development:**
    - Designed a deep convolutional neural network (CNN) with dropout and batch normalization, enabling the network to learn complex features while preventing overfitting.
2. **Data Augmentation:**
    - Implemented advanced augmentation techniques (e.g., rotations, flips, and affine transformations) to improve model robustness against variations in handwritten styles.
3. **Evaluation:**
    - Evaluated the model on the EMNIST Letters test dataset, providing a benchmark accuracy and analysis of misclassified samples.
4. **Error Analysis:**
    - Conducted a detailed error analysis by saving and reviewing misclassified images, leading to potential insights for improving the model.
5. **Reproducibility:**
    - Saved the model state (model_state.pth) to enable further usage or fine-tuning without requiring retraining.
6. **Framework Utilization:**

- Leveraged PyTorch and its ecosystem effectively for dataset handling, model training, and evaluation.

7. **Practical Applications:**
    - Laid the groundwork for practical handwriting recognition systems by demonstrating effective classification of lowercase English letters.

Team Sharing:

Task	**who**
Project Idea	**Shruthin**
Code	**Deepak & Shruthin**
PPT	**Hemanth & Rajashekhar**
Report	**Vivek**

Acknowledgments:
This project utilized various resources and tools, and their contributions are acknowledged as follows:

1. **Dataset:**
    - The EMNIST Letters dataset was obtained from the publicly available EMNIST dataset, which is an extension of the original MNIST dataset.
2. **Frameworks and Libraries:**
    - PyTorch: For building, training, and evaluating the deep learning model.
    - Torch vision: For dataset handling and implementing data augmentations.
    - Matplotlib: For visualizing and saving misclassified images.
3. **AI Assistance:**
    - OpenAI's ChatGPT was used to clarify concepts, draft implementation details, and structure the documentation effectively.
4. **Development Tools:**
    - Jupiter Notebook was used for interactive code development and testing.
5. **Additional Resources:**
    - PyTorch and Torch vision documentation for understanding APIs and implementation details.
    - Online tutorials and forums for resolving technical issues during development**.**

Conclusion:

This project successfully implemented a CNN to classify handwritten lowercase letters using the EMNIST Letters dataset, achieving a test accuracy of **87.38%**. Advanced data augmentation improved robustness, and misclassified images were analyzed for insights into model performance. The project established a reliable baseline for handwriting recognition, demonstrating effective use of PyTorch and providing a foundation for future improvements, such as handling ambiguous samples or expanding to include uppercase letters and digits.