Assignment 2

October 28, 2019

[1]:
```python
import numpy as np
import torch
```

Tensors

Question 1

[2]:
```python
x = torch.Tensor(5, 3)
print(x)
print("x was initialized with random numbers to form a 5x3 matrix with data␣
↪type", x.dtype)
print("Type of x is ", x.type())
```

```
tensor([[ 7.0456e-22,  4.5601e-41,  7.0456e-22],
        [ 4.5601e-41, -7.1279e+32,  3.0798e-41],
        [-7.1279e+32,  3.0798e-41,  2.4981e-38],
        [ 4.1144e-37,  2.5039e-32,  1.1450e-16],
        [ 1.0664e-25,  6.6647e-27,  1.0664e-25]])
x was initialized with random numbers to form a 5x3 matrix with data type
torch.float32
Type of x is  torch.FloatTensor
```

Question 2

[3]:
```python
y = torch.rand(5, 3)
print(y)
print("Random values are from a uniform distribution with range (0 - 1)")
print("Type of y is ", y.type())
print("For y = torch.randn(5, 3)")
y = torch.randn(5, 3)
print(y)
print("Random values are from a normal distribution with mean '0' and variance␣
↪'1'")
```

```
tensor([[0.4916, 0.1317, 0.6895],
        [0.1667, 0.2481, 0.9300],
        [0.2182, 0.2320, 0.4158],
        [0.5640, 0.8562, 0.1067],
        [0.4303, 0.8817, 0.1070]])
```

```
Random values are from a uniform distribution with range (0 - 1)
Type of y is  torch.FloatTensor
For y = torch.randn(5, 3)
tensor([[ 0.6867, -2.4114,  0.4417],
        [ 0.9607, -0.7176,  1.1731],
        [-0.4737,  0.7019,  0.7868],
        [ 1.4082, -0.1350, -0.8548],
        [-1.6654, -0.4366, -0.6158]])
Random values are from a normal distribution with mean '0' and variance '1'
```

Question 3

[4]:
```python
x = x.double()
y = y.double()
print(x)
print(y)
print("Type of x is ", x.type())
print("Type of y is ", y.type())
```

```
tensor([[ 7.0456e-22,  4.5601e-41,  7.0456e-22],
        [ 4.5601e-41, -7.1279e+32,  3.0798e-41],
        [-7.1279e+32,  3.0798e-41,  2.4981e-38],
        [ 4.1144e-37,  2.5039e-32,  1.1450e-16],
        [ 1.0664e-25,  6.6647e-27,  1.0664e-25]], dtype=torch.float64)
tensor([[ 0.6867, -2.4114,  0.4417],
        [ 0.9607, -0.7176,  1.1731],
        [-0.4737,  0.7019,  0.7868],
        [ 1.4082, -0.1350, -0.8548],
        [-1.6654, -0.4366, -0.6158]], dtype=torch.float64)
Type of x is  torch.DoubleTensor
Type of y is  torch.DoubleTensor
```

Question 4

[5]:
```python
x = torch.Tensor([[-0.1859,  1.3970,  0.5236],
                  [ 2.3854,  0.0707,  2.1970],
                  [-0.3587,  1.2359,  1.8951],
                  [-0.1189, -0.1376,  0.4647],
                  [-1.8968,  2.0164,  0.1092]])
y = torch.Tensor([[ 0.4838,  0.5822,  0.2755],
                  [ 1.0982,  0.4932, -0.6680],
                  [ 0.7915,  0.6580, -0.5819],
                  [ 0.3825, -1.1822,  1.5217],
                  [ 0.6042, -0.2280,  1.3210]])
print("Shape of x is ", x.shape)
print("Shape of y is ", y.shape)
```

```
Shape of x is  torch.Size([5, 3])
Shape of y is  torch.Size([5, 3])
```

Question 5

[6]:
```
z = torch.stack((x, y))
print("Shape of z is ", z.size())
print(torch.cat((x, y), 0).shape)
print(torch.cat((x, y), 1).shape)
```

```
Shape of z is  torch.Size([2, 5, 3])
torch.Size([10, 3])
torch.Size([5, 6])
```

torch.stack() gives a 3d tensor as output, whereas *torch.cat*() gives a concatenated 2d tensor along the row or column as output.

Question 6

[7]:
```
print("Accesing element in 2d tensor")
print(y[4][2])
print("Accesing the same element in 3d tensor")
print(z[1][4][2])
```

```
Accesing element in 2d tensor
tensor(1.3210)
Accesing the same element in 3d tensor
tensor(1.3210)
```

Question 7

[8]:
```
print("Elements corresponding to 5th row and 3rd column in z are ", z[:, 4, 2])
```

```
Elements corresponding to 5th row and 3rd column in z are  tensor([0.1092,
1.3210])
```

As z is a 3d tensor, there are two elements present. First element corresponds to the x tensor and the second to the y tensor.

Question 8

[9]:
```
print(x + y)
print(torch.add(x, y))
print(x.add(y))
torch.add(x, y, out=x)
print(x)
```

```
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
```

```
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
tensor([[ 0.2979,  1.9792,  0.7991],
        [ 3.4836,  0.5639,  1.5290],
        [ 0.4328,  1.8939,  1.3132],
        [ 0.2636, -1.3198,  1.9864],
        [-1.2926,  1.7884,  1.4302]])
```

All the above instructions are printing the same output. Yes, they are equivalent.

Question 9

[10]:
```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

$torch.randn(4, 4)$ generates a 4x4 matrix with normally distributed random elements.

$x.view(16)$ reshapes the 4x4 matrix to 1x16 matrix.

$x.view(-1, 8)$ reshapes the 4x4 matrix to 2x8 matrix. -1 mean that there exist a integer 'N' such that N*8 = 16, here N is 2. So, the shape of z is (2,8)

Question 10

[11]:
```
x = torch.randn(10, 10)
x = x.view(1, 100)
y = torch.randn(2, 100)
y = y.view(100, 2)
z = torch.mm(x, y)
print("Shape of the resulting matrix", z.shape)
```

```
Shape of the resulting matrix torch.Size([1, 2])
```

NumPy and PyTorch

Question 11

[12]:
```
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
print("Type of a is ", a.type())
```

```
print("Type of b is ", type(b))
```

```
tensor([1., 1., 1., 1., 1.])
[1. 1. 1. 1. 1.]
Type of a is  torch.FloatTensor
Type of b is  <class 'numpy.ndarray'>
```

a is a float tensor and b is a numpy array, but both have the same elements.

Question 12

[13]:
```
a[0] += 1
print(a)
print(b)
```

```
tensor([2., 1., 1., 1., 1.])
[2. 1. 1. 1. 1.]
```

a and b match, and they share their memory locations.

Question 13

[14]:
```
a.add_(1)
print(a)
print(b)
```

```
tensor([3., 2., 2., 2., 2.])
[3. 2. 2. 2. 2.]
```

$a.add\_(1)$ increments all the elements in tensor a by 1, this increment also reflects in b as they share their memory locations.

[15]:
```
a[:] += 1
print(a)
print(b)
```

```
tensor([4., 3., 3., 3., 3.])
[4. 3. 3. 3. 3.]
```

$a[:]+=1$ is similar to $a.add\_(1)$. It also increments all the elements in tensor a by 1, this increment also reflects in b.

[16]:
```
a = a.add(1)
print(a)
print(b)
```

```
tensor([5., 4., 4., 4., 4.])
[4. 3. 3. 3. 3.]
```

$a = a.add(1)$ increments all the elements in tensor a by 1, but this increment doesn't reflect in b and they don't share their memory locations.

Question 14

[17]:
```python
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

Converting *b* from NumPy array to a Torch tensor, increment of *a* reflects in *b* as they share their memory locations.

Question 15

[18]:
```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
x = torch.randn(5, 3).to(device)
y = torch.randn(5, 3, device=device)
z = x + y
print(z)
```

```
cuda
tensor([[ 0.9728, -0.2318,  1.0569],
        [ 0.3523, -1.4869,  0.4920],
        [-0.2055,  1.8624, -0.7422],
        [-2.2417,  0.3104,  2.1540],
        [ 1.1184,  0.6013, -0.1686]], device='cuda:0')
```

The result is 'cuda' because *torch.cuda* is available.

The first allocation instruction generates the torch tensor in CPU and then converts the tensor to device GPU, and the second allocation instruction generates the torch tensor directly in device GPU.

So, the second one is most efficient.

Question 16

[19]:
```python
print(z.cpu().numpy())
print(z.numpy())
```

```
[[ 0.9728469  -0.2317853   1.0568774 ]
 [ 0.3522702  -1.4868824   0.4920123 ]
 [-0.20550883  1.862351   -0.74222106]
 [-2.2416973   0.31038302  2.1540072 ]
 [ 1.1184045   0.60128546 -0.16859782]]
```

```
        TypeErrorTraceback (most recent call last)
```

```
    <ipython-input-19-11d5c486e6eb> in <module>
      1 print(z.cpu().numpy())
----> 2 print(z.numpy())


    TypeError: can't convert CUDA tensor to numpy. Use Tensor.cpu() to copy␣
↪the tensor to host memory first.
```

$z.cpu().numpy()$ results in moving data from GPU to CPU memory, and z is converted from cuda tensor to numpy array.

$z.numpy()$ results in an error because z is a cuda tensor, we cannot directly convert to numpy. First we need to change it to CPU memory, then we can change to numpy.

Autograd: automatic differentiation

Question 17

[20]:
```python
x = torch.ones(2, 2, requires_grad=True)
print(x)
y = x + 2
print(y)
print("requires_grad attribute of y is ", y.requires_grad)
print("grad attributes of x is ", x.grad)
print("grad attributes of y is ", y.grad)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
requires_grad attribute of y is  True
grad attributes of x is  None
grad attributes of y is  None
```

Question 18

[21]:
```python
z = y * y * 3
f = z.mean()
print(z, f)
```

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27.,
grad_fn=<MeanBackward0>)
```

z is equated as $(y * y) * 3$. First, y*y does element-wise multiplication. Second, the result is multiplied by 3. f takes a mean of z.

Write f in terms of entries of matrix x such that $f = f(x_1, x_2, x_3, x_4)$

$$x = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$

$y = x + 2 = \begin{bmatrix} x_1 + 2 & x_2 + 2 \\ x_3 + 2 & x_4 + 2 \end{bmatrix}$
$z = y * y * 3 = \begin{bmatrix} 3(x_1 + 2)^2 & 3(x_2 + 2)^2 \\ 3(x_3 + 2)^2 & 3(x_4 + 2)^2 \end{bmatrix}$
$f = z.mean() = \frac{1}{4}(3(x_1 + 2)^2 + 3(x_2 + 2)^2 + 3(x_3 + 2)^2 + 3(x_4 + 2)^2)$
$= \frac{3}{4}((x_1)^2 + (x_2)^2 + (x_3)^2 + (x_4)^2) + 3(x_1 + x_2 + x_3 + x_4) + 12$
$= \frac{3}{4}(\sum_{i=1}^{4}(x_i)^2) + 3(\sum_{i=1}^{4} x_i) + 12$

Question 19

[22]:
```
f.backward()
print("grad attributes of x is")
print(x.grad)
```

```
grad attributes of x is
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Question 20

$(\nabla_x f(x))_i = \frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_i}$
$\frac{\partial f(x_1,x_2,x_3,x_4)}{\partial x_i} = 3 + 1.5x_i$

As all the elements of x is equal to 1. We get the attributes as 4.5.

MNIST Data preparation

Question 21

[23]:
```
from matplotlib import pyplot
import MNISTtools

# Loading the training and testing datasets

xtrain, ltrain = MNISTtools.load(dataset="training", path=None)
xtest, ltest = MNISTtools.load(dataset="testing", path=None)

xtrain = xtrain.astype(np.float32)  # Converting array type from int to float
xtest = xtest.astype(np.float32)

# Normalizing the datasets

def normalize_MNIST_images(x):
    x = -1 + (2*x/255)
    return x

# Checking the normalize_MNIST_images() function

xtrain = normalize_MNIST_images(xtrain)
xtest = normalize_MNIST_images(xtest)
```

```
print("Range of normalized xtrain is [", np.min(xtrain), ", ", np.max(xtrain),␣
 ↪"]")
print("Range of normalized xtest is [", np.min(xtest), ", ", np.max(xtest), "]")
```

```
Range of normalized xtrain is [ -1.0 ,  1.0 ]
Range of normalized xtest is [ -1.0 ,  1.0 ]
```

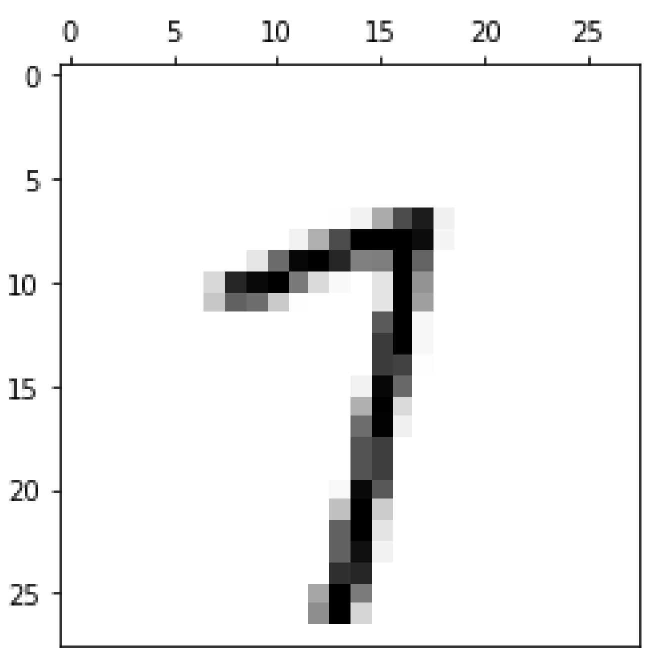
Question 22

[24]:
```
# Reorganising the tensors

xtrain = xtrain.reshape(28, 28, 1, 60000)
xtest = xtest.reshape(28, 28, 1, 10000)
xtrain = np.moveaxis(xtrain, [0, 1, 2, 3], [2, 3, 1, 0])
xtest = np.moveaxis(xtest, [0, 1, 2, 3], [2, 3, 1, 0])
print("New shape of xtrain is ", xtrain.shape)
print("New shape of xtest is ", xtest.shape)
```

```
New shape of xtrain is  (60000, 1, 28, 28)
New shape of xtest is  (10000, 1, 28, 28)
```

Question 23

[25]:
```
# Displaying one training sample

MNISTtools.show(xtrain[42, 0, :, :])
print("Digit displayed is ", ltrain[42])
```



```
Digit displayed is  7
```

Question 24

[26]:
```
# Convert all numpy array to torch tensor

xtrain = torch.from_numpy(xtrain)
ltrain = torch.from_numpy(ltrain)
xtest = torch.from_numpy(xtest)
ltest = torch.from_numpy(ltest)
```

CNN for MNIST classification

Question 25

Size of the feature maps after each convolution and maxpooling operation,

point (i) 24x24x6

point (ii) 12x12x6

point (iii) 8x8x16

point (iv) 4x4x16

Fully connected layer has,

point (v) 256 inputs

Question 26

[27]:
```
# Initialize LeNet network

import torch.nn as nn
import torch.nn.functional as F

# This is our neural networks class that inherits from nn.Module
class LeNet(nn.Module):
    # Here we define our network structure
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1   = nn.Linear(4*4*16, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    # Here we define one forward pass through the network
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
```

```python
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    # Determine the number of features in a batch of tensors
    def num_flat_features(self, x ):
        size = x.size()[1:]
        return np.prod(size)

net = LeNet()
print(net)
```

```
LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=256, 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)
)
```

Question 27

[28]:
```python
# Parameters of the initialized network

for name, param in net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

```
conv1.weight torch.Size([6, 1, 5, 5]) True
conv1.bias torch.Size([6]) True
conv2.weight torch.Size([16, 6, 5, 5]) True
conv2.bias torch.Size([16]) True
fc1.weight torch.Size([120, 256]) True
fc1.bias torch.Size([120]) True
fc2.weight torch.Size([84, 120]) True
fc2.bias torch.Size([84]) True
fc3.weight torch.Size([10, 84]) True
fc3.bias torch.Size([10]) True
```

Learnable parameters are weight and bias.

We can see that the *requires_grad* for all the parameters are *True*, so the gradients will be tracked by *autograd*.

Question 28

[29]:
```python
# Forward pass

with torch.no_grad():
```

```
    yinit = net(xtest)

_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(9.7400)
```

For the initial prediction, we take the weights and bias to be random. As there are 10 classes of data, there will be a 1/10 probability that our prediction is correct. So we get the performance around 10% .

Question 29

[30]:
```
N = xtrain.size()[0]  # Training set size
B = 100  # Minibatch size
NB = int((N+B-1)/B)  # Number of minibatches
gamma = .001  # Step size
rho = .9  # Momentum
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=gamma, momentum=rho)
```

Question 30

[31]:
```
# SGD and Backprop

def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    N = xtrain.size()[0]  # Training set size
    NB = int((N+B-1)/B)  # Number of minibatches
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(net.parameters(), lr=gamma, momentum=rho)
    for epoch in  range(T):
        running_loss = 0.0
        shuffled_indices = np.random.permutation(range(N))
        for k in  range(NB):
            # Extract k-th minibatch from xtrain and ltrain
            minibatch_indices = shuffled_indices[B*k:min(B*(k+1), N)]
            inputs = xtrain[minibatch_indices]
            labels = ltrain[minibatch_indices]

            # Initialize the gradients to zero
            optimizer.zero_grad()

            # Forward propagation
            outputs = net(inputs)

            # Error evaluation
            loss = criterion(outputs, labels)

            # Back propagation
```

```
            loss.backward()

            # Parameter update
            optimizer.step()

            # Print averaged loss per minibatch every 100 mini-batches
            # Compute and print statistics
            with torch.no_grad():
                running_loss += loss.item()
            if k % 100 == 99:
                print('[%d,%5d] loss:%.3f' %
                      (epoch + 1, k + 1, running_loss / 100))
                running_loss = 0.0
```

[32]:
```
# Training the network

net = LeNet()
backprop_deep(xtrain, ltrain, net, T=3)
```

```
[1,   100] loss:2.303
[1,   200] loss:2.299
[1,   300] loss:2.296
[1,   400] loss:2.290
[1,   500] loss:2.283
[1,   600] loss:2.273
[2,   100] loss:2.252
[2,   200] loss:2.205
[2,   300] loss:2.054
[2,   400] loss:1.454
[2,   500] loss:0.748
[2,   600] loss:0.481
[3,   100] loss:0.407
[3,   200] loss:0.325
[3,   300] loss:0.298
[3,   400] loss:0.264
[3,   500] loss:0.265
[3,   600] loss:0.245
```

Question 31

[33]:
```
# Testing performance of trained network

with torch.no_grad():
    ytrained = net(xtest)

_, lpred = ytrained.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(93.6200)
```

Accuracy of the trained network with 3 epochs is 93.62% whereas the initialization accuracy was 9.74% which is a huge improvement.

Question 32

[34]:
```
# Training the network with GPU

net_gpu = LeNet().to(device)
xtrain = xtrain.to(device)
ltrain = ltrain.to(device)
backprop_deep(xtrain, ltrain, net_gpu, T=10)
```

```
[1,   100] loss:2.302
[1,   200] loss:2.293
[1,   300] loss:2.283
[1,   400] loss:2.263
[1,   500] loss:2.218
[1,   600] loss:2.071
[2,   100] loss:1.640
[2,   200] loss:1.066
[2,   300] loss:0.753
[2,   400] loss:0.566
[2,   500] loss:0.445
[2,   600] loss:0.370
[3,   100] loss:0.317
[3,   200] loss:0.289
[3,   300] loss:0.262
[3,   400] loss:0.238
[3,   500] loss:0.219
[3,   600] loss:0.210
[4,   100] loss:0.202
[4,   200] loss:0.183
[4,   300] loss:0.173
[4,   400] loss:0.166
[4,   500] loss:0.159
[4,   600] loss:0.160
[5,   100] loss:0.157
[5,   200] loss:0.139
[5,   300] loss:0.134
[5,   400] loss:0.132
[5,   500] loss:0.138
[5,   600] loss:0.130
[6,   100] loss:0.127
[6,   200] loss:0.118
[6,   300] loss:0.118
[6,   400] loss:0.119
[6,   500] loss:0.119
```

```
[6,  600] loss:0.113
[7,  100] loss:0.102
[7,  200] loss:0.103
[7,  300] loss:0.108
[7,  400] loss:0.114
[7,  500] loss:0.095
[7,  600] loss:0.105
[8,  100] loss:0.097
[8,  200] loss:0.101
[8,  300] loss:0.092
[8,  400] loss:0.089
[8,  500] loss:0.092
[8,  600] loss:0.097
[9,  100] loss:0.084
[9,  200] loss:0.087
[9,  300] loss:0.089
[9,  400] loss:0.083
[9,  500] loss:0.088
[9,  600] loss:0.089
[10,  100] loss:0.083
[10,  200] loss:0.080
[10,  300] loss:0.081
[10,  400] loss:0.080
[10,  500] loss:0.087
[10,  600] loss:0.075
```

Question 33

[35]:
```
# Testing performance of trained network with GPU

xtest = xtest.to(device)
ltest = ltest.to(device)
with torch.no_grad():
    y_gpu = net_gpu(xtest)

_, lpred = y_gpu.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(97.9600, device='cuda:0')
```

Accuracy of the trained network with 10 epochs is 97.96% whereas the accuracy of the trained network with 3 epochs was 93.62%. There is an increase in the accuracy by 3.34%.