DeepLearning1_Supervised_Pytoch (1)

May 10, 2025

1 Convolutional Neural Networks

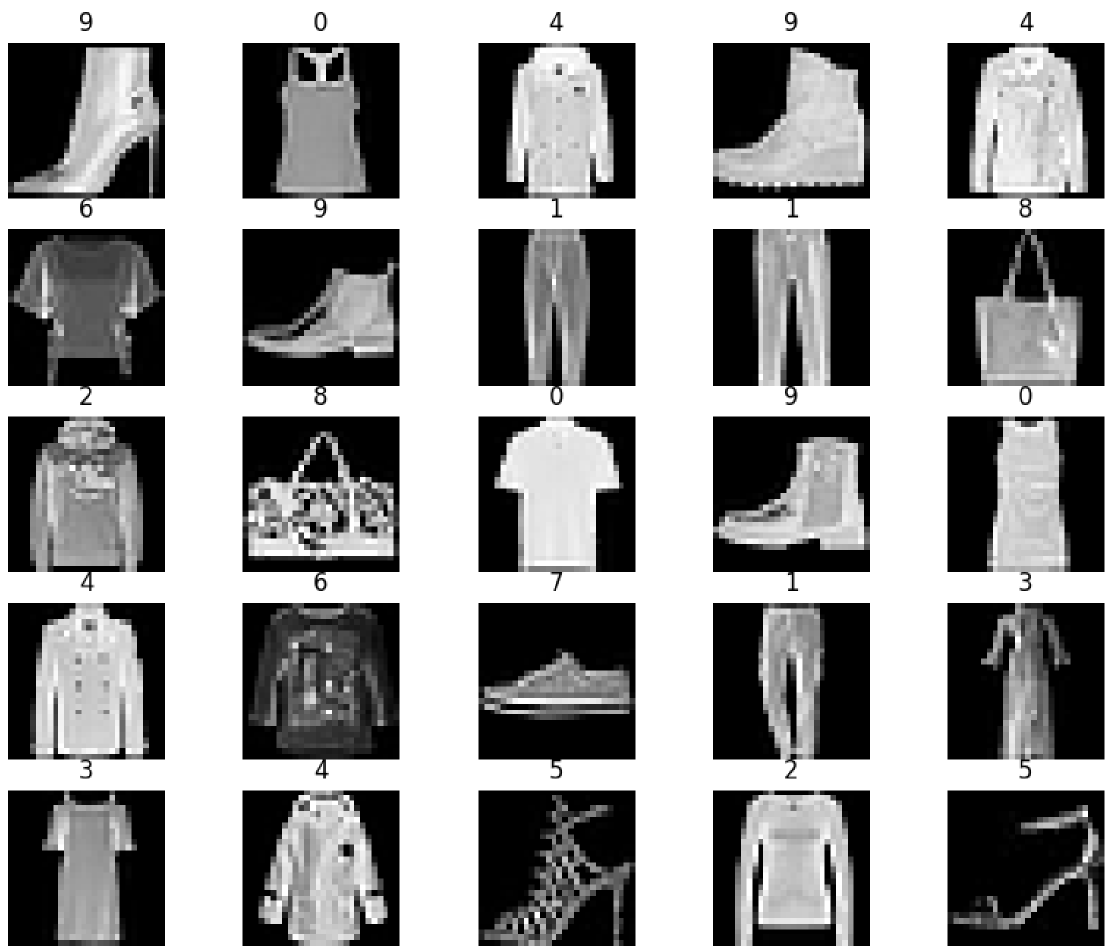
[1]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# PyTorch TensorBoard support
# from torch.utils.tensorboard import SummaryWriter
# import torchvision
# import torchvision.transforms as transforms

from datetime import datetime

import torchvision
import torchvision.transforms as transforms

from torchvision.datasets import FashionMNIST

import matplotlib.pyplot as plt
%matplotlib inline

from torch.utils.data import random_split
from torch.utils.data import DataLoader
import torch.nn.functional as F

from PIL import Image
#import torchvision.transforms as T
```

[2]:
```
mnist_dataset = FashionMNIST(root='data/', download=True, train=True,␣
↪transform=transforms.ToTensor())
print(mnist_dataset)
```

```
Dataset FashionMNIST
    Number of datapoints: 60000
    Root location: data/
    Split: Train
    StandardTransform
```

```
Transform: ToTensor()
```

[3]:
```python
# Print multiple images at once
figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(mnist_dataset), size=(1,)).item()
    img, label = mnist_dataset[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



[4]:
```python
train_data, validation_data = random_split(mnist_dataset, [50000, 10000])
## Print the length of train and validation datasets
print("length of Train Datasets: ", len(train_data))
```

```
print("length of Validation Datasets: ", len(validation_data))

batch_size = 128
train_loader = DataLoader(train_data, batch_size, shuffle = True)
val_loader = DataLoader(validation_data, batch_size, shuffle = False)
## MNIST data from pytorch already provides held-out test set!
```

```
length of Train Datasets:  50000
length of Validation Datasets:  10000
```

[5]:
```
## Basic set up for a logistic regression model (won't be used in practice or␣
↪for training)
input_size = 28 * 28
num_classes = 10
```

[6]:
```
# accuracy calculation
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim = 1)
    return(torch.tensor(torch.sum(preds == labels).item()/ len(preds)))
```

[7]:
```
# We put all of the above:
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return(out)

    # We add extra methods
    def training_step(self, batch):
        # when training, we compute the cross entropy, which help us update␣
↪weights
        images, labels = batch
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        return(loss)

    def validation_step(self, batch):
        images, labels = batch
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        # in validation, we want to also look at the accuracy
        # idealy, we would like to save the model when the accuracy is the␣
↪highest.
```

```python
        acc = accuracy(out, labels) ## calculate metrics/accuracy
        return({'val_loss':loss, 'val_acc': acc})

    def validation_epoch_end(self, outputs):
        # at the end of epoch (after running through all the batches)
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return({'val_loss': epoch_loss.item(), 'val_acc' : epoch_acc.item()})

    def epoch_end(self, epoch,result):
        # log epoch, loss, metrics
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, 
 ↪result['val_loss'], result['val_acc']))

# we instantiate the model
model = MnistModel()

# a simple helper function to evaluate
def evaluate(model, data_loader):
    # for batch in data_loader, run validation_step
    outputs = [model.validation_step(batch) for batch in data_loader]
    return(model.validation_epoch_end(outputs))

# actually training
def fit(epochs, lr, model, train_loader, val_loader, opt_func = torch.optim.
 ↪SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        ## Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward() ## backpropagation starts at the loss and goes 
 ↪through all layers to model inputs
            optimizer.step() ## the optimizer iterate over all parameters 
 ↪(tensors); use their stored grad to update their values
            optimizer.zero_grad() ## reset gradients

        ## Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return(history)
```

[8]:
```python
# test the functions, with a randomly initialized model (weights are random, e.
↪g., untrained)
result0 = evaluate(model, val_loader)
result0
```

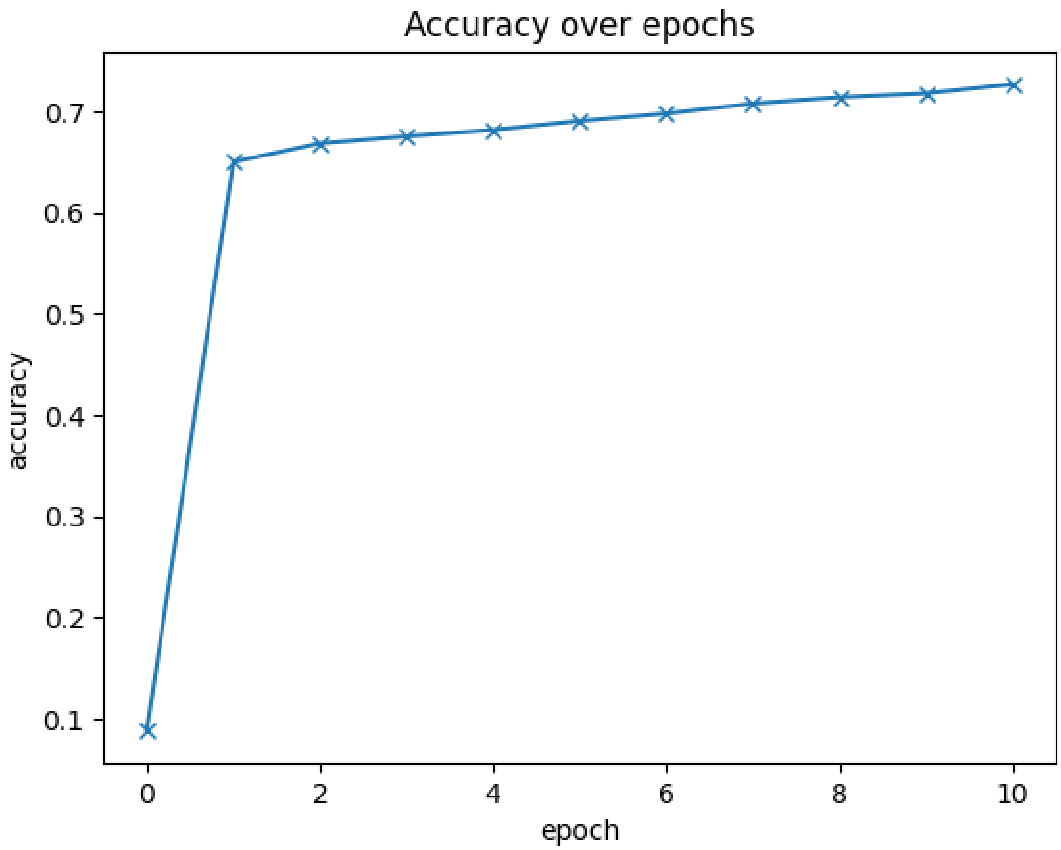
[8]: {'val_loss': 2.3385744094848633, 'val_acc': 0.08900316804647446}

[9]:
```python
# let's train for 10 epochs
history1 = fit(10, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 1.7122, val_acc: 0.6501
Epoch [1], val_loss: 1.4169, val_acc: 0.6684
Epoch [2], val_loss: 1.2479, val_acc: 0.6753
Epoch [3], val_loss: 1.1392, val_acc: 0.6817
Epoch [4], val_loss: 1.0635, val_acc: 0.6906
Epoch [5], val_loss: 1.0072, val_acc: 0.6979
Epoch [6], val_loss: 0.9637, val_acc: 0.7077
Epoch [7], val_loss: 0.9285, val_acc: 0.7139
Epoch [8], val_loss: 0.8995, val_acc: 0.7179
Epoch [9], val_loss: 0.8749, val_acc: 0.7268
```

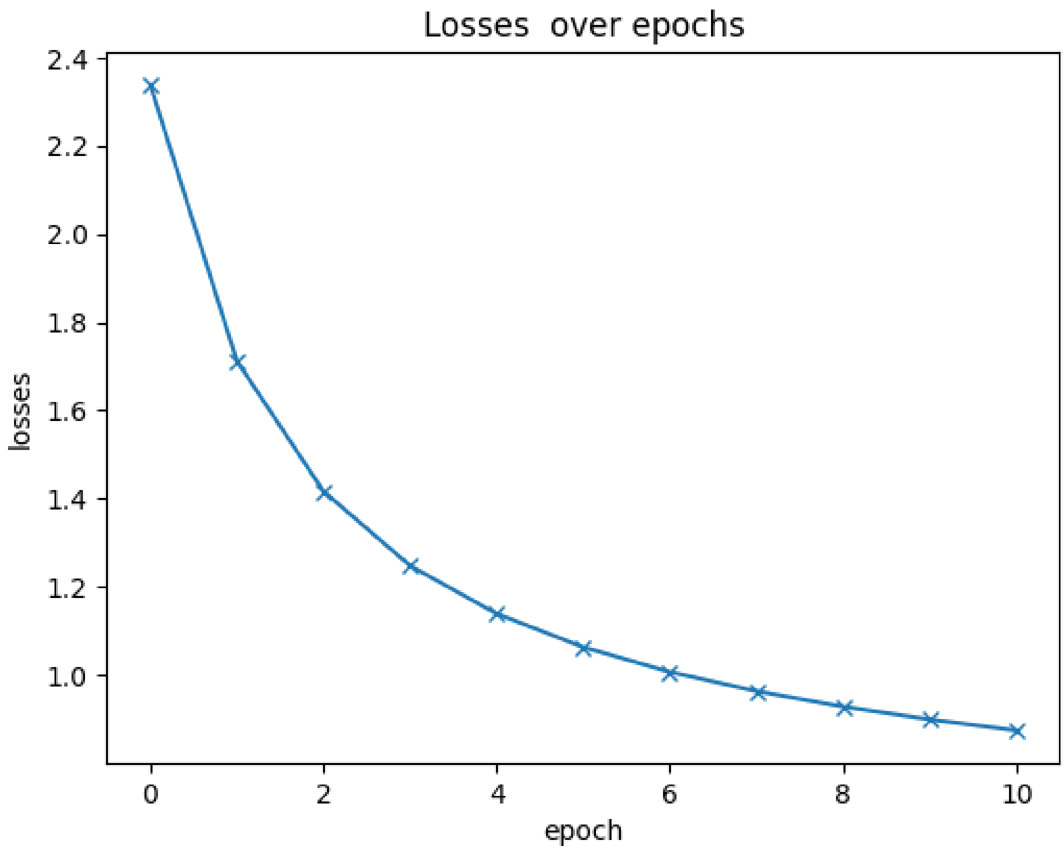
[10]:
```python
# we combine the first result (no training) and the training results of 5␣
↪epoches
# plotting accuracy
history = [result0] + history1
accuracies = [result['val_acc'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy over epochs')
```

[10]: Text(0.5, 1.0, 'Accuracy over epochs')



[11]:
```
# plotting losses
history = [result0] + history1
losses = [result['val_loss'] for result in history]
plt.plot(losses, '-x')
plt.xlabel('epoch')
plt.ylabel('losses')
plt.title('Losses  over epochs')
```

[11]:
```
Text(0.5, 1.0, 'Losses  over epochs')
```

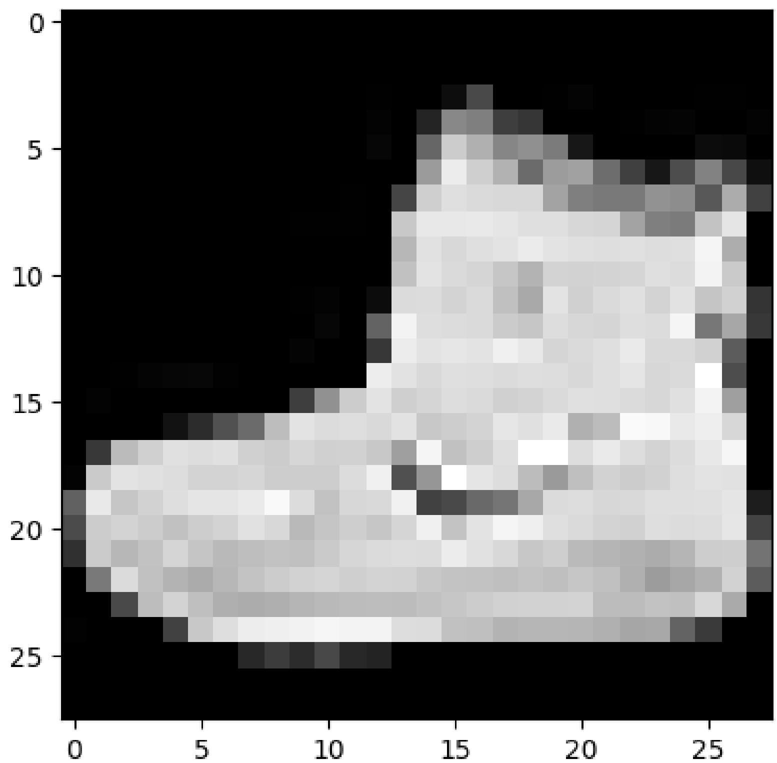


1.1 Final check using the (held-out) test dataset.

We will first load the test dataset (from MNIST) and individually check the prediction made by the model. And then, we will put through all images in the test dataset to obtain the final accuracy

[12]:
```
# Testing with individual images
## Define the test dataset
test_dataset = FashionMNIST(root='data/',train=True, transform=transforms.
  ↪ToTensor())

print("Length of Test Datasets: ", len(test_dataset))
img, label = test_dataset[0]
plt.imshow(img[0], cmap = 'gray')
print("Shape: ", img.shape)
print('Label: ', label)
```

```
Length of Test Datasets:  60000
Shape:  torch.Size([1, 28, 28])
Label:  9
```



[13]:
```
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds = torch.max(yb, dim = 1)
    return(preds[0].item())
```

[14]:
```
img, label = test_dataset[0]
print('Label:', label, ', Predicted :', predict_image(img, model))
```

```
Label: 9 , Predicted : 9
```

[15]:
```
# the final check on the test dataset (not used in any training)
test_loader = DataLoader(test_dataset, batch_size = 256, shuffle = False)
result = evaluate(model, test_loader)
result
```

[15]:
```
{'val_loss': 0.8856238126754761, 'val_acc': 0.7231825590133667}
```

1.2 Results using the same architectures

Accuracy of the randomly initialized model (untrained):0.089

After the training for 10 epochs accuracy raised from 0.6501 to 0.7268

Accuracy of testing dataset was 0.7231

2 Convolutional Neural Network (CNN)

[16]:
```
# We construct a fundamental CNN class.
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization

cnn = CNN()
print(cnn)
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
```

```
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

[17]:
```
loss_func = nn.CrossEntropyLoss()
loss_func

# unlike earlier example using optim.SGD, we use optim.Adam as the optimizer
# lr(Learning Rate): Rate at which our model updates the weights in the cells␣
↪each time back-propagation is done.
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
optimizer
```

[17]:
```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.01
    maximize: False
    weight_decay: 0
)
```

[18]:
```
train_data, validation_data = random_split(mnist_dataset, [50000, 10000])
## Print the length of train and validation datasets
print("length of Train Datasets: ", len(train_data))
print("length of Validation Datasets: ", len(validation_data))

batch_size = 128
train_loader = DataLoader(train_data, batch_size, shuffle = True)
val_loader = DataLoader(validation_data, batch_size, shuffle = False)
```

```
length of Train Datasets:  50000
length of Validation Datasets:  10000
```

[54]:
```
from torch.autograd import Variable

def train(num_epochs, cnn, loaders):
    cnn.train()
    optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
    loss_func = nn.CrossEntropyLoss()
```

```
    # Train the model
    total_step = len(loaders)

    for epoch in range(num_epochs):
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(loaders):

            # gives batch data, normalize x when iterate train_loader
            b_x = Variable(images)   # batch x
            b_y = Variable(labels)   # batch y
            output = cnn(b_x)[0]
            loss = loss_func(output, b_y)

            # clear gradients for this training step
            optimizer.zero_grad()

            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()


            # Calculate training accuracy for this batch
            test_output, _ = cnn(b_x)
            pred_y = torch.max(test_output, 1)[1].data
            accuracy = (pred_y == b_y).sum().item() / float(b_y.size(0))


            if (i + 1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.
↪2f}'
                       .format(epoch + 1, num_epochs, i + 1, total_step, loss.
↪item(), accuracy))
                pass
        pass
    pass
```

[55]:
```
# instiate the CNN model
cnn = CNN()
# for testing purpose, we calculate the accuracy of the initial
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_loader:
        test_output, last_layer = cnn(images)
```

```
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Accuracy of the model on the 10000 test images: %.2f' % accuracy)
```

```
Accuracy of the model on the 10000 test images: 0.11
```

[56]:
```
train(num_epochs=5, cnn=cnn, loaders=train_loader)
```

```
Epoch [1/5], Step [100/391], Loss: 0.5293, Accuracy: 0.82
Epoch [1/5], Step [200/391], Loss: 0.3211, Accuracy: 0.89
Epoch [1/5], Step [300/391], Loss: 0.3209, Accuracy: 0.90
Epoch [2/5], Step [100/391], Loss: 0.3545, Accuracy: 0.87
Epoch [2/5], Step [200/391], Loss: 0.2196, Accuracy: 0.94
Epoch [2/5], Step [300/391], Loss: 0.3121, Accuracy: 0.91
Epoch [3/5], Step [100/391], Loss: 0.5033, Accuracy: 0.87
Epoch [3/5], Step [200/391], Loss: 0.2659, Accuracy: 0.91
Epoch [3/5], Step [300/391], Loss: 0.2674, Accuracy: 0.95
Epoch [4/5], Step [100/391], Loss: 0.3560, Accuracy: 0.91
Epoch [4/5], Step [200/391], Loss: 0.2961, Accuracy: 0.92
Epoch [4/5], Step [300/391], Loss: 0.3585, Accuracy: 0.90
Epoch [5/5], Step [100/391], Loss: 0.2197, Accuracy: 0.94
Epoch [5/5], Step [200/391], Loss: 0.2045, Accuracy: 0.93
Epoch [5/5], Step [300/391], Loss: 0.2546, Accuracy: 0.90
```

3 Evaluate the model on test data

We must call model.eval() to set dropout and batch normalization layers to evaluation mode before running inference. model.train() set layers like dropout, batchnorm etc. to behave for training.

You can call either model.eval() or model.train(mode=False) to tell that you are testing the model.

[57]:
```
# Test the model, after the training
cnn.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_loader:
        test_output, last_layer = cnn(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
```

```
Test Accuracy of the model on the 10000 test images: 0.91
```

Run inference on individual images

[58]:
```python
sample = next(iter(test_loader))
imgs, lbls = sample

actual_number = lbls[:10].numpy()
actual_number

test_output, last_layer = cnn(imgs[:10])
pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
print(f'Prediction number: {pred_y}')
print(f'Actual number: {actual_number}')

hits = 0
for i in range(10):
    if actual_number[i] == pred_y[i]:
        hits += 1

print(f'Correct predictions: {hits} / 10')
```

```
Prediction number: [9 0 0 3 0 2 7 2 5 5]
Actual number: [9 0 0 3 0 2 7 2 5 5]
Correct predictions: 10 / 10
```

3.1 Results using the same architectures

Accuracy of the randomly initialized model (untrained):0.11

After the training for 10 epochs accuracy raised from 0.82 to 0.90

Accuracy of testing dataset was 0.91