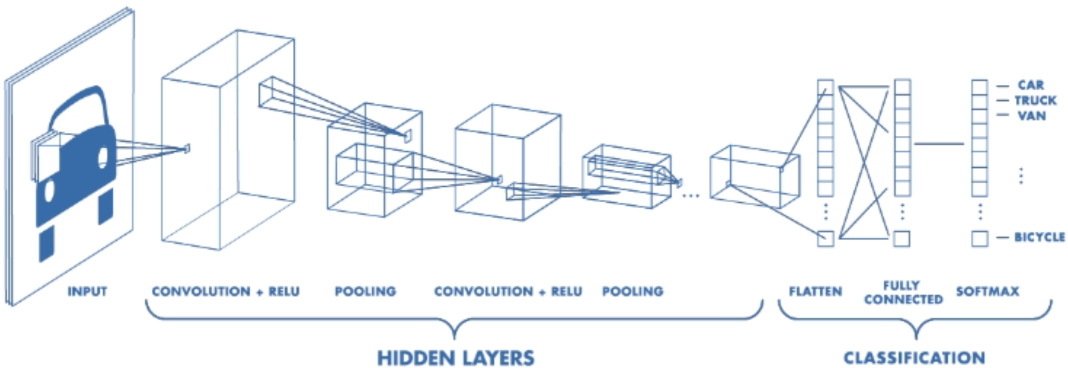
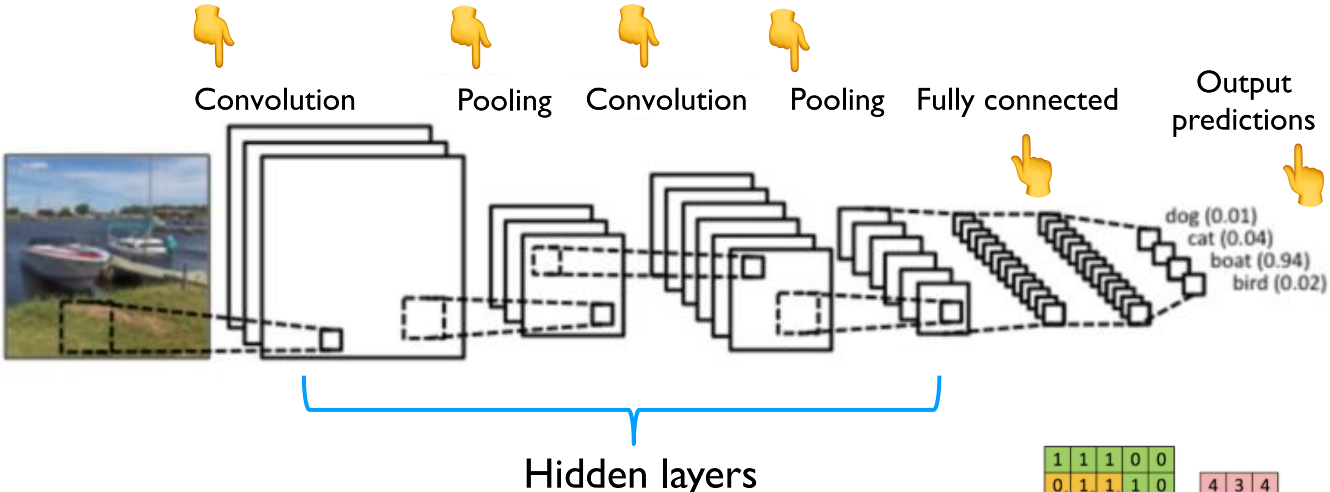
A Basic Introduction to Convolution Neural Network
-- Training An Image Classifier via PyTorch

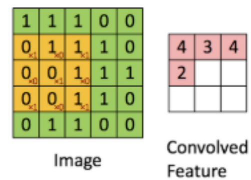


Rachel Chen

2020/11/26

Review: CNN Architecture





- **Convolution layer** (卷積層) : the core building block of CNN; it extracts features from input images.

- **Pooling layer** (池化層): extracts particular values from a input set; this reduce the number of parameters, memory footprint and computations, so it can control overfitting.

- **Fully connected layer** (完全連接層): the last block of CNN, related to the task of classification.

PyTorch

Deep Learning in Python!

PyTorch is a python package that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Deep Neural Networks built on a tape-based autograd system

Linear, Logistic, softmax models
DNN: Deep Neural Net
CNN: Convolutional Neural Net
RNN: Recurrent Neural Net

We can build state-of-the-art deep learning models with python via Pytorch! 😎

Say "Hello World" to the world of deep learning!

Training an image classifier via PyTorch

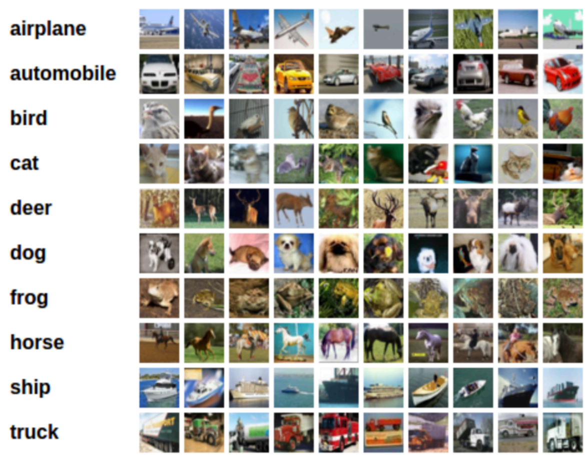
Data set: CIFAR10

Mainly used package: torchvision (for image)

Goal:

Understanding PyTorch's Tensor library and neural networks at a high level.

Train a small neural network to classify images.



Training an image classifier via PyTorch

We will do the following steps in order:

1. Load and normalizing the CIFAR10 and test datasets using torchvision
2. Define a Convolutional Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data

1 Load and normalizing the CIFAR10 and test datasets using torchvision

```python
#1. Loading and normalizing CIFAR10.
import torch
from torch.utils.data import Dataset, TensorDataset # Heart of PyTorch data loading utility
import torchvision # Use torchvision to load CIFAR10.
import torchvision.transforms as transforms # Use transforms to transform image.
```

Import libraries we need

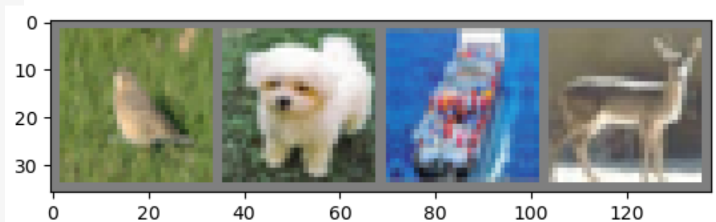
```python
# Use transforms.Compose to chain transforms together
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# Load data
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

The output of torchvision datasets are PIL images of range [0, 1]

We transform them to Tensors of normalized range [-1, 1]

```python
#Let us show some of the training images just for fun
if __name__ == '__main__':
    import numpy as np
    from matplotlib import pyplot as plt

    def imshow(img):
        img = img / 2 + 0.5     # unnormalize
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()


    # Get some random training images
    dataiter = iter(trainloader)
    images, labels = dataiter.next()

    # Show images
    imshow(torchvision.utils.make_grid(images))
    # Print labels
    print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

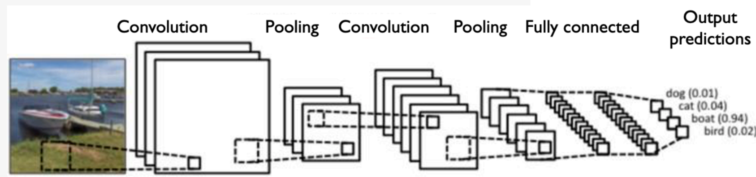
Show some of the training images just for fun



2 Define a Convolutional Neural Network

```python
# 2. Define a Convolutional Neural Network
import torch.nn as nn # Use torch.nn to construct a neural network
import torch.nn.functional as F

# Take 3-channel images
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv layers: 3 input image channel, 6 output channels, kernal size of 5x5 square convolution
        self.conv1 = nn.Conv2d(3, 6, 5) # Apply a 1D convlution over an input signal composed of several iput planes.
        self.conv2 = nn.Conv2d(6, 16, 5) # Apply a 2D convlution over an input signal composed of several iput planes.

        # Pooling layer: maximum pooling over a 2 x 2 window (kernel size = 2, stride = 2)
        self.pool = nn.MaxPool2d(2, 2) #Applies a 2D max pooling over an input signal composed of several input planes.

        # Linear layers: Applya linear transformation to the incoming data: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120) #torch.nn.Linear(in_features: int, out_features: int, bias: bool = True)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # ReLU layer is a non-linear activation function to give out the final value from a neuron, the result gives a range from 0 to infinity.
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))

        # Reshape the tensor: return a new tensor with same data as the self tensor but of a different shape.
        x = x.view(-1, 16 * 5 * 5) # -1: automatically calculate dimension

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

net = Net()
print(net)
```

<u>Rectified Linear Unit (ReLU)</u> Activation function (激勵函數)




```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

3 Define a loss function and optimizer

Let's use a Cross-Entropy loss function and SGD* with momentum as the optimizer.

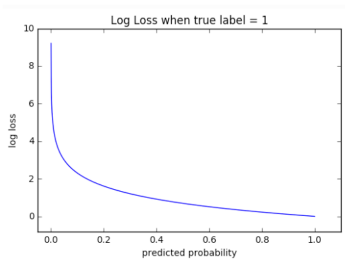
```
# 3. Define a Loss function and optimizer
import torch.optim as optim

# Use a Classification Cross-Entropy loss and SGD with momentum as our optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

*Stochastic Gradient Descent
(隨機梯度下降法 SGD)

◆ Loss Function (損失函數)

- A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.
- Frequently used loss functions include log, hinge, MSE, etc.

➡ Cross-Entropy loss (交叉熵損失函数, or log loss)



$$-(y\log(p)+(1-y)\log(1-p))$$

$$\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$

◆ Optimizer (優化器)

- The engine of machine learning — they make the computer learn.
- Frequently used optimizers include SGD, Adam, Adadelta, etc.

4 Train the network on the training data

```python
# 4. Train the network
def train():
    torch.multiprocessing.freeze_support()
    for epoch in range(10):  # Loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # Get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')

    PATH = './cifar_net.pth'
    torch.save(net.state_dict(), PATH) # Save our trained model


if __name__ == "__main__":
    train()
```

➡

```
[1,  2000] loss: 2.205
[1,  4000] loss: 1.847
[1,  6000] loss: 1.657
[1,  8000] loss: 1.579
[1, 10000] loss: 1.493
[1, 12000] loss: 1.450

[2,  2000] loss: 1.382
[2,  4000] loss: 1.337
[2,  6000] loss: 1.320
[2,  8000] loss: 1.307
[2, 10000] loss: 1.282
[2, 12000] loss: 1.273

[3,  2000] loss: 1.182
[3,  4000] loss: 1.182
[3,  6000] loss: 1.182
[3,  8000] loss: 1.185
[3, 10000] loss: 1.156
[3, 12000] loss: 1.149

[4,  2000] loss: 1.078
[4,  4000] loss: 1.063
[4,  6000] loss: 1.080
[4,  8000] loss: 1.086
[4, 10000] loss: 1.081
[4, 12000] loss: 1.093

[5,  2000] loss: 0.993
[5,  4000] loss: 0.986
[5,  6000] loss: 1.033
[5,  8000] loss: 1.003
[5, 10000] loss: 1.022
[5, 12000] loss: 1.027

Finished Training
```

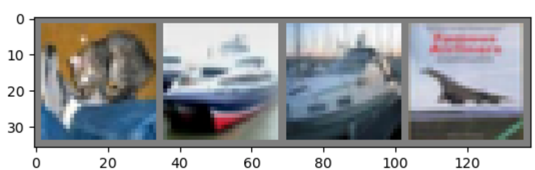
5 Test the network on the test data - 1/2

🔷 We have trained the network for 5 passes over the training dataset. But we need to check if the network has learnt anything at all.

🔷 We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

```
# 5. Test the network on the test data
    dataiter = iter(testloader)
    images, labels = dataiter.next()

    # Print some images from the test data
    import numpy as np
    from matplotlib import pyplot as plt
    def imshow(img):
        img = img / 2 + 0.5     # unnormalize
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()

    imshow(torchvision.utils.make_grid(images))
    print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

➡️



Out:

```
GroundTruth:    cat  ship  ship plane
```

5️⃣ Test the network on the test data – 2/2

Okay, now let us see what the neural network thinks these examples above are:

```python
# Load back
net = Net()
PATH = './cifar_net.pth'
net.load_state_dict(torch.load(PATH))

# See what our neural network thinks these images above are:
outputs = net(images)
print(outputs)
```

➡️

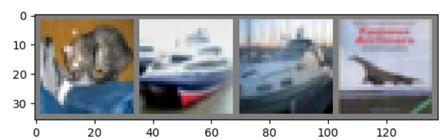
```
tensor([[ 0.5661, -1.2947,  1.4115,  1.0806, -0.1390, -0.6296,  0.5167, -1.1709,
          0.7995, -1.1832],
        [ 7.1678,  4.9775, -0.9476, -4.4104, -3.5120, -7.3618, -6.1025, -5.5900,
          8.5631,  4.4473],
        [ 3.4766,  2.4226, -0.3608, -1.8400, -2.8869, -3.5260, -4.8038, -1.6884,
          4.7213,  3.3924],
        [ 4.6499,  0.0174,  1.9208, -2.3537, -0.1950, -4.0568, -4.2183, -1.6824,
          4.7193,  0.0959]], grad_fn=<AddmmBackward>)
```

🔷 The outputs are energies for the 10 classes. The higher the energy for a class, the more the network thinks that the image is of the particular class.

So, let's get the index of the highest energy:

Out:

```
Predicted:    cat  ship plane plane
```



The results seem pretty good! 🤩

5 Test the network on the test data – 2/2

Let us look at how the network performs on the whole dataset.

```python
# Look how our cnn performs on the whole dataset:
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

➡

```
Accuracy of the network on the 10000 test images: 62 %
```

What are the classes that performed well, and the classes that did not perform well?

```python
# Check every classes' performance:
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

➡

```
Accuracy of plane : 62 %
Accuracy of   car : 64 %
Accuracy of  bird : 44 %
Accuracy of   cat : 41 %
Accuracy of  deer : 57 %
Accuracy of   dog : 61 %
Accuracy of  frog : 73 %
Accuracy of horse : 71 %
Accuracy of  ship : 69 %
Accuracy of truck : 77 %
```

What happen if I train more epochs? 🤔

Result of 2 epochs

```
[2,  2000] loss: 1.391
[2,  4000] loss: 1.362
[2,  6000] loss: 1.342
[2,  8000] loss: 1.331
[2, 10000] loss: 1.302
[2, 12000] loss: 1.254

Finished Training

Accuracy of the network on the 10000 test images: 55 %

Accuracy of plane : 65 %
Accuracy of   car : 70 %
Accuracy of  bird : 28 %
Accuracy of   cat : 26 %
Accuracy of  deer : 43 %
Accuracy of   dog : 54 %
Accuracy of  frog : 71 %
Accuracy of horse : 62 %
Accuracy of  ship : 69 %
Accuracy of truck : 67 %
```

Result of 5 epochs

```
[5,  2000] loss: 0.993
[5,  4000] loss: 0.986
[5,  6000] loss: 1.033
[5,  8000] loss: 1.003
[5, 10000] loss: 1.022
[5, 12000] loss: 1.027

Finished Training

Accuracy of the network on the 10000 test images: 62 %

Accuracy of plane : 62 %
Accuracy of   car : 64 %
Accuracy of  bird : 44 %
Accuracy of   cat : 41 %
Accuracy of  deer : 57 %
Accuracy of   dog : 61 %
Accuracy of  frog : 73 %
Accuracy of horse : 71 %
Accuracy of  ship : 69 %
Accuracy of truck : 77 %
```

Result of 10 epochs

```
[10,  2000] loss: 0.788
[10,  4000] loss: 0.826
[10,  6000] loss: 0.843
[10,  8000] loss: 0.858
[10, 10000] loss: 0.858
[10, 12000] loss: 0.849

Finished Training

Accuracy of the network on the 10000 test images: 62 %

Accuracy of plane : 70 %
Accuracy of   car : 80 %
Accuracy of  bird : 49 %
Accuracy of   cat : 46 %
Accuracy of  deer : 46 %
Accuracy of   dog : 56 %
Accuracy of  frog : 70 %
Accuracy of horse : 67 %
Accuracy of  ship : 69 %
Accuracy of truck : 72 %
```

https://github.com/rachelpeichen/PyTorch_CIFAR10

👍 Self-learning resources for programming

English

- Python for everybody (provided by University of Michigan) via Coursera.com
- Machine learning (Andrew-Ng, Sandford, Google Brain) via Coursera.com
- Data Science (provided by IBM corp.) via Coursera.com

Chinese

- Python courses lectured in Chinese via Udemy.com
- 輕鬆學Python 3 零基礎彩色圖解、專業入門

If you need a teacher, I highly recommend the courses provided by NTU CSIE!

- 台灣大學資訊系統訓練班

https://github.com/rachelpeichen/Interactive_Pytorch_MNIST

PyTorch_MNIST — -bash — 77×60

```
Train Epoch: 10 [24320/60000 (41%)]     Loss: 0.004441
Train Epoch: 10 [24960/60000 (42%)]     Loss: 0.000221
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.015975
Train Epoch: 10 [26240/60000 (44%)]     Loss: 0.003772
Train Epoch: 10 [26880/60000 (45%)]     Loss: 0.039793
Train Epoch: 10 [27520/60000 (46%)]     Loss: 0.019310
Train Epoch: 10 [28160/60000 (47%)]     Loss: 0.084274
Train Epoch: 10 [28800/60000 (48%)]     Loss: 0.004663
Train Epoch: 10 [29440/60000 (49%)]     Loss: 0.025823
Train Epoch: 10 [30080/60000 (50%)]     Loss: 0.020883
Train Epoch: 10 [30720/60000 (51%)]     Loss: 0.002969
Train Epoch: 10 [31360/60000 (52%)]     Loss: 0.003411
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.004482
Train Epoch: 10 [32640/60000 (54%)]     Loss: 0.005340
Train Epoch: 10 [33280/60000 (55%)]     Loss: 0.015296
Train Epoch: 10 [33920/60000 (57%)]     Loss: 0.000488
Train Epoch: 10 [34560/60000 (58%)]     Loss: 0.021868
Train Epoch: 10 [35200/60000 (59%)]     Loss: 0.141543
Train Epoch: 10 [35840/60000 (60%)]     Loss: 0.058963
Train Epoch: 10 [36480/60000 (61%)]     Loss: 0.014396
Train Epoch: 10 [37120/60000 (62%)]     Loss: 0.005167
Train Epoch: 10 [37760/60000 (63%)]     Loss: 0.070762
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.072799
Train Epoch: 10 [39040/60000 (65%)]     Loss: 0.003566
Train Epoch: 10 [39680/60000 (66%)]     Loss: 0.005654
Train Epoch: 10 [40320/60000 (67%)]     Loss: 0.027527
Train Epoch: 10 [40960/60000 (68%)]     Loss: 0.058061
Train Epoch: 10 [41600/60000 (69%)]     Loss: 0.008346
Train Epoch: 10 [42240/60000 (70%)]     Loss: 0.032695
Train Epoch: 10 [42880/60000 (71%)]     Loss: 0.000944
Train Epoch: 10 [43520/60000 (72%)]     Loss: 0.068977
Train Epoch: 10 [44160/60000 (74%)]     Loss: 0.001548
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.048027
Train Epoch: 10 [45440/60000 (76%)]     Loss: 0.023103
Train Epoch: 10 [46080/60000 (77%)]     Loss: 0.021393
Train Epoch: 10 [46720/60000 (78%)]     Loss: 0.044896
Train Epoch: 10 [47360/60000 (79%)]     Loss: 0.030724
Train Epoch: 10 [48000/60000 (80%)]     Loss: 0.004800
Train Epoch: 10 [48640/60000 (81%)]     Loss: 0.002677
Train Epoch: 10 [49280/60000 (82%)]     Loss: 0.021818
Train Epoch: 10 [49920/60000 (83%)]     Loss: 0.023337
Train Epoch: 10 [50560/60000 (84%)]     Loss: 0.063333
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.147943
Train Epoch: 10 [51840/60000 (86%)]     Loss: 0.004389
Train Epoch: 10 [52480/60000 (87%)]     Loss: 0.000850
Train Epoch: 10 [53120/60000 (88%)]     Loss: 0.041578
Train Epoch: 10 [53760/60000 (90%)]     Loss: 0.086868
Train Epoch: 10 [54400/60000 (91%)]     Loss: 0.042052
Train Epoch: 10 [55040/60000 (92%)]     Loss: 0.007098
Train Epoch: 10 [55680/60000 (93%)]     Loss: 0.008708
Train Epoch: 10 [56320/60000 (94%)]     Loss: 0.021191
Train Epoch: 10 [56960/60000 (95%)]     Loss: 0.001754
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.017068
Train Epoch: 10 [58240/60000 (97%)]     Loss: 0.001016
Train Epoch: 10 [58880/60000 (98%)]     Loss: 0.000250
Train Epoch: 10 [59520/60000 (99%)]     Loss: 0.000165

Test set: Average loss: 0.0263, Accuracy: 9920/10000 (99%)
```