CSE4079.1 - Introduction to Deep Learning

Digit and Rotation Classification on MNIST Dataset Using a Multi-Task Learning Approach

Group Members:

Muhammed Talha Karagül - 150120055

Abdullah Kan - 150121076

ABSTRACT

The objective of this study is to test the capability of a Convolutional Neural Network (CNN), developed using the PyTorch library, to simultaneously classify handwritten digits and their corresponding rotation angles from a single image. For this purpose, a custom dataset was created by selecting specific digits from the standard MNIST dataset and augmenting them with systematic rotations at 0, 90, 180, and 270 degrees. The model was designed with a multi-task learning architecture, featuring shared convolutional layers and task-specific fully connected output layers. The model was trained for 10 epochs using the Adam optimizer. Upon evaluation on the test set, the model achieved high performance, reaching 98.86% accuracy in the digit classification task and 99.59% accuracy in the rotation classification task. These findings demonstrate that the proposed CNN model and the multi-task learning approach are highly effective in extracting both content-based (digit) and context-based (rotation) features from images.

MATERIALS AND METHODS

Dataset:

- **Source:** The foundation of the project is the standard MNIST dataset, which consists of 28x28 pixel grayscale images of handwritten digits.
- **Customization:** The dataset was customized to fit the project's multi-task goal:
    - **Digit Selection:** Only a subset of digits, specifically [1, 2, 3, 4, 5, 7], was selected from the original dataset.
    - **Data Augmentation:** Each selected image was systematically rotated at four distinct angles: 0, 90, 180, and 270 degrees. This quadrupled the size of the effective dataset.
    - **Label Generation:** Two separate labels were generated for each image: a digit label (re-indexed from 0 to 5) and a rotation label (indexed from 0 to 3).

```python
class CustomRotatedMNIST(Dataset):
  def __init__(self, train=True):
    self.base_dataset = datasets.MNIST(root='./data', train=train, download=True, transform=transforms.ToTensor())
    self.rotation_angles = [0, 90, 180, 270]
    self.rotation_labels = {0:0, 90:1, 180:2, 270:3}
    self.valid_digits = [1, 2, 3, 4, 5, 7]  # sadece bu rakamlar kullanılacak
    self.label_map = {label: idx for idx, label in enumerate(self.valid_digits)}

    self.samples = []
    to_pil = ToPILImage()
    to_tensor = ToTensor()

    for img_tensor, digit_label in self.base_dataset:
      if digit_label not in self.valid_digits:
          continue

      new_label = self.label_map[digit_label]  # yeniden numaralandır (örneğin 1 → 0, 2 → 1, ...)

      for angle in self.rotation_angles:
        rotated_img = rotate(to_pil(img_tensor),angle)
        rotated_tensor = to_tensor(rotated_img)
        self.samples.append((rotated_tensor, new_label, self.rotation_labels[angle]))

  def __len__(self):
      return len(self.samples)

  def __getitem__(self, idx):
      image, digit_label, rotation_label = self.samples[idx]
      return image, digit_label, rotation_label
```

Model Architecture:

- The model is a Convolutional Neural Network (CNN) designed for multi-task output. The architecture is detailed below:
    - **Input Layer:** 28x28x1 grayscale images.
    - **conv1 & pool1:** A convolutional layer with a 3x3 kernel that transforms 1 input channel to 8 feature maps, followed by a max-pooling layer. These layers capture low-level features like edges and curves.
    - **conv2 & pool2:** A second set of convolutional and pooling layers that increases the feature map depth from 8 to 16, learning more complex patterns.
    - **fc1 (Shared Layer):** A fully connected layer with 128 neurons that takes the flattened output from the convolutional layers. This layer acts as a **shared feature bottleneck**, learning a common representation for both tasks.
    - **Output Layers (fc_digit & fc_rotation):** Two separate fully connected layers that take the 128-neuron vector from the shared layer as input. fc_digit outputs a 6-class prediction for the digit, while fc_rotation outputs a 4-class prediction for the rotation.

```python
#2 creat FCNN
class NN (nn.Module):
    def __init__(self, input_size,num_classes): #28x28=784 input size, 10 classes
        super(NN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=8,kernel_size=(3,3), stride=(1,1),padding=(1,1))
        self.pool = nn.MaxPool2d(kernel_size=(2,2),stride=(1,1),padding=(1,1))
        self.conv2 = nn.Conv2d(in_channels=8,out_channels=16,kernel_size=(3,3), stride=(1,1),padding=(1,1))
        self.fc1 = nn.Linear(14400, 128)
        self.fc_digit = nn.Linear(128, num_classes)
        self.fc_rotation = nn.Linear(128, 4)
    def forward(self,x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = x.reshape(x.shape[0], -1) #flattening the data
        x = self.fc1(x)
        x = F.relu(x)
        digit_output = self.fc_digit(x)
        rotation_output = self.fc_rotation(x)
        return digit_output, rotation_output
```

Hyperparameters:

Parameter	Value
Learning Rate	0.001
Batch Size	64
Number of Epochs	10

Model Training

This code snippet displays the core training function responsible for updating the model's weights. For each batch of data, the following key steps are performed:

1. Forward Pass: The input data is passed through the model to obtain simultaneous predictions for both the digit (digit_output) and the rotation (rotation_output).
2. Loss Calculation: The CrossEntropyLoss function is used to calculate the loss for each task independently (loss_digit, loss_rotation).
3. Combined Loss: The total loss is computed by summing the individual losses from both tasks. This combined loss signal guides the model to learn features that are beneficial for both objectives concurrently.
4. Backpropagation: The standard backpropagation process is executed. The optimizer's gradients are cleared (optimizer.zero_grad()), the gradient of the total loss with respect to the model parameters is computed (loss.backward()), and the optimizer updates the weights (optimizer.step()).

This loop iterates for a predefined number of epochs, progressively minimizing the combined loss and training the model to be proficient at both digit and rotation classification.

```python
def train(model, train_loader, optimizer, criterion_digit, criterion_rotation, device, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        total_loss = 0
        for batch_idx, (data, digit_labels, rotation_labels) in enumerate(train_loader):
            data = data.to(device)
            digit_labels = digit_labels.to(device)
            rotation_labels = rotation_labels.to(device)

            digit_output, rotation_output = model(data)
            loss_digit = criterion_digit(digit_output, digit_labels)
            loss_rotation = criterion_rotation(rotation_output, rotation_labels)
            loss = loss_digit + loss_rotation

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch+1}/{num_epochs} | Avg Loss: {avg_loss:.4f}")
```

Evaluation Metrics:

- The model's performance was measured using the following standard classification metrics:
  - **Accuracy:** $(TP + TN)/(TP + TN + FP + FN)$
  - **Precision:** $TP/(TP + FP)$
  - **Recall:** $TP/(TP + FN)$
  - **F1-Score:** $2\times(Precision\times Recall)/(Precision + Recall)$
  - The macro average for Precision, Recall, and F1-Score was used to ensure that each class contributed equally to the final metric, which is important for handling potential class imbalances.

```python
def calculate_metrics(preds, targets, num_classes):
    TP = torch.zeros(num_classes)
    FP = torch.zeros(num_classes)
    FN = torch.zeros(num_classes)

    for cls in range(num_classes):
        TP[cls] = ((preds == cls) & (targets == cls)).sum().item()
        FP[cls] = ((preds == cls) & (targets != cls)).sum().item()
        FN[cls] = ((preds != cls) & (targets == cls)).sum().item()

    precision = TP / (TP + FP + 1e-8)
    recall = TP / (TP + FN + 1e-8)
    f1 = 2 * precision * recall / (precision + recall + 1e-8)
    accuracy = (preds == targets).sum().item() / len(targets)

    macro_precision = precision.mean().item()
    macro_recall = recall.mean().item()
    macro_f1 = f1.mean().item()

    print(f"TP: {TP}")
    print(f"FP: {FP}")
    print(f"FN: {FN}")

    return {
        "accuracy": accuracy,
        "macro_precision": macro_precision,
        "macro_recall": macro_recall,
        "macro_f1": macro_f1
    }
```

Evaluate

This code snippet represents the evaluation function that assesses the model's performance on the test dataset. It operates without updating the model's weights and serves to measure the effectiveness of the learned representations for both tasks.

- Evaluation Mode Activation: The model is switched to evaluation mode using model.eval(), which disables certain training-specific behaviors such as dropout and batch normalization updates.
- Forward Pass: For each batch in the test loader, the input data is passed through the model to obtain predictions for both the digit (digit_output) and the rotation (rotation_output) classification tasks.
- Prediction Collection: Class predictions are obtained using the argmax function on the output tensors. These predicted labels, along with the ground truth labels, are stored for the entire dataset to enable comprehensive metric computation.
- All metrics are computed using macro-averaging to ensure equal importance is given to each class, which is particularly important in the presence of class imbalance.
- Task-Specific Reporting: Evaluation results are reported separately for both tasks—digit classification and rotation classification—highlighting the model's ability to generalize its learning to unseen data.

This function provides an objective, quantitative summary of the model's dual-task performance and is a critical step in validating the effectiveness of the multi-task learning approach.

Evaluation Metrics:

- The model's performance was measured using the following standard classification metrics:
  - **Accuracy:** $(TP + TN)/(TP + TN + FP + FN)$
  - **Precision:** $TP/(TP + FP)$
  - **Recall:** $TP/(TP + FN)$
  - **F1-Score:** $2\times(Precision\times Recall)/(Precision + Recall)$
  - The `macro` average for Precision, Recall, and F1-Score was used to ensure that each class contributed equally to the final metric, which is important for handling potential class imbalances.

```python
def calculate_metrics(preds, targets, num_classes):
    TP = torch.zeros(num_classes)
    FP = torch.zeros(num_classes)
    FN = torch.zeros(num_classes)

    for cls in range(num_classes):
        TP[cls] = ((preds == cls) & (targets == cls)).sum().item()
        FP[cls] = ((preds == cls) & (targets != cls)).sum().item()
        FN[cls] = ((preds != cls) & (targets == cls)).sum().item()

    precision = TP / (TP + FP + 1e-8)
    recall = TP / (TP + FN + 1e-8)
    f1 = 2 * precision * recall / (precision + recall + 1e-8)
    accuracy = (preds == targets).sum().item() / len(targets)

    macro_precision = precision.mean().item()
    macro_recall = recall.mean().item()
    macro_f1 = f1.mean().item()

    print(f"TP: {TP}")
    print(f"FP: {FP}")
    print(f"FN: {FN}")

    return {
        "accuracy": accuracy,
        "macro_precision": macro_precision,
        "macro_recall": macro_recall,
        "macro_f1": macro_f1
    }
```

Model Training Performance: The average loss per epoch during the training phase is presented in the table below. A consistent decrease in loss indicates that the model was learning effectively.

```
Epoch 1/10 | Avg Loss: 0.2460
Epoch 2/10 | Avg Loss: 0.0778
Epoch 3/10 | Avg Loss: 0.0574
Epoch 4/10 | Avg Loss: 0.0458
Epoch 5/10 | Avg Loss: 0.0368
Epoch 6/10 | Avg Loss: 0.0312
Epoch 7/10 | Avg Loss: 0.0264
Epoch 8/10 | Avg Loss: 0.0225
Epoch 9/10 | Avg Loss: 0.0190
Epoch 10/10 | Avg Loss: 0.0173
```

Epoch	Average Loss
1	0,2680
2	0,0778
3	0,0574
4	0,0458
5	0,0368
6	0,0312
7	0,0264
8	0,0225
9	0,0190
10	0,0173

Test Performance Evaluation:

```
📊 Digit Classification Metrics:
Accuracy       : 98.86%
Macro F1       : 0.9885
Precision      : 0.9886
Recall         : 0.9884

📊 Rotation Classification Metrics:
Accuracy       : 99.59%
Macro F1       : 0.9959
Precision      : 0.9959
Recall         : 0.9959
```

```python
def main():
    #4 set the hyperparameters
    input_size= 784 #28x28
    num_classes = 6 #0-9
    learning_rate = 0.001
    batch_size= 64
    num_epochs=10

    #3 set the device
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Dataset ve DataLoader
    train_dataset = CustomRotatedMNIST(train=True)
    test_dataset = CustomRotatedMNIST(train=False)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    # Model
    model = NN(input_size=input_size, num_classes=num_classes)

    # Loss ve Optimizer
    criterion_digit = nn.CrossEntropyLoss()
    criterion_rotation = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Eğitim
    train(model, train_loader, optimizer, criterion_digit, criterion_rotation, device, num_epochs)

    # Test
    evaluate(model, test_loader, device)

    # Modeli Kaydet
    torch.save(model.state_dict(), "digit_rotation_model.pth")
    print("\n✅ Model saved as digit_rotation_model.pth")


# 🔁 Giriş noktası (özellikle .py dosyası olarak çalıştırırken önemlidir)
if __name__ == "__main__":
    main()
```

```
Epoch 1/10 | Avg Loss: 0.2460
Epoch 2/10 | Avg Loss: 0.0778
Epoch 3/10 | Avg Loss: 0.0574
Epoch 4/10 | Avg Loss: 0.0458
Epoch 5/10 | Avg Loss: 0.0368
Epoch 6/10 | Avg Loss: 0.0312
Epoch 7/10 | Avg Loss: 0.0264
Epoch 8/10 | Avg Loss: 0.0225
Epoch 9/10 | Avg Loss: 0.0190
Epoch 10/10 | Avg Loss: 0.0173
TP: tensor([4527., 4070., 3976., 3900., 3513., 4053.])
FP: tensor([51., 47., 21., 43., 41., 74.])
FN: tensor([13., 58., 64., 28., 55., 59.])
TP: tensor([6048., 6061., 6054., 6054.])
FP: tensor([18., 30., 25., 26.])
FN: tensor([31., 18., 25., 25.])

📊 Digit Classification Metrics:
Accuracy      : 98.86%
Macro F1      : 0.9885
Precision     : 0.9886
Recall        : 0.9884

📊 Rotation Classification Metrics:
Accuracy      : 99.59%
Macro F1      : 0.9959
Precision     : 0.9959
Recall        : 0.9959

✅ Model saved as digit_rotation_model.pth
```

Conclusion:

This project successfully demonstrated the effectiveness of a multi-task learning approach for simultaneously classifying digits and their rotation. The developed CNN model achieved outstanding accuracy on both tasks, confirming that a single network can efficiently learn a shared representation suitable for extracting both content-based and context-based information from images.

References:

- [Pytorch CNN example (Convolutional Neural Network)]