assignment2

October 28, 2019

1 Venkatesh Prasad Venkataramanan PID : A53318036

[2]:
```
import numpy as np
import torch
import MNISTtools
from matplotlib import pyplot as plt
```

1.1 Question 1

[3]:
```
x = torch.Tensor(5 , 3)
print(x)
print(type(x))
```

```
tensor([[5.7037e+34, 4.5675e-41, 5.6992e+34],
        [4.5675e-41, 5.7038e+34, 4.5675e-41],
        [5.6993e+34, 4.5675e-41, 5.7022e+34],
        [4.5675e-41, 6.6279e+34, 4.5675e-41],
        [5.7059e+34, 4.5675e-41, 5.7059e+34]])
<class 'torch.Tensor'>
```

X was initialised as a tensor object from PyTorch.

1.2 Question 2

[4]:
```
y = torch.rand(5 , 3)
print(y)
print(type(y))
y2 = torch.randn(5 , 3)
print(y2)
print(type(y2))
```

```
tensor([[0.1377, 0.8655, 0.2795],
        [0.9360, 0.5037, 0.7908],
        [0.2819, 0.2305, 0.8202],
        [0.4840, 0.0554, 0.6504],
```

```
        [0.0875, 0.5708, 0.1319]])
<class 'torch.Tensor'>
tensor([[ 0.0705, -0.9844, -0.9151],
        [-0.6822,  0.0144, -0.3439],
        [-0.7303, -1.0881, -0.6431],
        [ 0.1751, -1.0674, -1.5497],
        [-0.4476, -0.8006,  0.5768]])
<class 'torch.Tensor'>
```

As we can see, y is a Torch Tensor.
The rand function produces a uniform distribution between 0 and 1.
The randn function produces a normalised distribution with mean 0, variance 1.

1.3 Question 3

[6]:
```
x = x.double()
y = y.double()
print(x)
print(y)

print(x.type())
print(y.type())
```

```
tensor([[5.7037e+34, 4.5675e-41, 5.6992e+34],
        [4.5675e-41, 5.7038e+34, 4.5675e-41],
        [5.6993e+34, 4.5675e-41, 5.7022e+34],
        [4.5675e-41, 6.6279e+34, 4.5675e-41],
        [5.7059e+34, 4.5675e-41, 5.7059e+34]], dtype=torch.float64)
tensor([[0.1377, 0.8655, 0.2795],
        [0.9360, 0.5037, 0.7908],
        [0.2819, 0.2305, 0.8202],
        [0.4840, 0.0554, 0.6504],
        [0.0875, 0.5708, 0.1319]], dtype=torch.float64)
torch.DoubleTensor
torch.DoubleTensor
```

x and y are both Torch Double Tensors.

1.4 Question 4

[6]:
```
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 (x.shape)
print (y.shape)
```

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

The shape of x and y is [5,3]

1.5 Question 5

[7]:
```
z = torch.stack((x,y))
z2 = torch.cat((x,y),0)
z3 = torch.cat((x,y),1)
print(z.shape)
print(z2.shape)
print(z3.shape)
```

```
torch.Size([2, 5, 3])
torch.Size([10, 3])
torch.Size([5, 6])
```

The torch stack function stacks into a three dimensional tensor.
The cat function concatenates along a particular axis.

1.6 Question 6

[8]:
```
print(y[4,2])
print(z[1,4,2])
```

```
tensor(1.3210)
tensor(1.3210)
```

1.7 Question 7

[9]:
```
print(z[:,4,2])
print(len(z[:,4,2]))
```

```
tensor([0.1092, 1.3210])
2
```

There are 2 elements

1.8 Question 8

[10]:
```
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.

1.9 Question 9

[11]:
```
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])
```

x.view(16) flattens a 4 * 4 vector into a 1 * 16 vector.
x.view(-1,8) keeps one axis to be 8, and calculates the other axis as per x's dimensions.

1.10 Question 10

[12]:
```
x=torch.randn(10,10)
y=torch.randn(2,100)
x2=x.view(1,100)
y2=y.view(100,2)
res=torch.mm(x2,y2)
print(res)
print(res.shape)
```

```
tensor([[-10.3214,  11.2763]])
torch.Size([1, 2])
```

1.11 Question 11

[13]:
```
a=torch.ones(5)
print(a)
b=a.numpy()
print(b)

print(type(a))
print(type(b))
```

```
tensor([1., 1., 1., 1., 1.])
[1. 1. 1. 1. 1.]
<class 'torch.Tensor'>
<class 'numpy.ndarray'>
```

a is a torch tensor.
b is a numpy array.

1.12 Question 12

[14]:
```
a[0]+=1
print(a)
print(b)
```

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

They match. They share underlying memory locations, as we can see that, b is changing, as a is changing.

1.13 Question 13

[15]:
```
a.add_(1)
print(a,b)
a[:]+=1
print(a,b)
a=a.add(1)
print(a,b)
```

```
tensor([3., 2., 2., 2., 2.]) [3. 2. 2. 2. 2.]
tensor([4., 3., 3., 3., 3.]) [4. 3. 3. 3. 3.]
tensor([5., 4., 4., 4., 4.]) [4. 3. 3. 3. 3.]
```

From observation, one can say that, a.add_(1) and a[:]+=1 is also modifying b.
But, a.add(1) does not.

1.14 Question 14

[16]:
```
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)
```

As mentioned in the question, torch tensors support conversion to numpy and back.

1.15 Question 15

[17]:
```
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
```

x has been created on the cpu first, and then transferred to GPU.
y has been created directly on the GPU, so it is more efficient in comparison, since transfers between CPU and GPU are expensive.

1.16 Question 16

[18]:
```
print(z.cpu().numpy())
print(z.numpy())
```

```
[[-2.1995711  -2.9684575   0.63747984]
 [-0.9683557  -1.1878556  -0.1688531 ]
 [-0.02476832  0.1635682   0.2107917 ]
 [ 1.0837587   0.42707998  0.11120033]
 [-1.9955192  -0.04232109  1.970746  ]]
```

```
        TypeErrorTraceback (most recent call last)

        <ipython-input-18-11d5c486e6eb> 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.
```

We are converting z to cpu location first, then converting to numpy.
Since z is now existing on CUDA, we can't convert to numpy. We need to convert to CPU first.

1.17 Question 17

[22]:
```
x = torch.ones(2, 2, requires_grad=True)
print(x)
y = x + 2
print(y)
print(y.requires_grad)
print(x.grad)
print(y.grad)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
True
None
None
```

The requires_grad attribute of y is True.
The grad attribute of x is None .

The grad attribute of y is None

1.18 Question 18

[19]:
```
z=y*y*3
f=z.mean()
print(z,f)
```

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

The relation has been attached.

1.19 Question 19

[20]:
```
f.backward()
```

[21]:
```
x.grad
```

[21]:
```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

1.20 Question 20

The calculation has been attached.

1.21 Question 21

[22]:
```
import MNISTtools
import numpy as np
from matplotlib import pyplot
```

[23]:
```
xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")
print(xtrain.shape)
print(ltrain.shape)
print(xtest.shape)
print(ltest.shape)
```

```
(784, 60000)
(60000,)
(784, 10000)
(10000,)
```

[24]:
```
xtrain, ltrain = MNISTtools.load(dataset="training", path="/datasets/
 ↪MNIST")#loaded data
xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")

xtrain = xtrain.astype(np.float32)#float conversion
xtest = xtest.astype(np.float32)

def normalize_MNIST_images(x):
    x = -1.0 + (2*x)/255#mapping [0,255] to [-1,1]
    return x


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

1.22 Question 22

[25]:
```
xtrain = np.reshape(xtrain,(28,28,1,60000))
print (xtrain.shape)


xtest = np.reshape(xtest,(28,28,1,10000))
print (xtest.shape)
```

```
(28, 28, 1, 60000)
(28, 28, 1, 10000)
```

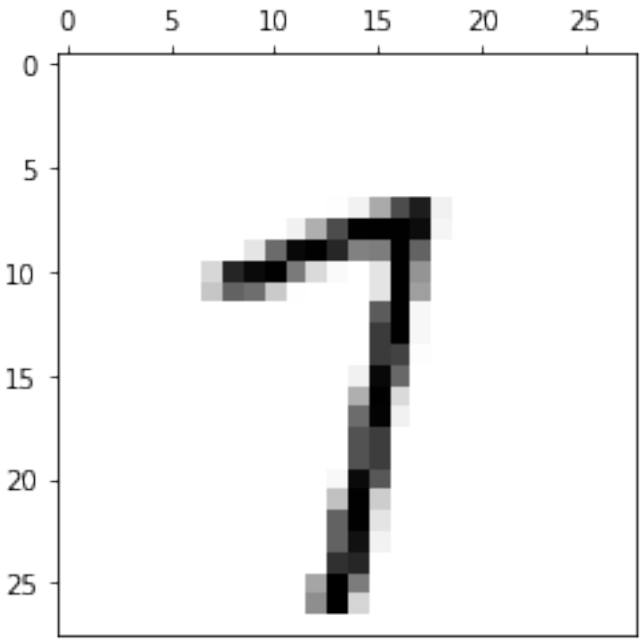
[26]:
```
xtrain = np.transpose(xtrain.reshape(28,28,1,60000),[3,2,0,1])
xtest = np.transpose(xtest.reshape(28,28,1,10000),[3,2,0,1])

print (xtrain.shape)
print (xtest.shape)
```

```
(60000, 1, 28, 28)
(10000, 1, 28, 28)
```

1.23 Question 23

[27]:
```
import matplotlib as pyplot
MNISTtools.show(xtrain[42,0,:,:])
print(ltrain[42])
```



7

1.24 Question 24

[28]:
```
xtrain = torch.from_numpy(xtrain)
ltrain = torch.from_numpy(ltrain)
xtest = torch.from_numpy(xtest)
ltest = torch.from_numpy(ltest)
```

1.25 Question 25

The input image is 28 * 28 pixels.
(i) After the first convolutional layer, we have the size as 6 * 24 * 24.
(ii) After the maxpooling layer, we have size as 6 * 12 * 12.
(iii) After the second convolutional layer, we have the size as 16 * 8 * 8.
(iv) After the maxpooling layer, we have the size as 16 * 4 * 4.
So input at (v) has 256 units.

Then we use a linear network to convert to 120 units. Then to 84 units. Then to 10 output units.

1.26 Question 26

[29]:
```
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*4*4,120)
        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)
)
```

1.27 Question 27

[30]:
```
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
```

The learnable parameters are thus:

Weight of the 1st Convolutional Neural Network. Bias of the 1st Convolutional Neural Network.

Weight of the 2nd Convolutional Neural Network. Bias of the 2nd Convolutional Neural Network.

Weight of the 1st Fully Connected Layer. Bias of the 1st Fully Connected Layer.

Weight of the 2nd Fully Connected Layer. Bias of the 2nd Fully Connected Layer.

Weight of the 3rd Fully Connected Layer. Bias of the 3rd Fully Connected Layer.

They are going to be tracked for all parameters.

1.28 Question 28

[31]:
```
with torch.no_grad():
    yinit = net(xtest)
```

[32]:
```
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(9.7200)
```

On a random net, we are getting 1/10th of our predictions correctly, which is logical, since a random guess would have 10% probability.

1.29 Question 29

[33]:
```
def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    N = xtrain.size()[0]     # Training set size
    NB = 600             # Number of minibatches
    criterion = nn.CrossEntropyLoss()
```

```
    optimizer = torch.optim.SGD(net.parameters(), lr=gamma, momentum=rho)
```

1.30 Question 30

[35]:
```
def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    N = xtrain.size()[0]     # Training set size
    NB = 600            # 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(NB)
        for k in range(NB):
            # Extract k-th minibatch from xtrain and ltrain
            i = shuffled_indices[k]
            minibatch_indices = range(i*B,i*B+B)
            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: 2.302
[1,   200] loss: 2.297
[1,   300] loss: 2.290
[1,   400] loss: 2.280
[1,   500] loss: 2.259
[1,   600] loss: 2.216
[2,   100] loss: 2.085
[2,   200] loss: 1.628
[2,   300] loss: 0.951
[2,   400] loss: 0.592
[2,   500] loss: 0.476
[2,   600] loss: 0.394
[3,   100] loss: 0.330
[3,   200] loss: 0.299
[3,   300] loss: 0.275
[3,   400] loss: 0.247
[3,   500] loss: 0.241
[3,   600] loss: 0.243
```

Hence we get a loss of 0.243, which is very close to the expected value of 0.22.

1.31 Question 31

[36]:
```
with torch.no_grad():
    yinit = net(xtest)
```

[37]:
```
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(94.0700)
```

We get an accuracy of 94.07%, which is a huge improvement to the 9.72% we got off the initialised network.

1.32 Question 32

[39]:
```
xtrain = xtrain.to(device)
ltrain = ltrain.to(device)
net = LeNet().to(device)
backprop_deep(xtrain, ltrain, net, T=10)
```

```
[1,   100] loss: 2.304
[1,   200] loss: 2.299
[1,   300] loss: 2.295
[1,   400] loss: 2.289
[1,   500] loss: 2.280
[1,   600] loss: 2.263
```

```
[2,   100] loss: 2.227
[2,   200] loss: 2.130
[2,   300] loss: 1.790
[2,   400] loss: 1.150
[2,   500] loss: 0.810
[2,   600] loss: 0.583
[3,   100] loss: 0.477
[3,   200] loss: 0.393
[3,   300] loss: 0.352
[3,   400] loss: 0.298
[3,   500] loss: 0.277
[3,   600] loss: 0.265
[4,   100] loss: 0.228
[4,   200] loss: 0.218
[4,   300] loss: 0.217
[4,   400] loss: 0.188
[4,   500] loss: 0.187
[4,   600] loss: 0.175
[5,   100] loss: 0.164
[5,   200] loss: 0.158
[5,   300] loss: 0.162
[5,   400] loss: 0.148
[5,   500] loss: 0.153
[5,   600] loss: 0.141
[6,   100] loss: 0.129
[6,   200] loss: 0.134
[6,   300] loss: 0.127
[6,   400] loss: 0.131
[6,   500] loss: 0.120
[6,   600] loss: 0.119
[7,   100] loss: 0.116
[7,   200] loss: 0.123
[7,   300] loss: 0.108
[7,   400] loss: 0.104
[7,   500] loss: 0.115
[7,   600] loss: 0.096
[8,   100] loss: 0.116
[8,   200] loss: 0.092
[8,   300] loss: 0.096
[8,   400] loss: 0.100
[8,   500] loss: 0.095
[8,   600] loss: 0.093
[9,   100] loss: 0.089
[9,   200] loss: 0.087
[9,   300] loss: 0.091
[9,   400] loss: 0.088
[9,   500] loss: 0.084
[9,   600] loss: 0.093
```

```
[10,   100] loss: 0.076
[10,   200] loss: 0.081
[10,   300] loss: 0.087
[10,   400] loss: 0.086
[10,   500] loss: 0.073
[10,   600] loss: 0.085
```

1.33 Question 33

[40]:
```
xtest = xtest.to(device)
with torch.no_grad():
    yinit = net(xtest)
```

[41]:
```
ltest = ltest.to(device)
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

```
tensor(97.9200, device='cuda:0')
```

The accuracy has improved from 94.07% to 97.92% , after running for 10 iterations.
Hence we achieve an accuracy of nearly 98%, which is an improvement to the 96% we got off the artificial neural network-based classifier in assignment 1.