assignment2

October 30, 2019

1 Assignment 2 - CNNs and PyTorch

1.0.1 Name: Anirudh Swaminathan

1.0.2 PID: A53316083

1.0.3 Email ID: aswamina@eng.ucsd.edu

Notebook created by Anirudh Swaminathan from ECE department majoring in Intelligent Systems, Robotics and Control for the course ECE285 Machine Learning for Image Processing for Fall 2019

1.1 Getting Started

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

1.2 Tensors

Question 1

[2]:
```
# Construct 5*3 tensor and print it
x = torch.Tensor(5, 3)
print(x)

# printing its type
print(type(x))

# printing its data type
print(x.dtype)
```

```
tensor([[1.3563e-19, 1.8888e+31, 4.7414e+16],
        [4.0047e-11, 6.4097e-10, 5.8253e-10],
        [6.4097e-10, 1.3567e-19, 4.1486e-08],
        [1.4585e-19, 6.3369e-10, 7.9348e+17],
        [1.3556e-19, 1.3563e-19, 1.3563e-19]])
```

```
<class 'torch.Tensor'>
torch.float32
```

x was randomly initialized. x is of type **torch.Tensor** and it's data is of type torch.float32

Question 2

[3]:
```
y = torch.rand(5, 3)
print(y)

# Finding the type of y
print(type(y))
print(y.dtype)

# Using randn() instead of rand()
y1 = torch.randn(5, 3)
print(y1)
print(type(y1))
print(y1.dtype)
```

```
tensor([[0.3412, 0.5528, 0.5889],
        [0.6602, 0.6299, 0.1609],
        [0.5155, 0.8596, 0.4628],
        [0.1864, 0.1299, 0.4162],
        [0.1052, 0.3734, 0.9375]])
<class 'torch.Tensor'>
torch.float32
tensor([[ 1.4837, -0.8288, -1.2014],
        [-0.0122,  1.9979,  1.8102],
        [ 0.3882, -1.2326, -1.2263],
        [ 1.8649, -2.0401, -0.6632],
        [ 0.8967,  0.1870,  0.8523]])
<class 'torch.Tensor'>
torch.float32
```

y is a $(5, 3)$ tensor with random values distributed in a uniform distribution from 0 to 1 y is of type **torch.Tensor** and it's data is of type torch.float32 y_1 is a $(5, 3)$ tensor with random values distributed as a Gaussian with mean 0 and variance 1. So, y_1 is a tensor filled with random values from a standard normal distribution So, if we use $torch.randn()$ function instead of $torch.rand()$, we may get negative values for $torch.randn()$ but not for $torch.rand()$ function.

Question 3

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

```
tensor([[1.3563e-19, 1.8888e+31, 4.7414e+16],
```

```
        [4.0047e-11, 6.4097e-10, 5.8253e-10],
        [6.4097e-10, 1.3567e-19, 4.1486e-08],
        [1.4585e-19, 6.3369e-10, 7.9348e+17],
        [1.3556e-19, 1.3563e-19, 1.3563e-19]], dtype=torch.float64)
tensor([[0.3412, 0.5528, 0.5889],
        [0.6602, 0.6299, 0.1609],
        [0.5155, 0.8596, 0.4628],
        [0.1864, 0.1299, 0.4162],
        [0.1052, 0.3734, 0.9375]], dtype=torch.float64)
```

The type displayed when we print x and y are $torch.float64$

Question 4

[5]:
```python
# Initialize tensors with values directly
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]])
```

[6]:
```python
# Display the shapes of the two tensors x and y
print(x.shape, y.shape)
```

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

Shape of x is $(5, 3)$. Shape of y is $(5, 3)$.

Question 5

[7]:
```python
# Stack 2 tensors
z = torch.stack((x, y))
```

[8]:
```python
print(z, z.dtype, z.shape)
```

```
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]],

        [[ 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]]]) torch.float32 torch.Size([2, 5, 3])
```

[9]:
```python
# Now, compare it with torch.cat()
z1 = torch.cat((x, y), 0)
z2 = torch.cat((x, y), 1)
print(z1, z1.shape)
print(z2, z2.shape)
```

```
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],
        [ 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]]) torch.Size([10, 3])
tensor([[-0.1859,  1.3970,  0.5236,  0.4838,  0.5822,  0.2755],
        [ 2.3854,  0.0707,  2.1970,  1.0982,  0.4932, -0.6680],
        [-0.3587,  1.2359,  1.8951,  0.7915,  0.6580, -0.5819],
        [-0.1189, -0.1376,  0.4647,  0.3825, -1.1822,  1.5217],
        [-1.8968,  2.0164,  0.1092,  0.6042, -0.2280,  1.3210]]) torch.Size([5,
6])
```

The shape of the tensor z is $(2, 5, 3)$. $torch.stack()$ stacks its arguments on a new dimension, i.e., on top of one another in this case.

We also compared it to $torch.cat()$. In $torch.cat((x, y), 0)$, the tensor y is concatenated to tensor x along axis 0, i.e., it is concatenated along the row. This results in a tensor that is of the shape $(10, 3)$ that is obtained by combining the two tensors, each of shape $(5, 3)$ row-wise. The output of this $torch.cat()$ is still the same dimension $(2D)$.

Similarly, in $torch.cat((x, y), 1)$, the tensor y is concatenated to tensor x along axis 1, i.e., it is concatenated along the column. This results in a tensor that is of the shape $(5, 6)$ that is obtained by combining the two tensors, each of shape $(5, 3)$ column-wise. The output of this $torch.cat()$ is still the same dimension $(2D)$.

Question 6

[10]:
```python
# Accessing the element of the 5th row and 3rd column in 2d tensor y
ele = y[4, 2]
print("The element of the 5th row and 3rd column in 2d tensor y:", ele.item())

# Accessing the same element in the 3D tensor z
ele_3d = z[1, 4, 2]
print("Accessing the same element in the 3d tensor z, we have", ele_3d.item())
```

```
The element of the 5th row and 3rd column in 2d tensor y: 1.3209999799728394
Accessing the same element in the 3d tensor z, we have 1.3209999799728394
```

Hence, we were able to access the element of the 5^{th} row and 3^{rd} column in the $2D$ tensor y. We were also able to access the same element from the $3D$ tensor z.

Question 7

[11]:
```
# Print all elements corresponding to the 5th row and 3rd column in z
eles = z[:, 4, 2]
print("Printing all elements corresponding to the 5th row and 3rd column in z:
↪", eles)
print(eles.shape)
```

```
Printing all elements corresponding to the 5th row and 3rd column in z:
tensor([0.1092, 1.3210])
torch.Size([2])
```

There are 2 elements in z that correspond to the 5^{th} row and 3^{rd} column of the tensor z. This is beacause z is the stacked tensor of x and y. Hence, the 1^{st} returned element corresponds to the element at the 5^{th} row and 3^{rd} column of the tensor x, and the 2^{nd} returned element corresponds to the element at the 5^{th} row and 3^{rd} column of the tensor y.

Question 8

[12]:
```
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 4 methods of addition print the same output. Also, all the 4 methods are equivalent. They all take in 2 tensors x and y, and then output a new tensor. They all do NOT modify the tensors x and y. Tensor x seems modified in the last statement only because $out = x$ was specified, which meant that *torch* stored the output of the addition operation between x and y in the variable x.

Question 9

[13]:
```python
# create a tensor whose values are sampled from a Normal Distribution with mean␣
↪0 and variance 1
x = torch.randn(4, 4)

# store a reshaped version of x in y such that it is a 1D tensor of size 16
y = x.view(16)

# store the reshaped version of x in z such that it is a 2D tensor of size (2,␣
↪8)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
print(x)
print(y)
print(z)
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
tensor([[-0.2910, -0.5872,  0.2615, -0.4767],
        [ 1.7549, -1.2167, -0.7197,  0.8474],
        [-0.4785, -1.1303, -1.3507,  0.2692],
        [-0.0975, -0.0522, -0.7660,  1.2422]])
tensor([-0.2910, -0.5872,  0.2615, -0.4767,  1.7549, -1.2167, -0.7197,  0.8474,
        -0.4785, -1.1303, -1.3507,  0.2692, -0.0975, -0.0522, -0.7660,  1.2422])
tensor([[-0.2910, -0.5872,  0.2615, -0.4767,  1.7549, -1.2167, -0.7197,
0.8474],
        [-0.4785, -1.1303, -1.3507,  0.2692, -0.0975, -0.0522, -0.7660,
1.2422]])
```

The 1^{st} statement creates tensor x of shape $(4, 4)$ using the $randn()$ function whose values are sampled from a Normal Distribution with $\mu = 0$ and $\sigma^2 = 1$. The 2^{nd} statement stores a reshaped version of x in y such that it is a $1D$ tensor of size 16. The 3^{rd} statement stores a reshaped version of x in z such that it is a $2D$ tensor of size $(2, 8)$. The -1 in the argument for the $view()$ function states that the 1^{st} dimension of z should be inferred by *torch* directly given that the 2^{nd} dimension of z is 8. This conversion is just $4 * 4 = 16$; $\frac{16}{8} = 2$. Thus, the 1^{st} dimension of z should be 2.

Question 10

[14]:
```python
# Generate random x of dimension 10*10
x = torch.randn(10, 10)
```

```
# Generate random y of dimension 2*100
y = torch.randn(2, 100)
print(x.size(), y.size())

# reshape x to become a row vector and make it compatible for matrix␣
↪multiplication
x = x.view(1, 100)

# reshape y to become a matrix compatible for matrix multiplication with x
y = y.view(100, 2)
print(x.size(), y.size())

# perform row vector by matrix multiplication
z = torch.mm(x, y)
print(z.size())
print(z)
```

```
torch.Size([10, 10]) torch.Size([2, 100])
torch.Size([1, 100]) torch.Size([100, 2])
torch.Size([1, 2])
tensor([[ -7.5519, -10.7117]])
```

We created a tensor x of size $(10, 10)$ and tensor y of size $(2, 100)$. We then reshaped the tensor x to row vector of size $(1, 100)$. Tensor y was also reshaped to size $(100, 2)$ to make it conformable for matrix multiplication with x. Finally, the result of the matrix multiplication carried out by $torch.mm(x, y)$ is stored in the tensor z. Tensor z is of size $(1, 100) * (100, 2) = (1, 2)$.

1.3 Numpy and PyTorch

Question 11

[15]:
```
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)

print(type(a), type(b))
print(a.dtype, b.dtype)
print(a.size(), b.shape)
```

```
tensor([1., 1., 1., 1., 1.])
[1. 1. 1. 1. 1.]
<class 'torch.Tensor'> <class 'numpy.ndarray'>
torch.float32 float32
torch.Size([5]) (5,)
```

Variable a is a $1D$ tensor of size (5) carrying data of type $torch.float32$. Variable b is a $1D$ numpy array of shape $(5,)$ carrying data of type $float32$. b is the *numpy* version of tensor a and both carry the same data.

Question 12

[16]:
```
a[0] += 1
print(a)
print(b)
```

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

Tensor a and numpy array b both share the same underlying memory location. Modifying a changes b and modifying b changes a if the tensor a is on the CPU, which is the case here.

Question 13

[17]:
```
a.add_(1)
print(a)
print(b)
```

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

The *add_*(1) modifies a in-place, thus modifying numpy array b also.

[18]:
```
a[:] += 1
print(a)
print(b)
```

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

This statement modifies a, thus modifying numpy b also.

[19]:
```
a = a.add(1)
print(a)
print(b)
```

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

The satement $a.add(1)$ adds 1 to a and returns a new tensor. Since we store the result in a, only a is now the variable that points to the new tensor output, but the underlying memory location is not modified. Thus, b is not modified.

Question 14

[20]:
```
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)
```

Numpy array a and tensor b share the same underlying location. Modifying a changes b and vice-versa if the tensor is in the CPU.

Question 15

[21]:
```
# GPU experiments
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)

# Create a tensor on CPU and then move it to GPU
x = torch.randn(5, 3).to(device)

# Create a tensor directly on the GPU
y = torch.randn(5, 3, device=device)
z = x + y
print(x.size(), x.dtype, x.device)
print(y.size(), y.dtype, y.device)
print(z.size(), z.dtype, z.device)
print(x)
print(y)
print(z)
```

```
cuda
torch.Size([5, 3]) torch.float32 cuda:0
torch.Size([5, 3]) torch.float32 cuda:0
torch.Size([5, 3]) torch.float32 cuda:0
tensor([[-0.8340, -0.2621, -1.5960],
        [ 0.4451,  2.2970, -0.5133],
        [ 2.9873,  0.7170, -0.3918],
        [ 0.1028,  0.1764,  0.6040],
        [ 0.6886,  0.1032, -0.7320]], device='cuda:0')
tensor([[-0.4412,  0.2622, -1.3004],
        [-0.6348,  0.8524, -1.7512],
        [ 0.6262, -0.7539,  0.8823],
        [ 0.2665,  0.1195,  1.0902],
        [-0.2456,  1.0017,  0.8983]], device='cuda:0')
tensor([[-1.2752e+00,  1.3217e-04, -2.8964e+00],
        [-1.8975e-01,  3.1494e+00, -2.2645e+00],
        [ 3.6135e+00, -3.6901e-02,  4.9054e-01],
        [ 3.6936e-01,  2.9586e-01,  1.6942e+00],
        [ 4.4299e-01,  1.1049e+00,  1.6633e-01]], device='cuda:0')
```

Tensor x is first created in the CPU and then transferred to the GPU using the $.to()$ command. Tensor y is created in the GPU directly with the *device* argument in the $randn()$ function. I feel that the allocation instruction for y is more efficient than the one for x as creating a tensor on CPU and then transferring it to GPU is 2 steps, with the additional overhead of transferring it between devices. Directly allocating the tensor to the GPU would avoid these extra steps, and hence would be more efficient comparatively

Question 16

[22]:
```
# This line runs fine
print(z.cpu().numpy())

# The following line produces an error
print(z.numpy())
```

```
[[-1.2751775e+00  1.3217330e-04 -2.8964458e+00]
 [-1.8974915e-01  3.1494401e+00 -2.2645404e+00]
 [ 3.6135235e+00 -3.6900699e-02  4.9054387e-01]
 [ 3.6936343e-01  2.9585814e-01  1.6942282e+00]
 [ 4.4298744e-01  1.1048564e+00  1.6632962e-01]]
```

```
        TypeErrorTraceback (most recent call last)

        <ipython-input-22-abcfc2eaf4a0> in <module>
          3
          4 # The following line produces an error
    ----> 5 print(z.numpy())


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

In the 1^{st} line, the tensor z is copied to CPU first using the $.cpu()$ function. Then, it is converted to a numpy array in the CPU. The 2^{nd} line throws a *TypeError* as *torch* can't convert the CUDA tensor to numpy directly. It means that the conversion has to be carried out in the CPU, and hence $z.cpu()$ has to be used first before the conversion.

1.4 Autograd: automatic differentiation

Question 17

[23]:
```
x = torch.ones(2, 2, requires_grad=True)
print(x)
y = x + 2
print(y)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

[24]:
```python
print(y.requires_grad)
print(x.grad)
print(y.grad)
print(x.grad_fn)
print(y.grad_fn)
```

```
True
None
None
None
<AddBackward0 object at 0x7f95530e7ef0>
```

Since the $requires\_grad$ attribute of x is $True$ and we are performing operations on x to obtain y, y will have its attribute $requires\_grad$ set to $True$ automatically. The $grad$ attributes of both the tensors x and y are $None$. This is because the gradient has not been computed yet for these tensors using the $.backward()$ function.

Question 18

[25]:
```python
z = y * y * 3
f = z.mean()
print(z, f)
```

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

We find that z has elements that are the square of each elemnent of y times 3. The tensor f contains that value that is the mean of the elements of z.

We shall now discuss the relation between f and the 4 entries of x - namely $x_1, x_2, x_3, and x_4$

$$f = f(x_1, x_2, x_3, x_4)$$

We know that

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

Given,

$$y = x + 2$$

So,

$$y = \begin{bmatrix} x_1 + 2 & x_2 + 2 \\ x_3 + 2 & x_4 + 2 \end{bmatrix}$$

Given,

$$z = y * y * 3$$

So,

$$z = \begin{bmatrix} x_1+2 & x_2+2 \\ x_3+2 & x_4+2 \end{bmatrix} \otimes \begin{bmatrix} x_1+2 & x_2+2 \\ x_3+2 & x_4+2 \end{bmatrix} * 3$$

$$\implies z = \begin{bmatrix} 3(x_1+2)^2 & 3(x_2+2)^2 \\ 3(x_3+2)^2 & 3(x_4+2)^2 \end{bmatrix}$$

Given,

$$f = z.mean()$$

So,

$$f = \frac{3(x_1+2)^2 + 3(x_2+2)^2 + 3(x_3+2)^2 + 3(x_4+2)^2}{4}$$

$$\implies f = \frac{3}{4} * \left[x_1^2 + 4x_1 + 4 + x_2^2 + 4x_2 + 4 + x_3^2 + 4x_3 + 4 + x_4^2 + 4x_4 + 4\right]$$

Hence, finally we have that

$$f(x_1, x_2, x_3, x_4) = \frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12$$

Question 19

[26]:
```
f.backward()
print(x.grad)
```

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

[27]:
```
print(f.grad, z.grad, y.grad, x.grad)
print(f.grad_fn, z.grad_fn, y.grad_fn, x.grad_fn)
print(f.requires_grad, z.requires_grad, y.requires_grad, x.requires_grad)
```

```
None None None tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
<MeanBackward0 object at 0x7f95530e7080> <MulBackward0 object at 0x7f95530f6a90>
<AddBackward0 object at 0x7f95530f67f0> None
True True True True
```

By running the code above, we compute the gradient of the tensor f with respect to x using autograd. We find that

$$\nabla_x f(x) = \left(\frac{\delta f(x)}{\delta x}\right)^T = \begin{bmatrix} 4.5000 & 4.5000 \\ 4.5000 & 4.5000 \end{bmatrix}$$

Question 20 Given,

$$(\nabla_x f(x))_i = \frac{\delta f(x_1, x_2, x_3, x_4)}{\delta x_i}$$

From question 18, we know that

$$f(x_1, x_2, x_3, x_4) = \frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12$$

Taking $i = 1$,

$$(\nabla_x f(x))_1 = \frac{\delta f(x_1, x_2, x_3, x_4)}{\delta x_1}$$

$$(\nabla_x f(x))_1 = \frac{\delta\left(\frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12\right)}{\delta x_1}$$

$$(\nabla_x f(x))_1 = \frac{3}{2}x_1 + 3$$

Taking $i = 2$,

$$(\nabla_x f(x))_2 = \frac{\delta f(x_1, x_2, x_3, x_4)}{\delta x_2}$$

$$(\nabla_x f(x))_2 = \frac{\delta\left(\frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12\right)}{\delta x_2}$$

$$(\nabla_x f(x))_2 = \frac{3}{2}x_2 + 3$$

Taking $i = 3$,

$$(\nabla_x f(x))_3 = \frac{\delta f(x_1, x_2, x_3, x_4)}{\delta x_3}$$

$$(\nabla_x f(x))_3 = \frac{\delta\left(\frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12\right)}{\delta x_3}$$

$$(\nabla_x f(x))_3 = \frac{3}{2}x_3 + 3$$

Taking $i = 4$,

$$(\nabla_x f(x))_4 = \frac{\delta f(x_1, x_2, x_3, x_4)}{\delta x_4}$$

$$(\nabla_x f(x))_4 = \frac{\delta\left(\frac{3}{4}x_1^2 + \frac{3}{4}x_2^2 + \frac{3}{4}x_3^2 + \frac{3}{4}x_4^2 + 3x_1 + 3x_2 + 3x_3 + 3x_4 + 12\right)}{\delta x_4}$$

$$(\nabla_x f(x))_4 = \frac{3}{2}x_4 + 3$$

Thus, we have

$$\nabla_x f(x) = \begin{bmatrix} (\nabla_x f(x))_1 & (\nabla_x f(x))_2 \\ (\nabla_x f(x))_3 & (\nabla_x f(x))_4 \end{bmatrix}$$

$$\implies \nabla_x f(x) = \begin{bmatrix} \frac{3}{2}x_1 + 3 & \frac{3}{2}x_2 + 3 \\ \frac{3}{2}x_3 + 3 & \frac{3}{2}x_4 + 3 \end{bmatrix}$$

Here, in this example, we have taken the tensor x to be 2×2 matrix filled with just 1. Hence, we have

$$x = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Thus, we have $x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1$. This means that we have

$$\nabla_x f(x) = \begin{bmatrix} \frac{3}{2}x_1 + 3 & \frac{3}{2}x_2 + 3 \\ \frac{3}{2}x_3 + 3 & \frac{3}{2}x_4 + 3 \end{bmatrix} = \begin{bmatrix} \frac{3}{2}1 + 3 & \frac{3}{2}1 + 3 \\ \frac{3}{2}1 + 3 & \frac{3}{2}1 + 3 \end{bmatrix}$$

$$\implies \nabla_x f(x) = \begin{bmatrix} 4.5 & 4.5 \\ 4.5 & 4.5 \end{bmatrix}$$

This is consistent with the output of the command $x.grad$ in question 19, where we obtained

$$\nabla_x f(x) = \left(\frac{\delta f(x)}{\delta x} \right)^T = \begin{bmatrix} 4.5000 & 4.5000 \\ 4.5000 & 4.5000 \end{bmatrix}$$

Hence, we have mathematically verified that autograd produces the correct answer for computing the derivatives of the scalar f with respect to the tensor x.

1.5 MNIST Data Preparation

Question 21

[28]:
```
import numpy as np
import MNISTtools
from matplotlib import pyplot

# Normalize MNIST Images
def normalize_MNIST_images(x):
    # Convert the uint8 input into float32 for ease of normalization
    fl_x = x.astype(np.float32)

    # Normalize [0 to 255] to [-1 to 1]
    # This means mapping 0 to -1, 255 to 1, and 127.5 to 0
    ret = 2*(fl_x - 255/2.0) / 255
    return ret
```

[29]:
```
# load the training data
xtrain, ltrain = MNISTtools.load(path='./datasets/MNIST')

# Normalize the training images
norm_x_train = normalize_MNIST_images(xtrain)
```

[30]:
```
# Load the testing data
xtest, ltest = MNISTtools.load(dataset='testing', path='./datasets/MNIST')

# Normalize the test images
norm_x_test = normalize_MNIST_images(xtest)
```

We have reused the code from Assignment 1 to load and normalize the MNIST testing and training data, and have converted the training and testing labels to one-hot encoded vectors.

Question 22

[31]:
```
# Reshape the normalized training dataset to torch format
train_reshaped = np.reshape(norm_x_train, (28, 28, 1, 60000))
```

```
test_reshaped = np.reshape(norm_x_test, (28, 28, 1, 10000))

print(train_reshaped.shape)
print(test_reshaped.shape)
```

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

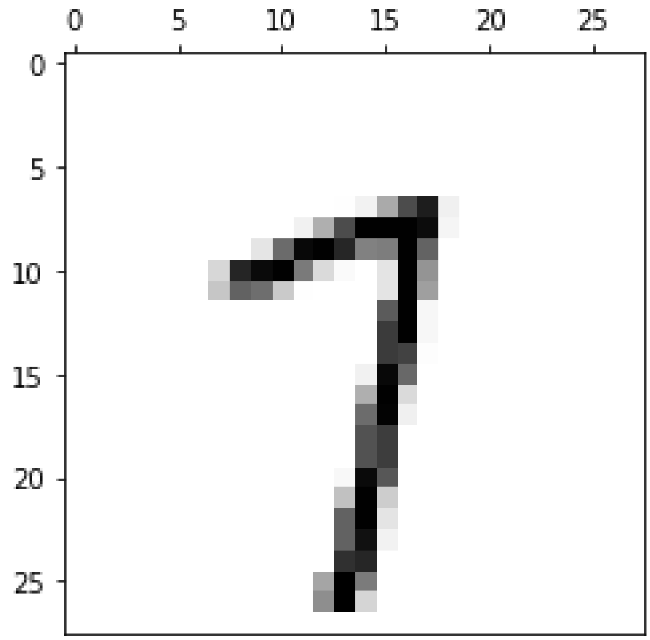
[32]:
```
# Make the numpy ndarrays compatible with torch format of Batch size * Number␣
↪of input channels * Image Height * Image width
xtrain = np.moveaxis(train_reshaped, [0, 1, 2, 3], [2, 3, 1, 0])
xtest = np.moveaxis(test_reshaped, [0, 1, 2, 3], [2, 3, 1, 0])

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

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

Hence, we have reorganised the numpy arrays *xtrain* and *xtest* such that they conform to the torch conventions for tensors, which is Batch size * Number of input channels * Image Height * Image width. As given in the hint, we initially reshaped the training data to the shape $(28, 28, 1, 60000)$ and the testing data to the shape $(28, 28, 1, 10000)$ respectively. We then used *np.moveaxis*() to move the 0^{th} dimension, i.e., the image height to 2^{nd} position, the 1^{st} dimension, i.e., the image width to 3^{rd} position, the 2^{nd} dimension, i.e., the number of channels to the 1^{st} position, and the 3^{rd} dimension, i.e., the batch size to the 0^{th} position.

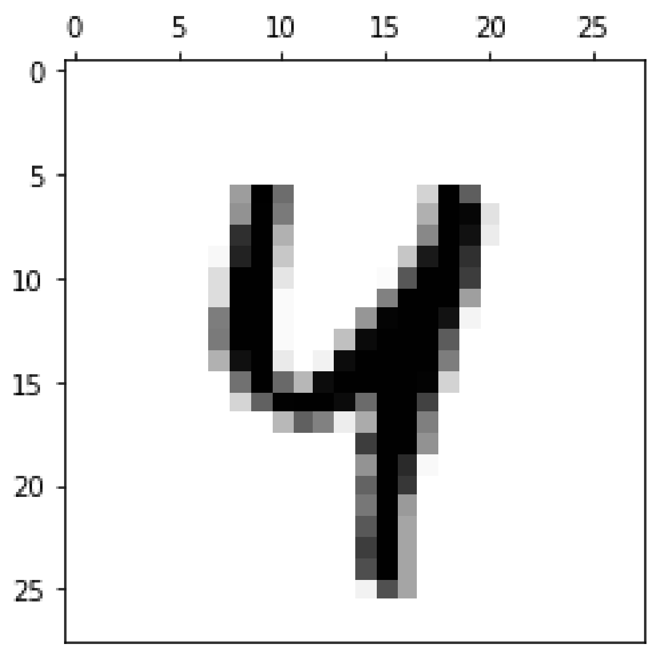
Question 23

[33]:
```
# check if our reorganization of training data was correct
MNISTtools.show(xtrain[42, 0, :, :])
print(ltrain[42])
```



7

[34]:
```
# check if our reorganization of testing data was correct
MNISTtools.show(xtest[42, 0, :, :])
print(ltest[42])
```



4

Hence, we have verified that our reorganization was indeed correct for both the training and testing set of images, as we displayed the images at index 42 for both, and the corresponding labels for both were correctly displayed.

Question 24

[35]:
```
# Wrap all the data into a torch tensor
xtrain = torch.from_numpy(xtrain)
ltrain = torch.from_numpy(ltrain)
xtest = torch.from_numpy(xtest)
ltest = torch.from_numpy(ltest)
```

[36]:
```
print(type(xtrain), xtrain.dtype, xtrain.size())
print(type(ltrain), ltrain.dtype, ltrain.size())
print(type(xtest), xtest.dtype, xtest.size())
print(type(ltest), ltest.dtype, ltest.size())
```

```
<class 'torch.Tensor'> torch.float32 torch.Size([60000, 1, 28, 28])
<class 'torch.Tensor'> torch.int64 torch.Size([60000])
<class 'torch.Tensor'> torch.float32 torch.Size([10000, 1, 28, 28])
<class 'torch.Tensor'> torch.int64 torch.Size([10000])
```

We have now converted the dataset into torch tensors

1.6 Convolutional Neural Network (CNN) for MNIST classification

Question 25 Given, our LeNet is composed of 2 convolutional layers, activated by ReLU followed by MaxPooling. Then we have 3 fully connected layers to get the output as 10 neurons in the final layer.

i. The first convolutional layer connects the input image to 6 feature maps with 5×5 convolutions ($K = 5$) and followed by ReLU. Since for input images of size W $\times$ H, the output feature maps have size [W $-$ K $+$ 1] $\times$ [H $-$ K $+$ 1], we have the size of the feature maps after the 1^{st} convolutional layer as $60000 \times 6 \times 24 \times 24$

ii. The 1^{st} maxpooling layer has ($L = 2$). Since for input images of size W $\times$ H, the output feature maps have size $\left[\frac{W}{L}\right] \times \left[\frac{H}{L}\right]$, we have the size of the feature maps after the 1^{st} maxpooling layer as $60000 \times 6 \times 12 \times 12$

iii. The second convolutional layer connects the 6 input channels to 16 output channels with 5×5 convolutions and followed by ReLU. Since for input images of size W $\times$ H, the output feature maps have size [W $-$ K $+$ 1] $\times$ [H $-$ K $+$ 1], we have the size of the feature maps after the 2^{nd} convolutional layer as $60000 \times 16 \times 8 \times 8$

iv. The 2^{nd} maxpooling layer has ($L = 2$). Since for input images of size W $\times$ H, the output feature maps have size $\left[\frac{W}{L}\right] \times \left[\frac{H}{L}\right]$, we have the size of the feature maps after the 2^{nd} maxpooling layer as $60000 \times 16 \times 4 \times 4$

v. The fully connected layer connects the the 16 feature maps to 120 output units. Since each feature map is of size 4×4, we have the total number of neurons as $16 * 4 * 4 = 256$. Hence, the third layer has 256 input units.

Question 26

[37]:
```python
import torch.nn as nn
import torch.nn.functional as F

# This is our neural network class that inherits from nn.Module
class LeNet(nn.Module):

    # Here we define our network structure
    def __init__(self):
        super(LeNet, self).__init__()

        # The first convolutional layer having 6 filters, each of size (5, 5)
        self.conv1 = nn.Conv2d(1, 6, 5)

        # The second convolutional layer having 16 filters each of size (5, 5)
        # that operates on the previous layer having 6 filters
        self.conv2 = nn.Conv2d(6, 16, 5)

        # The first fully connected layer from 16*4*4 after the maxpooling␣
↪after the second convolutional layer
        # to 120 units. So, we have 256 input units and 120 output units in␣
↪this Linear Layer
        self.fc1 = nn.Linear(256, 120)

        # The second fully connected layer from 120 inputs to 84 outputs
        self.fc2 = nn.Linear(120, 84)

        # The third fully connected layer from 84 inputs to 10 outputs
        self.fc3 = nn.Linear(84, 10)

    # Here, we define one forward pass through the network
    def forward(self, x):
        # We pass the input to the 1st convolutional layer. We next apply ReLU.
        # We finally perform (2*2) maxpooling on it
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))

        # We then pass this output to the 2nd convolutional layer. We next␣
↪apply ReLU.
        # We finally perform (2*2) maxpooling on it
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
```

```
        # We now flatten the tensors from the second maxpooling layer from␣
↪16*4*4 to 256
        x = x.view(-1, self.num_flat_features(x))

        # We then pass the flattened tensor x to the 1st Fully Connected Layer.␣
↪We next apply ReLU
        x = F.relu(self.fc1(x))

        # We next pass this through the next fully connected layer from 120 to␣
↪84 units
        x = F.relu(self.fc2(x))

        # We finally pass this output from 84 units to 10 units using the␣
↪Linear Layer
        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)
)
```

Network Definition

We have thus completed the code that defines the network architecture of our model. We defined the 1^{st} convolutional layer having 6 filters, each of size $(5, 5)$ using $nn.Conv2d(1, 6, 5)$. We then defined the 2^{nd} convolutional layer having 16 filters each of size $(5, 5)$. Each operates on the previous layer having 6 filters. Hence, we defined that layer using $nn.Conv2d(6, 16, 5)$. We defined the first fully connected layer from $16 * 4 * 4$ after the maxpooling after the 2^{nd} convolutional layer to 120 units. So, we have 256 input units and 120 output units in this Linear Layer. We define this layer using $nn.Linear(256, 120)$. The 2^{nd} fully connected layer is from 120 inputs to 84 outputs. We define it using $nn.Linear(120, 84)$. The 3^{rd} Linear layer is from 84 inputs to 10 outputs. We define it using the $nn.Linear(84, 10)$.

Forward Pass

The forward pass through the network is run as follows. We pass the input to the 1^{st} convolutional layer. We next apply ReLU. We finally perform $(2*2)$ maxpooling on it. We use $F.max\_pool2d(F.relu(self.conv1(x)),(2,2))$ for this purpose. We then pass this output to the 2^{nd} convolutional layer. We next apply ReLU. We finally perform $(2*2)$ maxpooling on it. We use $F.max\_pool2d(F.relu(self.conv2(x)),(2,2))$. We now flatten the tensors from the 2^{nd} maxpooling layer from $16*4*4$ to 256. We use $x.view(-1, self.num\_flat\_features(x))$. The $num\_flat\_features(x)$ function returns the number of neurons that are present if we flatten the tensor x. We then pass the flattened tensor x to the 1^{st} Fully Connected Layer. We next apply ReLU. We use $F.relu(self.fc1(x))$. We next pass this through the next fully connected layer from 120 to 84 units. We use $F.relu(self.fc2(x))$. We finally pass this output from 84 units to 10 units using the Linear Layer. We use $self.fc3(x)$. Hence, we have interpreted and completed the code for initializing our LeNet network

Question 27

[38]:
```
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 as returned by the $net.named\_parameters()$ function is as follows:-

$conv1.weight, conv1.bias, conv2.weight, conv2.bias, fc1.weight, fc1.bias, fc2.weight, fc2.bias, fc3.weight, fc3.bia$

These are the weights and biases for the 2 Convolution layers, as well as for all the 3 fully connected layers. The $requires\_grad$ attribute is set to True for all the parameters, and hence, the gradients are going to be tracked for all the parameters.

Question 28

[39]:
```
with torch.no_grad():
    yinit = net(xtest)
```

[40]:
```
_, lpred = yinit.max(1)

# calculate the precentage of correctly predicted labels for the initial run
init_acc = 100 * (ltest == lpred).float().mean()
print(init_acc)
```

```
tensor(11.9200)
```

We have computed the labels for the randomly initialized network, and then computed the percentage of correctly predicted samples and displayed it.

Question 29 and Question 30

[41]:
```
def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    # training set size
    N = xtrain.size()[0]

    # number of minibatches
    NB = int((N+B-1)/B)
    criterion = nn.CrossEntropyLoss()

    # learning rate lr is gamma, which is the step size
    optimizer = torch.optim.SGD(net.parameters(), lr=gamma, momentum=rho)

    for epoch in range(T):
        running_loss = 0.0

        # shuffle the indices to access the data
        shuffled_indices = np.random.permutation(range(N))
        for k in range(NB):
            # Extract the k-th minibatch from xtrain and ltrain
            # get the shuffled indices for a given minibatch
            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)

            # Backpropagation
            loss.backward()

            # Parameter update
            optimizer.step()

            # Print the averaged loss per minibatch every 100 minibatches
            # 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
```

[42]:
```
net = LeNet()
backprop_deep(xtrain, ltrain, net, T=3)
```

```
[1,   100] loss: 2.301
[1,   200] loss: 2.292
[1,   300] loss: 2.282
[1,   400] loss: 2.264
[1,   500] loss: 2.228
[1,   600] loss: 2.139
[2,   100] loss: 1.833
[2,   200] loss: 1.205
[2,   300] loss: 0.644
[2,   400] loss: 0.442
[2,   500] loss: 0.364
[2,   600] loss: 0.314
[3,   100] loss: 0.273
[3,   200] loss: 0.261
[3,   300] loss: 0.243
[3,   400] loss: 0.228
[3,   500] loss: 0.223
[3,   600] loss: 0.209
```

We have implemented *backprop_deep*() to complete training the network. The backpropagation occurs for T epochs using mini-batch stochastic gradient descent with momentum. We forward propogate, then we calculate the loss, backpropogate the gradients and finally update the parameters. We evaluate the average loss per 100 minibatches, and then print it.

3 epochs training

We ran the SGD with momentum = 0.9 and with learning rate = 0.001 on the training set for 3 epochs. The loss reduced from 2.301 after 100 minibatches of the 1^{st} epoch to 0.209 at the end of 3 epochs.

Question 31

[43]:
```
with torch.no_grad():
    yfin = net(xtest)
```

[44]:
```
_, lpred_fin = yfin.max(1)

# calculate the precentage of correctly predicted labels in the test set after␣
 ↪training for 3 epochs
fin_acc = 100 * (ltest == lpred_fin).float().mean()
```

```python
print(fin_acc)
```

```
tensor(94.4100)
```

Thus, we have re-evaluated the predictions of our trained network on the testing dataset.

3 epochs training vs. testing

We tested the performance of the network parameters trained on the dataset by SGD with momentum using the testing set. Final accuracy on the testing set is 94.4100%. Initial accuracy on the testing set was 11.9200%. Thus, the accuracy of our network in classifying MNIST images of the testing set improved by 82.49% after training for 3 epochs on the training set using minibatch Stochastic Gradient Descent with momentum = 0.9.

Question 32

[49]:
```python
# initialize the net and move it to the GPU
net = LeNet().to(device)

# Move all the relevant tensors to GPU
xtrain = xtrain.to(device)
xtest = xtest.to(device)
ltrain = ltrain.to(device)
ltest = ltest.to(device)

# Calculate the initial accuracy percentage of classification of images on the
↪testing set
with torch.no_grad():
    yinit_gpu = net(xtest)

# Calculate the predictions now.
_, lpred_gpu = yinit_gpu.max(1)

# calculate the precentage of correctly predicted labels for the initial run on
↪the GPU
init_acc_gpu = 100 * (ltest == lpred_gpu).float().mean()
print(init_acc_gpu)
```

```
tensor(9.6900, device='cuda:0')
```

[50]:
```python
# Run backprop on this network for 10 epochs
backprop_deep(xtrain, ltrain, net, T=10)
```

```
[1,   100] loss: 2.301
[1,   200] loss: 2.297
[1,   300] loss: 2.293
[1,   400] loss: 2.287
[1,   500] loss: 2.279
```

```
[1,   600] loss: 2.265
[2,   100] loss: 2.237
[2,   200] loss: 2.168
[2,   300] loss: 1.926
[2,   400] loss: 1.345
[2,   500] loss: 0.762
[2,   600] loss: 0.530
[3,   100] loss: 0.419
[3,   200] loss: 0.363
[3,   300] loss: 0.320
[3,   400] loss: 0.297
[3,   500] loss: 0.269
[3,   600] loss: 0.249
[4,   100] loss: 0.220
[4,   200] loss: 0.207
[4,   300] loss: 0.196
[4,   400] loss: 0.191
[4,   500] loss: 0.185
[4,   600] loss: 0.176
[5,   100] loss: 0.161
[5,   200] loss: 0.149
[5,   300] loss: 0.153
[5,   400] loss: 0.148
[5,   500] loss: 0.139
[5,   600] loss: 0.140
[6,   100] loss: 0.125
[6,   200] loss: 0.132
[6,   300] loss: 0.115
[6,   400] loss: 0.117
[6,   500] loss: 0.116
[6,   600] loss: 0.122
[7,   100] loss: 0.115
[7,   200] loss: 0.113
[7,   300] loss: 0.106
[7,   400] loss: 0.101
[7,   500] loss: 0.108
[7,   600] loss: 0.094
[8,   100] loss: 0.094
[8,   200] loss: 0.097
[8,   300] loss: 0.098
[8,   400] loss: 0.096
[8,   500] loss: 0.096
[8,   600] loss: 0.092
[9,   100] loss: 0.095
[9,   200] loss: 0.091
[9,   300] loss: 0.081
[9,   400] loss: 0.079
[9,   500] loss: 0.083
```

```
[9,   600] loss: 0.082
[10,   100] loss: 0.080
[10,   200] loss: 0.074
[10,   300] loss: 0.078
[10,   400] loss: 0.084
[10,   500] loss: 0.078
[10,   600] loss: 0.087
```

We reinitialized a new network and transferred it to the GPU to train. We simply used the $.to(device)$ function to accomplish this task. The accuracy on the testing dataset was calculated initially and displayed.

3 epochs GPU

3 epochs on the GPU reduced the loss from 2.303 initially after 100 minibatches of 1^{st} epoch to 0.277

3 epochs without GPU

3 epochs without GPU reduced the loss from 2.301 initially after 100 minibatches of 1^{st} epoch to 0.209

10 epochs GPU

10 epochs on the GPU reduced the loss from 2.301 initially after 100 minibatches of 1^{st} epoch to 0.087

Question 33

[51]:
```python
with torch.no_grad():
    yfin_gpu = net(xtest)

_, lpred_fin_gpu = yfin_gpu.max(1)

# calculate the precentage of correctly predicted labels in the test set after␣
↪training for 3 epochs on GPU
fin_acc_gpu = 100 * (ltest == lpred_fin_gpu).float().mean()
print(fin_acc_gpu)
```

```
tensor(98.1700, device='cuda:0')
```

Thus, we have re-evaluated the predictions of our network trained on the GPU on the testing set.

3 epochs GPU - Test vs. Train

We tested the performance of the network parameters that were trained for 3 epochs on the GPU and we got an accuracy percentage of 92.7000% on this test set. Training loss after 3 epochs on the GPU was: 0.277.

3 epochs without GPU vs. 3 epochs with GPU

Testing accuracy percentage after training without GPU for 3 epochs was : 94.4100%

10 epochs GPU - Test vs. Train

We tested the performance of the network parameters that were trained for 10 epochs on the GPU and we got an accuracy percentage of 98.1700% on this test set. Training loss after 10 epochs was: 0.087.

10 epochs GPU vs 3 epochs with and without GPU

Compared to 3 epochs on the GPU, our training accuracy using 10 epochs on GPU improved by 5.47% from 92.7000% to 98.1700%. Compared to 3 epochs without GPU, our training accuracy using 10 epochs on GPU improved by 3.76% from 94.4100% to 98.1700%.

1.6.1 Observations

After training for 3 epochs without GPU, our network had a training loss of 0.209. Training the network parameters with GPU for 3 epochs, the training loss was 0.277. After training for 3 epochs without GPU, our network had accuracy percentage as 94.4100% on the testing set. After training for 3 epochs with GPU, our network had accuracy percentage as 92.7000% on the testing set. Training the network parameters with GPU for 10 epochs, the training loss was 0.087. After training for 10 epochs with GPU, our network had accuracy percentage as 98.1700% on the testing set.

1.6.2 Inferences

Comparing the performance of the network using GPU vs. not using it, we conclude that training using GPU does **not** have any significant increase in performance in terms of loss or accuracy for training on the same number of epochs without GPU. However, we find that the training the network on the GPU is significantly faster than training the network without a GPU. This would hence allow us to increase the number of epochs that we would be able to run a network, and thus increase the overall performance of our methods.

1.7 Conclusion

Thus, we have learnt about Convolutional Neural Networks, and implemented LeNet in PyTorch to classify MNIST handwritten images. We trained the whole network without GPU, as well as trained it using GPU and compared their performance.

Assignment completed by:-Name: Anirudh Swaminathan PID: A53316083 Email ID: aswamina@eng.ucsd.edu