CSC671 Deep Learning

HANDWRITTEN DIGIT RECOGNITION

Convolutional Neural Network

By

Yan Peng

CONTENT

- Introduction
- Dataset Preparation
- Model Architecture
- Training the Model
- Model Evaluation
- Visualizing Training Result
- Real Time Prediction
- Conclusion

Introduction

This project not only describes an approach for efficiently recognize digit numbers written by different people with the help convolution neural network (CNN), taking into different writing style and ink color.

For real-time prediction we also build a nice user interface to draw digits on a canvas and then the model will predict the digit.

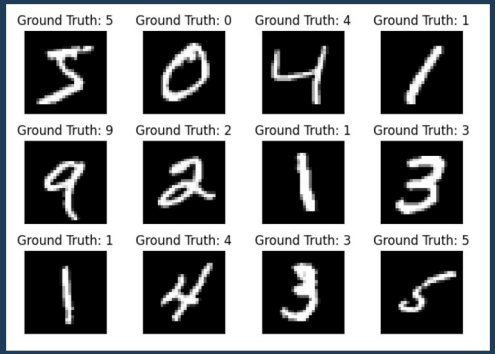
The MNIST dataset consists of 60,000 training examples and 10,000 test examples, primarily used for the training and evaluation of supervised machine learning algorithms designed for recognizing handwritten digits.

Dataset Preparation

```
# convert the image to tensor,and normalize it
transform = transforms.Compose([ transforms.ToTensor(),
                                 transforms.Normalize((0.1307,), (0.3081,))])
# download dataset
train_set = datasets.MNIST(root='./MNIST', train=True,
download=True, transform=transform)

test_set = datasets.MNIST(root='./MNIST', train=False,
download=True, transform=transform)

# load dataset
train_loader=DataLoader(train_set,batch_size=batch_siz
e,shuffle=True)
test_loader=DataLoader(test_set,batch_size=batch_size,shuffle
=False)
```

```
Shape of X [N, C, H, W]: torch.Size([256, 1, 28, 28])
Shape of y: torch.Size([256]) torch.int64
```

Sample image from the dataset.



Each Image size: 28*28 pixels

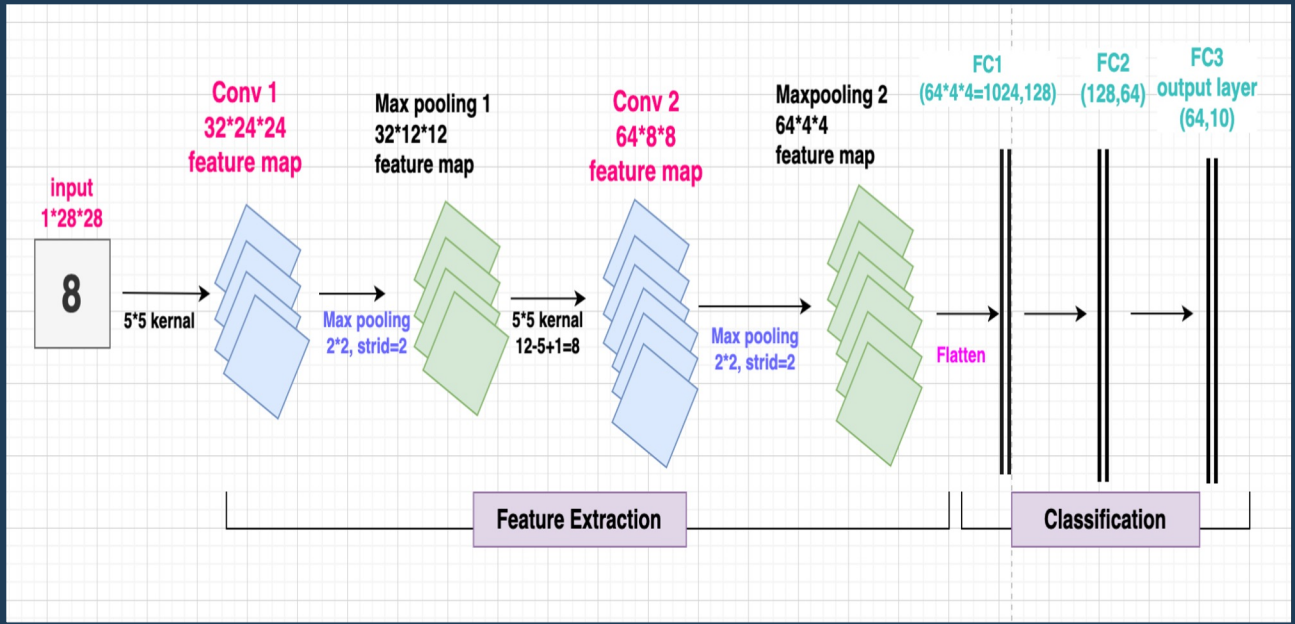
```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 224 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 224 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 254 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 254 254 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  17 254 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0 254 254 254 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0 161 254 254 254 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0 254 254 254  31 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 254 254 254 254   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 254 254 254   0   0 254 254 254 131 131 131 131   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0 254 254 254 254 254 254 254 254 254 254 254 254 254   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0 239 254 254 254 254 254 254 254 254 254 254 254 254 254   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0 254 254 254 254 254 254 254 254 254 254 123 123 123 123   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0  93 254 254 252   0   0   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 191 254 254 127   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 254 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 237 254 254   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

Model Architecture(CNN)

CNN were specially designed for image data and have multiple layers that processes the image, extracts features and classifies to correct class.

- **Convolution layer:** it consists several filters that perform feature extraction.
- **ReLU**: non-linearity in our ConvNet.
- **Pooling layer**: down-sampling operation that reduces the dimensions of the feature map. (Here we have used the Max Pooling).
- **Fully connected layer**: forms when the flattened matrix from the pooling layer is fed as an input, which classifies and identifies the images.

Model Architecture(CNN)



convolutional formular:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

Model Architecture(CNN) -- code

```python
class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d( in_channels: 1,  out_channels: 32,  kernel_size: 5), # in_channels, out_channels, kernel_size
            nn.ReLU(),
            nn.MaxPool2d( kernel_size: 2,  stride: 2), # kernel_size, stride
            nn.Conv2d( in_channels: 32,  out_channels: 64,  kernel_size: 5),
            nn.ReLU(),
            nn.MaxPool2d( kernel_size: 2,  stride: 2)
        )

        self.fc = nn.Sequential(
            nn.Linear(64*4*4,  out_features: 128),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear( in_features: 128,  out_features: 64),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear( in_features: 64,  out_features: 10)
        )
    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output
```

Output of the Structure of the Model

```
MyNet(
  (conv): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Sequential(
    (0): Linear(in_features=1024, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

Training the Model

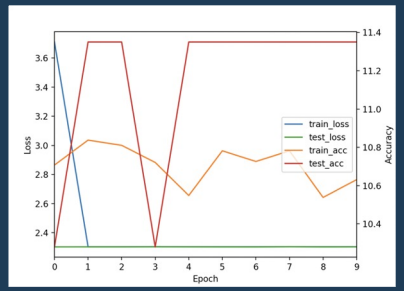
```python
def train(dataloader, model, loss_fn, optimizer):
    model.train()
    # Initialize variables to track the total number of samples, correct predictions, and epoch loss
    total = 0
    correct = 0.0
    epoch_loss = 0.0
    # Iterate through batches in the dataloader
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction error
        pred = model(X)
        # Calculate the loss between the predicted values and the actual labels
        loss = loss_fn(pred, y)
        # Accumulate the loss for the entire epoch
        epoch_loss += loss.item()
        # Get the predicted class by finding the index of the maximum value along the second dimension
        predicted=pred.argmax(dim=1)
        # Update total number of samples and correct predictions
        total += y.size(0)
        correct += (predicted==y).sum().item()
        # Backpropagation: Zero the gradients, perform backward pass, and update the model parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print training progress for every K batches (in this case, 1/10th of the dataset)
        L = len(dataloader)
        K = L // 10
        if batch % K == 0:
            accuracy = 100 * correct / total
            print(f"train loss:{loss.item():.3f}", end=" ")
            print(f"train accuracy:{accuracy:.2f}%")
    # Calculate the average loss and accuracy for the entire epoch
    epoch_loss /= len(dataloader)
    return epoch_loss, accuracy
```

Training the Model

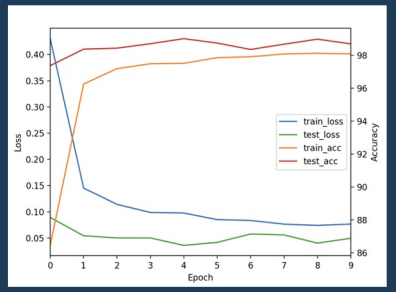
1) Different learning rate:

learning rate	train loss	train accuracy
0.05	2.300	10.83%
0.005	0.063	98.39%
0.001	0.026	98.99%

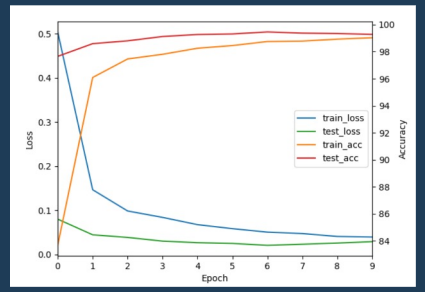
learning rate=0.05



learning rate=0.005



learning rate=0.001
is the best result



2) Different optimization algorithm

optimization algorithm	train loss	train accuracy	epoch
SGD	0.760	74.07%	25
Adam	0.026	98.99%	10

Adam often converges faster in terms of the number of epochs, which might explain why the good results with fewer epochs compared to SGD.

The difference in performance between the SGD optimizer and the Adam optimizer could be influenced by several factors, eg: Momentum,Batch Size,Overfitting, or Learning rate.

Testing the Model

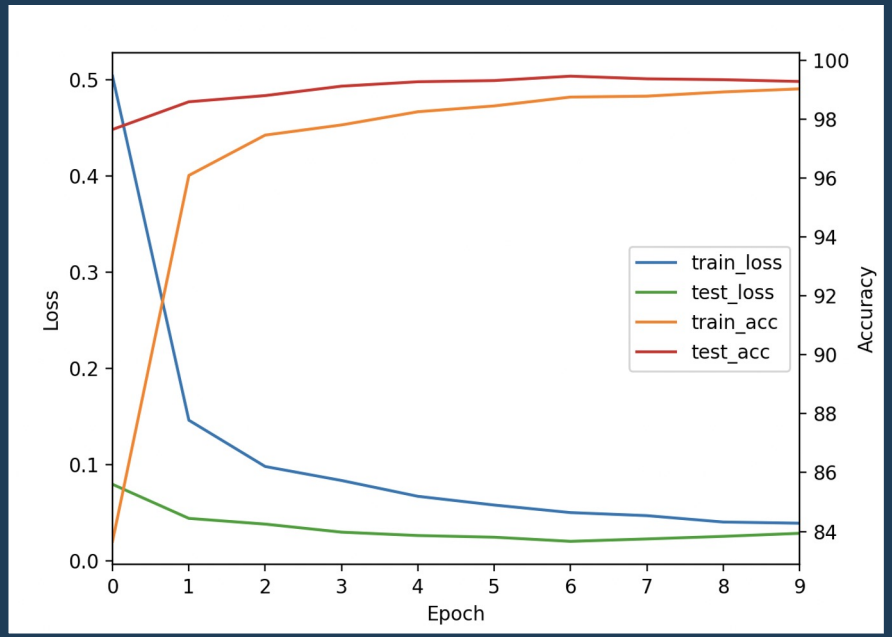
Test Results:

learning rate	test loss	test accuracy
0.05	0.009	9.58%
0.005	0.000	99.03%
0.001	0.000	99.25%

optimization algorithm	test loss	test accuracy	epoch
SGD	0.002	89.19%	25
Adam	0.000	99.25%	10

```python
# Testing CNN and calculate the accuracy and save the best model
def test(testloader,model,loss_fn):
    model.eval()
    # Initialize variables to track the test loss, correct predictions, and total samples
    test_loss=0
    correct=0
    total=0
    with torch.no_grad():
        # Iterate through batches in the testloader.(X is the image and y is the label)
        for X,y in testloader:
            # Compute predictions using the model
            pred=model(X)
            # Calculate the loss between the predicted values and the actual labels
            test_loss+=loss_fn(pred,y).item()
            # Get the predicted class by finding the index of the maximum value along the second dimension
            predicted=pred.argmax(dim=1)

            # Update total number of samples and correct predictions
            total+=y.size(0)
            correct+=(predicted==y).sum().item()

        print("test loss:{:.3f},test accuracy:{:.2f}%".format( *args: test_loss/total,100*correct/total))

    # Calculate the average loss and accuracy for the entire test set
    test_loss /= len(testloader)
    accuracy = 100 * correct / total
    return test_loss, accuracy
```

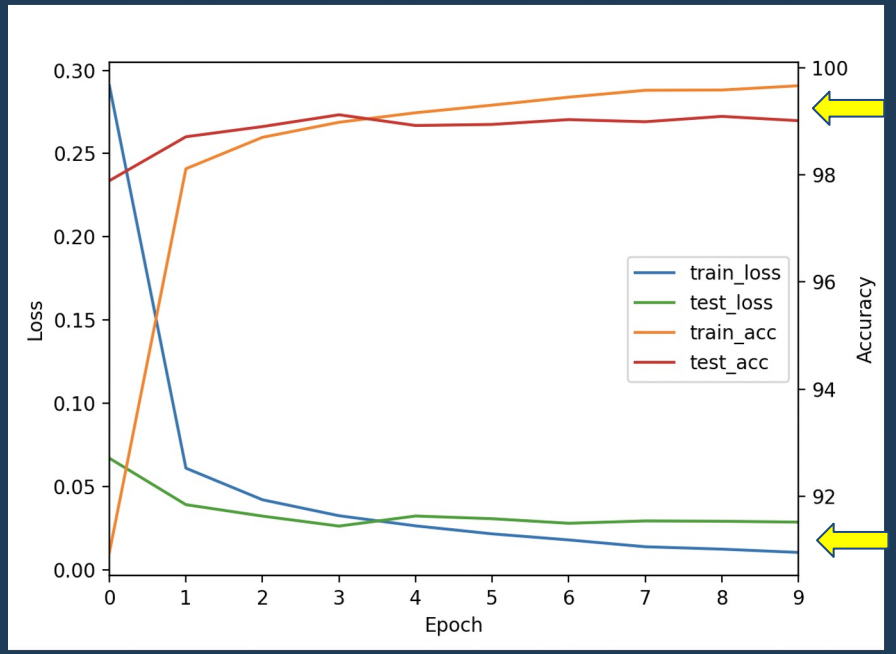
Visualizing Training & Testing Result with Dropout



test accuracy is slightly higher than train accuracy.

test loss is slightly lower than train loss.

Visualizing Training & Testing Result without Dropout

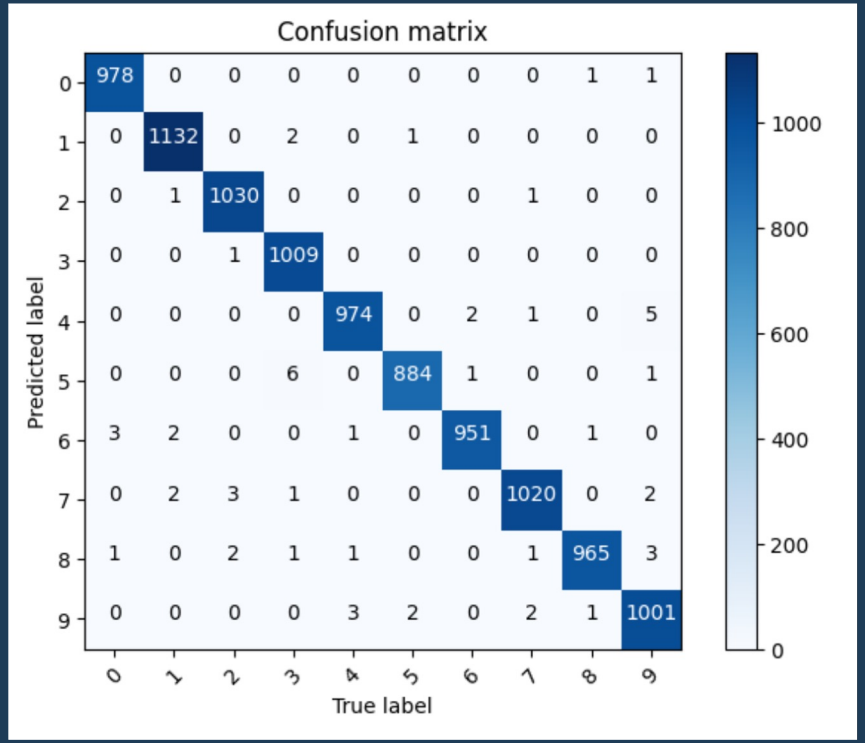


train accuracy is much higher than test accuracy.

Notice how the test loss for without-dropout run diverges a little after just a few epochs

trian loss is much lower than test loss.

Visualizing Testing Result - Confusion Matrix



row is true label, column is predicted label
e.g. 0 is classified as 0, 978 times out of 980.(99%)
diagonal shows that most digits are classified correctly
off diagonal numbers show misclassified digits
e.g. 3 and 5 or 4 and 9 are often misclassified as each other.

```
y_true = []
y_pred = []
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        y_true += labels.tolist()
        y_pred += predicted.tolist()

confusion_mat = confusion_matrix(y_true, y_pred)
print(confusion_mat)
```

Real-Time Prediction

command:
streamlit run hello_streamlit.py

Conclusion

In this project, the variations of accuracies for handwritten digit were observed for 10 epochs by varying the hidden layers using CNN model and MNIST dataset.

The project achieved high accuracy in recognizing handwritten digits, with the best model reaching 99.25% accuracy. And the total lowest test loss is 0.000.

Thank You