Assignment2

October 28, 2019

1 A53299801 - SIDDARTH MEENAKSHI SUNDARAM

[14]:
```
import numpy as np
import torch
```

Q1

[15]:
```
#1
x = torch.Tensor(5, 3)
print(x)
print(x.dtype)
```

```
tensor([[1.7220e-19, 4.5862e-41, 9.7723e-33],
        [3.0649e-41, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 0.0000e+00, 0.0000e+00],
        [0.0000e+00, 1.0573e-05, 5.3928e-05],
        [1.0267e-08, 3.2722e+21, 5.4074e+22]])
torch.float32
```

x is a randomly initialized torch tensor and has tensor float32 type data in it.

Q2

[7]:
```
#2
y = torch.rand(5, 3)
print(y)
print(y.type)
print(y.dtype)
m = torch.randn(5, 3)
print(m)
```

```
tensor([[0.7457, 0.7923, 0.7371],
        [0.3874, 0.9519, 0.4159],
        [0.0239, 0.6625, 0.5345],
        [0.3470, 0.9751, 0.8164],
        [0.7884, 0.9875, 0.2636]])
<built-in method type of Tensor object at 0x7fd7bc631af8>
torch.float32
tensor([[ 0.0875, -4.2743,  0.9688],
        [ 0.3362, -0.8952, -0.8127],
        [ 0.9349,  1.3945, -0.3724],
        [ 0.1297, -0.3361, -2.2466],
        [-0.4067,  0.8415,  0.3909]])
```

The type of y is a torch tensor and has float32 elements in it.

Rand - Uniformlly distributed random values

i.e. the y value is uniform distribution initialized randomly with range [0, 1)

Randn - Normally distributed random values

i.e. the m value is standard normal distribution initialized randomly.

Q3

```
[4]: #3
x = x.double()
y = y.double()
print(x)
print(y)
```

```
tensor([[-1.9798e-07,  3.0729e-41,  5.7453e-44],
        [ 0.0000e+00,         nan,  1.8040e+28],
        [ 1.3733e-14,  6.4076e+07,  2.0706e-19],
        [ 7.3909e+22,  2.4176e-12,  1.1625e+33],
        [ 8.9605e-01,  1.1632e+33,  5.6003e-02]], dtype=torch.float64)
tensor([[0.3267, 0.0934, 0.0838],
        [0.4676, 0.2407, 0.4326],
        [0.4270, 0.8477, 0.3468],
        [0.2688, 0.9133, 0.5159],
        [0.6870, 0.5458, 0.6191]], dtype=torch.float64)
```

Datatype has now changed to torch.float64 from torch.float32   Q4

```
[78]: #4
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("Answer:\n")
print("X shape is:",x.shape," -> a torch tensor with dimensions 5x3","\nY shape␣
 ↪is:",y.shape," -> a torch tensor with dimensions 5x3")
```

```
print("x datatype:",x.dtype,"\ny datatype:",y.dtype)
```

```
Answer:

X shape is: torch.Size([5, 3])  -> a torch tensor with dimensions 5x3
Y shape is: torch.Size([5, 3])  -> a torch tensor with dimensions 5x3
x datatype: torch.float32
y datatype: torch.float32
```

Q5

[85]:
```
#5
z = torch.stack((x, y))
#print(z)
print("Shape of Z is:",z.shape)
```

```
Shape of Z is: torch.Size([2, 5, 3])
```

[86]:
```
z1=torch.cat((x, y), 0)
z2=torch.cat((x, y), 1)
print(z1.shape)
print(z2.shape)
```

```
torch.Size([10, 3])
torch.Size([5, 6])
```

In case of stacking two 2D tensors, it becomes a one 3D tensor, but in case of cat(concatenation), the tensors are concatenated along the row in case of z1 and along the column in case of z2, i.e. the tensors remain as 2D tensors.

Q6

[93]:
```
#6
print("Value of element at the 5th row and 3rd column in the 2d tensor y is:
 ↪",y[4,2])
print("Taking the same element from z tensor:",z[1,4,2])
```

```
Value of element at the 5th row and 3rd column in the 2d tensor y is:
tensor(1.3210)
Taking the same element from z tensor: tensor(1.3210)
```

We can see that the element value is the same.

Q7

[94]:
```
#7
print(z[:,4,2])
#print(z[0,4,2],z[1,4,2]) #Confirming by printing values separately.
```

```
tensor([0.1092, 1.3210])
```

Printing all the elements corresponding to the 5th row and 3rd column in z. In this case, it prints z[0,4,2] and z[1,4,2] as above. There are 2 elements in this case.

Q8

```
[10]: #8
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]])
```

They are printing the same output.

```
[11]: #8
print((x+y) - torch.add(x, y))
print(torch.add(x, y)-x.add(y))
m=x.add(y)
torch.add(x, y, out=x)
print(x-m)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

They are all equivalent as we can see that all difference between the different summation types is zero.

Q9

[96]:
```
#9
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])
```

The first instruction generates a 4x4 tensor.

The second one converts in to a 1x16 tensor.

The -1 in the row space denotes that the row may vary but the column has 8 elements. The given torch tensor is converted into some Nx8 tensor. N depends on the the number of elements in the original tensor i.e. N = (Number of elements in orginal tensor / 8) = 16 / 8 = 2.

Q10

[99]:
```
#10
x = torch.rand(10,10)

y = torch.rand(2,100)

#print(x,y)
```

[100]:
```
x=x.view(-1,100)
print(x.shape)
#print(x.shape)
y=y.view(100,-1)
print(y.shape)
p=torch.mm(x,y)
print(p.shape)
```

```
torch.Size([1, 100])
torch.Size([100, 2])
torch.Size([1, 2])
```

Converted x into [1, 100] tensor and y into a [100, 2] tensor. The resulting Matrix multiplication is a [1, 2] row vector.

Q11

```
[129]: #11
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
```

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

```
[131]: print(a.shape)
print(b.shape)
print(a.type,type(b))
print(a.dtype,b.dtype)
```

```
torch.Size([5])
(5,)
<built-in method type of Tensor object at 0x7f9ba5e909d8> <class
'numpy.ndarray'>
torch.float32 float32
```

We can see that a is Torch tensor and b is a numpy n-dimensional array and they hold the same values, datatype and also possess the same dimension.

Q12

```
[132]: #12
a[0] += 1
print(a)
print(b)
```

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

The values of a and b match.

Also, We can see that any changes in the a tensor affects the numpy array b, Hence we can say that they share the underlying memory location.

We can see that the converse is also true. That is change in numpy array b also changes the tensor a.

Q13

```
[134]: #13
print(b)
a.add_(1)
print("M1", a, b)
```

```
a[:] += 1
print("M2", a, b)
a = a.add(1)
print("M3", a, b)
```

```
[2. 1. 1. 1. 1.]
M1 tensor([3., 2., 2., 2., 2.]) [3. 2. 2. 2. 2.]
M2 tensor([4., 3., 3., 3., 3.]) [4. 3. 3. 3. 3.]
M3 tensor([5., 4., 4., 4., 4.]) [4. 3. 3. 3. 3.]
```

a.add_(1) adds 1 to every element.

a[:] += 1 also adds 1 to every element

a = a.add(1) adds 1 only to a and the underlying memory location is changed, anymore changes to a will affect only a.

Q14

[136]:
```
#14
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a,a.dtype)
print(b)
```

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

The numpy array has been converted to torch tensor. Value of 1 is added to the array a. We can see that the value of the torch tensor has also changed. This is further proof that they share underlying memory locations.

Q15

[143]:
```
#15
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
```

```
cuda
```

The first instruction initializes the matrix in cpu and then transfers it to gpu whereas the second one directly initializes in the gpu. Hence the second instruction may be more efficient.

Q16

[140]:
```
#16
print(z.cpu().numpy())
print(z.numpy())
```

```
[[ 1.7573642  -1.0952468   1.184261  ]
 [ 3.342084   -0.47547942 -1.3795434 ]
 [ 2.1563416   0.07843184  0.748652  ]
 [ 3.1624985   2.2253366   2.6280928 ]
 [-0.46363577 -2.117697    2.1221347 ]]
```

```
        TypeErrorTraceback (most recent call last)

        <ipython-input-140-99235fdef16a> in <module>()
          1 #16
          2 print(z.cpu().numpy())
    ----> 3 print(z.numpy())


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

The first instruction converts the CUDA tensor to cpu variable and then converts into a numpy array. Z can't be directly converted from a CUDA tensor into numpy without converting into a CPU device variable first.

Q17

[145]:
```
#17
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
print(y)
print("\nrequires_grad attribute of y:",y.requires_grad)
print("x grad attribute:",x.grad,"\ny grad attribute:",y.grad)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)

requires_grad attribute of y: True
x grad attribute: None
y grad attribute: None
```

Q18

[146]:
```
#18
z = y * y * 3
f = z.mean()
print(z,"\n", f)
```

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

```
tensor(27., grad_fn=<MeanBackward1>)
```

Result is the mean of the elements of the element-wise multiplication of y matrix with itself , multiplied by 3. This is can seen from the grad_fn= MulBackward0 and grad_fn= MeanBackward1

[148]:
```
x1=(x[0,0]+2)
x2=(x[0,1]+2)
x3=(x[1,0]+2)
x4=(x[1,1]+2)

f1 = 3*(x1 * x1+ x4 * x4 + 2 * (x2 * x3) + x1 * x3 + x2 * x4 + x3 * x4 + x1 *␣
 ↪x2) / 8
print("Answer 18:")
print(f1)
```

```
Answer 18:
tensor(27., grad_fn=<DivBackward0>)
```

$$z = y * y * 3 = \begin{bmatrix} x_1+2 & x_2+2 \\ x_3+2 & x_4+2 \end{bmatrix} * \begin{bmatrix} x_1+2 & x_2+2 \\ x_3+2 & x_4+2 \end{bmatrix} * 3$$
$$z = \begin{bmatrix} (x_1+2)^2 & (x_2+2)^2 \\ (x_3+2)^2 & (x_4+2)^2 \end{bmatrix} * 3$$
$$f(x_1,x_2,x_3,x_4) = f = z.mean()$$
$$Hence, f = (x_1+2)^2 + (x_2+2)^2 + (x_3+2)^2 + (x_4+2)^2 * \frac{3}{4}$$
$$f = \frac{3}{4}(x_1^2 + 2x_1 + 4 + x_2^2 + 2x_2 + 4 + x_3^2 + 2x_3 + 4 + x_4^2 + 2x_4 + 4)$$
$$f(x_1,x_2,x_3,x_4) = \frac{3}{4}(x_1^2 + x_2^2 + x_3^2 + x_4^2 + 4(x_1+x_2+x_3+x_4) + 16)$$

Substituting we can check that,

$$f(1,1,1,1) = \frac{3}{4} * (1+1+1+1+4(1+1+1+1)+16)$$
$$= 0.75 * 36$$
$$= 27$$

The computer has done element wise multiplication and arrived at the answer of 27. We can see that the caluculation by hand produces the same result.

Hence Verified

Q19

[149]:
```
#19
f.backward() # That's it!
print(x.grad)
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

All gradients are equal to 4.5

Q20

$$(\nabla_x f(x))_i = (\frac{\partial f(x_i)}{\partial x_i})^T = \frac{3}{4}(2x_i + 4)$$
$$x_i = 1 \implies (\nabla_x f(x))_i = \frac{3}{4}(2(1) + 4)$$

$\implies (\nabla_x f(x))_i = 4.5$

All gradients are equal to 4.5 as all x values are equal to 1.

Q21

```
[10]: #21
import MNISTtools
xtrain, ltrain = MNISTtools.load(dataset="training", path=None) #Loading␣
 ↪training sets into xtrain and ltrain
xtest, ltest = MNISTtools.load(dataset="testing", path=None) #Loading testing␣
 ↪sets into xtest and ltest
xtrain = xtrain.astype(np.float32)
xtest = xtest.astype(np.float32)
def normalize_MNIST_images(x):
    x = -1 + (2*x/255)
    return x
xtrain = normalize_MNIST_images(xtrain) #Normalizing xtrain
xtest = normalize_MNIST_images(xtest)   #Normalizing xtest
```

```
[11]: print(np.max(xtest))
```

```
1.0
```

Q22

```
[12]: #22
xtrain=xtrain.reshape(28, 28, 1, 60000)
xtest=xtest.reshape(28,28,1,10000)
print(xtrain.shape)
#print(xtest.shape)
```

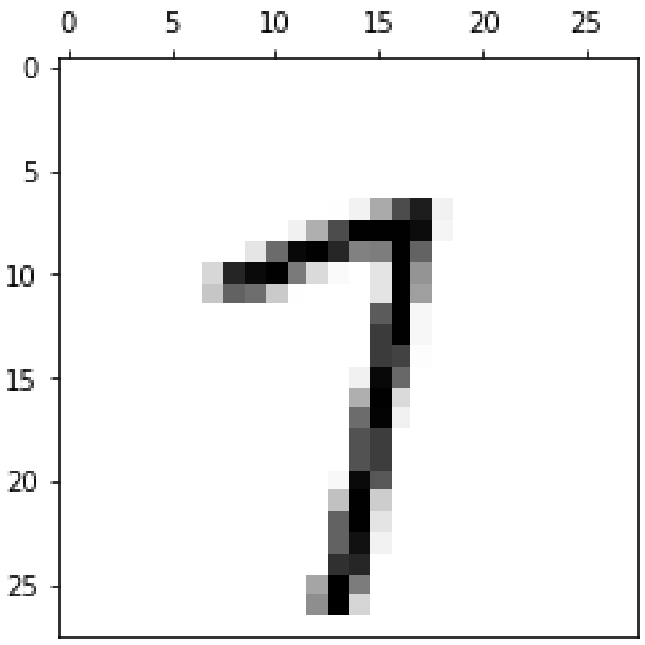
```
(28, 28, 1, 60000)
```

```
[13]: #Changing input format from Height x Width x Input channel x Batch Size to␣
 ↪Batch Size x Input channel x Height x Width
xtrain = np.moveaxis(xtrain, [0, 1, 2, 3], [2, 3, 1, 0])
print(xtrain.shape)
xtest = np.moveaxis(xtest, [0, 1, 2, 3], [2, 3, 1, 0])
print(xtrain.shape)
```

```
(60000, 1, 28, 28)
(60000, 1, 28, 28)
```

Q23

```
[58]: #23 Verifying data reorganisation
import MNISTtools
MNISTtools.show(xtrain[42, 0, :, :])
print("Label is:", ltrain[42])
```



```
Label is: 7
```

We can see that the label matches with the image at xtrain [42, 0, :, :]

Q24

[59]:
```python
#24 Wrapping data into torch Tensor
xtrain = torch.from_numpy(xtrain)
ltrain = torch.from_numpy(ltrain)
xtest = torch.from_numpy(xtest)
ltest = torch.from_numpy(ltest)
```

Q25, Q26

[60]:
```python
#25
#26

#Code is to develop a combination of CNN's and

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(16*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))
        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)
)
```

Size of feature map at

```
i. 24x24x6
ii. 12x12x6
iii. 8x8x16
iv. 4x4x16
```

Input units of third layer

```
v. 256
```

Q27

[61]:
```
#27
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
```

Biases and weights are learnable parameters. All parameters will be tracked by autograd as the param.requires_grad is true for all cases.

Q28

[62]:
```
#28
with torch.no_grad():
    yinit = net(xtest)
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(10.1600)
```

The training accuracy is 10.16% at initialization. That is, without any training, the network gives the correct output only about 10% of the time (which can be interpreted with probability) .

Q29, Q30

[64]:
```
#29 #30
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)␣
 ↪#learning  rate(lr)=gamma
    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

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

```
[1,   100] loss: 2.303
[1,   200] loss: 2.298
[1,   300] loss: 2.291
[1,   400] loss: 2.284
[1,   500] loss: 2.273
[1,   600] loss: 2.252
[2,   100] loss: 2.208
[2,   200] loss: 2.067
[2,   300] loss: 1.547
[2,   400] loss: 0.766
[2,   500] loss: 0.458
[2,   600] loss: 0.367
[3,   100] loss: 0.311
[3,   200] loss: 0.261
[3,   300] loss: 0.241
[3,   400] loss: 0.242
[3,   500] loss: 0.221
[3,   600] loss: 0.189
```

Q31

[65]:
```
#31
#Testing Performance of network on testing dataset
with torch.no_grad():
    yinit = net(xtest)
_, lpred = yinit.max(1)
```

```
print(100 * (ltest == lpred).float().mean())
```

```
tensor(94.7500)
```

The accuracy has improved from 10.16% during initialization to 94.75% on the trained network(after 3 epochs).

Q32

[67]:
```
#32 Moving training data into GPU and training again

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

```
[1,   100] loss: 2.304
[1,   200] loss: 2.295
[1,   300] loss: 2.285
[1,   400] loss: 2.264
[1,   500] loss: 2.224
[1,   600] loss: 2.111
[2,   100] loss: 1.708
[2,   200] loss: 1.036
[2,   300] loss: 0.639
[2,   400] loss: 0.496
[2,   500] loss: 0.420
[2,   600] loss: 0.375
[3,   100] loss: 0.325
[3,   200] loss: 0.291
[3,   300] loss: 0.288
[3,   400] loss: 0.252
[3,   500] loss: 0.253
[3,   600] loss: 0.223
[4,   100] loss: 0.216
[4,   200] loss: 0.192
[4,   300] loss: 0.196
[4,   400] loss: 0.188
[4,   500] loss: 0.165
[4,   600] loss: 0.173
[5,   100] loss: 0.158
[5,   200] loss: 0.150
[5,   300] loss: 0.152
[5,   400] loss: 0.154
[5,   500] loss: 0.139
[5,   600] loss: 0.136
[6,   100] loss: 0.136
[6,   200] loss: 0.129
[6,   300] loss: 0.119
```

```
[6,   400] loss: 0.127
[6,   500] loss: 0.115
[6,   600] loss: 0.119
[7,   100] loss: 0.118
[7,   200] loss: 0.110
[7,   300] loss: 0.107
[7,   400] loss: 0.102
[7,   500] loss: 0.108
[7,   600] loss: 0.101
[8,   100] loss: 0.098
[8,   200] loss: 0.094
[8,   300] loss: 0.098
[8,   400] loss: 0.094
[8,   500] loss: 0.099
[8,   600] loss: 0.094
[9,   100] loss: 0.097
[9,   200] loss: 0.083
[9,   300] loss: 0.086
[9,   400] loss: 0.083
[9,   500] loss: 0.087
[9,   600] loss: 0.085
[10,   100] loss: 0.083
[10,   200] loss: 0.080
[10,   300] loss: 0.084
[10,   400] loss: 0.077
[10,   500] loss: 0.077
[10,   600] loss: 0.082
```

Q33

[68]:
```
#33

xtest=xtest.to(device)
ltest=ltest.to(device)
with torch.no_grad():
    yinit = net(xtest)
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(97.7000, device='cuda:0')
```

The accuracy has improved from 94.75% to 97.7% using 10 epochs and learning using a gpu.