

ECE 285 Assignment #3

CNNs and PyTorch

Mehrnaz Motamed

A53245061

1. Convolutional Neural Networks (CNNs)

2. PyTorch

3. Getting started

In [1]:

```
from __future__ import print_function
import numpy as np
import torch
```

4. Tensors

1. Construct a 5×3 tensor and print it with the following commands

How was x initialized? What is the type of x?

```
In [2]:
```

```
x = torch.Tensor(5, 3)
print(x)

0.0000e+00  0.0000e+00  0.0000e+00
0.0000e+00 -7.9701e+29  4.5647e-41
-8.0133e+29  4.5647e-41 -1.0967e+30
 4.5647e-41 -8.0135e+29  4.5647e-41
 0.0000e+00  0.0000e+00  0.0000e+00
[torch.FloatTensor of size 5x3]
```

```
In [3]:
```

```
type(x)
```

```
Out[3]:
```

```
torch.FloatTensor
```

torch.Tensor(5, 3) will create a Tensor with uninitialized memory. The memory can contain any garbage, as it is uninitialized. As it is shown, x is a float tensor of dimension 5,3.

2. Construct a randomly initialized matrix:

```
In [4]:
```

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

0.5843  0.6462  0.8903
0.9531  0.2722  0.6158
0.3343  0.4886  0.4324
0.5589  0.1457  0.2577
0.7562  0.6952  0.7575
[torch.FloatTensor of size 5x3]
```

How are the random values distributed? What is the type of x? What if we use instead y = torch.randn(5, 3)?

```
In [5]:
```

```
type(y)
```

```
Out[5]:
```

```
torch.FloatTensor
```

```
In [6]:
```

```
type(x)
```

```
Out[6]:
```

```
torch.FloatTensor
```

```
In [7]:
```

```
y = torch.randn(5, 3)
print(y)
```

```
0.6685 -0.3817 -0.4677
-0.5518 -0.5978  1.0372
 0.0527  1.1290 -0.7025
 1.4336 -0.3639  0.0673
-0.8869  1.6635 -2.5619
[torch.FloatTensor of size 5x3]
```

```
In [8]:
```

```
type(y)
```

```
Out[8]:
```

```
torch.FloatTensor
```

***torch.rand(5,3)** returns a tensor filled with random numbers from a **uniform** distribution on the interval between 0 and 1, while

torch.randn(5,3) returns a tensor filled with random numbers from a **normal** distribution with mean 0 and variance 1 (also called the standard normal distribution).*

3. Run the following

```
In [9]:
```

```
x = x.double()
y = y.double()
print(x)
print(y)
```

```
0.0000e+00  0.0000e+00  0.0000e+00
0.0000e+00 -7.9701e+29  4.5647e-41
-8.0133e+29  4.5647e-41 -1.0967e+30
 4.5647e-41 -8.0135e+29  4.5647e-41
 0.0000e+00  0.0000e+00  0.0000e+00
[torch.DoubleTensor of size 5x3]
```

```
0.6685 -0.3817 -0.4677
-0.5518 -0.5978  1.0372
 0.0527  1.1290 -0.7025
 1.4336 -0.3639  0.0673
-0.8869  1.6635 -2.5619
[torch.DoubleTensor of size 5x3]
```

```
In [10]:
```

```
type(x)
```

```
Out[10]:
```

```
torch.DoubleTensor
```

```
In [11]:
```

```
type(y)
```

```
Out[11]:
```

```
torch.DoubleTensor
```

4. We can also initialize directly tensors with prescribed values. Create the following two tensors. What are their shape?

```
In [12]:
```

```
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]])
```

```
In [13]:
```

```
x
```

```
Out[13]:
```

```
-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  
[torch.FloatTensor of size 5x3]
```

```
In [14]:
```

```
y
```

```
Out[14]:
```

```
 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.FloatTensor of size 5x3]
```

```
In [15]:
```

```
x.shape
```

```
Out[15]:
```

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

```
In [16]:
```

```
y.shape
```

```
Out[16]:
```

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

5. You can stack the two 2d tensors in a 3d tensor as

```
In [17]:
```

```
z = torch.stack((x, y))
```

What is the shape of z?

How does it compare to torch.cat((x, y), 0) and torch.cat((x, y), 1)?

```
In [18]:
```

```
z.shape
```

```
Out[18]:
```

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

```
In [19]:
```

```
type(z)
```

```
Out[19]:
```

```
torch.FloatTensor
```

In [20]:

```
z
```

Out[20]:

```
(0 ,,,) =
-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

(1 ,,,) =
 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.FloatTensor of size 2x5x3]
```

In [21]:

```
torch.cat((x, y), 0)
```

Out[21]:

```
-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.FloatTensor of size 10x3]
```

In [22]:

```
torch.cat((x, y), 0).shape
```

Out[22]:

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

In [23]:

```
torch.cat((x, y), 1)
```

Out[23]:

```
-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.FloatTensor of size 5x6]
```

```
In [24]:
```

```
torch.cat((x, y), 1).shape
```

```
Out[24]:
```

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

As one can see, `torch.stack((x, y))` stacks x, y in a 253 tensor, while `torch.cat((x, y), 0)` and `torch.cat((x, y), 1)` Concatenate x, y in the given dimension (0 or 1) and produce a 103 or 56 tensor (twice the original size).

6. Report the value of the element at the 5th row and 3rd column in the 2d tensor y, and try accessing the same element in the 3d tensor z.

Hint: You can use standard NumPy-like indexing on all tensors.

```
In [25]:
```

```
y[4,2]
```

```
Out[25]:
```

```
1.3209999799728394
```

```
In [26]:
```

```
z[1,4,2]
```

```
Out[26]:
```

```
1.3209999799728394
```

7. Similarly print all the elements corresponding to the 5th row and 3rd column in z. How many elements are there?

```
In [27]:
```

```
z[:,4,2]
```

```
Out[27]:
```

```
0.1092  
1.3210  
[torch.FloatTensor of size 2]
```

8. Adding two tensors is fairly straightforward. There are multiple syntaxes for this operation. Sum x and y (from question 4) as follows

In [28]:

```
print(x + y)
print(torch.add(x, y))
print(x.add(y))
torch.add(x, y, out=x)
print(x)

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
[torch.FloatTensor of size 5x3]
```

```
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
[torch.FloatTensor of size 5x3]
```

```
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
[torch.FloatTensor of size 5x3]
```

```
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
[torch.FloatTensor of size 5x3]
```

Check whether the above instructions are printing the same output? Are they equivalent?

In [29]:

```
(x+y).shape
```

Out[29]:

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

In [30]:

```
(x.add(y)).shape
```

Out[30]:

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

They are practically doing the same operation in different ways. The outputs are the same, and the operations are equivalent.

9. To reshape a tensor, you can use torch.view:

In [31]:

```
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])
```

In [32]:

```
x
```

Out[32]:

```
0.2535 -0.4329  0.7510 -0.4980
-0.7724  1.3299  1.2768  0.8653
0.5311  1.2683 -0.1675  0.0805
0.7978  0.5122 -0.3073 -1.2762
[torch.FloatTensor of size 4x4]
```

In [33]:

```
y
```

Out[33]:

```
0.2535
-0.4329
0.7510
-0.4980
-0.7724
1.3299
1.2768
0.8653
0.5311
1.2683
-0.1675
0.0805
0.7978
0.5122
-0.3073
-1.2762
[torch.FloatTensor of size 16]
```

In [34]:

```
z
```

Out[34]:

```
0.2535 -0.4329  0.7510 -0.4980 -0.7724  1.3299  1.2768  0.8653
0.5311  1.2683 -0.1675  0.0805  0.7978  0.5122 -0.3073 -1.2762
[torch.FloatTensor of size 2x8]
```

Interpret the effect of each of these instructions. What does the -1 mean?

In [35]:

```
x.view(2, 8)
```

Out[35]:

```
0.2535 -0.4329  0.7510 -0.4980 -0.7724  1.3299  1.2768  0.8653  
0.5311  1.2683 -0.1675  0.0805  0.7978  0.5122 -0.3073 -1.2762  
[torch.FloatTensor of size 2x8]
```

*x.view(16) stacks the rows of x consecutively so that the produced tensor is 161. Also, x.view(-1, 8) stacks the rows to get a tensor with 8 columns.**

view(-1,8) behaves like -1 in numpy.reshape(), i.e. the actual value for this dimension will be inferred so that the number of elements in the view matches the original number of elements.

10. Generate 2 random tensors x and y of dimensions 10×10 and 2×100, respectively, resize them such that the instruction torch.mm(x, y) performs a row vector by matrix multiplication resulting in a row vector of dimensions 1 × 2.

In [36]:

```
x = torch.rand(10,10)  
y = torch.rand(2,100)  
x = x.view(1,100)  
y = y.view(100,2)  
torch.mm(x,y)
```

Out[36]:

```
23.1089  20.7065  
[torch.FloatTensor of size 1x2]
```

5. NumPy and PyTorch

11. Run the below code snippet and report your observation. What are the types of a and b?

In [37]:

```
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
```

```
1
1
1
1
1
[torch.FloatTensor of size 5]

[1. 1. 1. 1. 1.]
```

In [38]:

```
b.shape
```

Out[38]:

```
(5,)
```

In [39]:

```
type(b)
```

Out[39]:

```
numpy.ndarray
```

In [40]:

```
a.shape
```

Out[40]:

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

In [41]:

```
type(a)
```

Out[41]:

```
torch.FloatTensor
```

In [42]:

```
id(a)
```

Out[42]:

```
139912607003856
```

In [43]:

```
id(b)
```

Out[43]:

```
139912604844240
```

`a.numpy()` converts a tensor of size [5] to the corresponding array of size (5,).

12. Let us add one to the first element of the tensor a from the previous question and report the new values of a and b.

In [44]:

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

```
2
1
1
1
1
[torch.FloatTensor of size 5]

[2. 1. 1. 1. 1.]
```

Do they match? Do they share their underlying memory locations?

In [45]:

```
id(a)
```

Out[45]:

```
139912607003856
```

In [46]:

```
id(b)
```

Out[46]:

```
139912604844240
```

Changing a will result in changing b. Ofcourse, `id(a)` is not equal to `id(b)`; but, `id(a)` matched the original `id(a)` and `id(b)` matches the original `id(b)`.

13. Compare the effect of each of these three instructions separately:

In [47]:

```
# a.add_(1)
# a[:] += 1
# a = a.add(1)
```

For each of them, report the new values of a and b. What are the similarities and differences?

```
In [48]:
```

```
a = torch.ones(5);
b = a.numpy()
a[0] += 1;
print(a)
print(id(a))
#print(a.add_(1))
print(id(a.add_(1)))
print(a)
print(b)
```

```
2
1
1
1
1
[torch.FloatTensor of size 5]
```

```
139912607282672
139912607282672
```

```
3
2
2
2
2
[torch.FloatTensor of size 5]
```

```
[3. 2. 2. 2. 2.]
```

a.add(1) adds 1 to all the elements of a. Everytime a.add(1) is written in the code, this operation will be conducted once more. This instruction affects the original value of a (the result will be stored in the same ID). Ultimately, it will influence the values of b.

```
In [49]:
```

```
a = torch.ones(5);
b = a.numpy()
print(id(a))
a[0] += 1;
a[:] += 1
print(a)
print(id(a))
print(b)
```

```
139912607282024
```

```
3
2
2
2
2
[torch.FloatTensor of size 5]
```

```
139912607282024
[3. 2. 2. 2. 2.]
```

This will again affect the original value of *a* and *b*.

In [50]:

```
a = torch.ones(5);
b = a.numpy()
print(id(a))
a[0] += 1;
a = a.add(1)
print(id(a))
print(a)
print(b)
```

```
139912607282528
139912607282960
```

```
3
2
2
2
2
[torch.FloatTensor of size 5]

[2. 1. 1. 1. 1.]
```

using *a=a.add(1)* will create a new *a* in a new location (ID), so that *b* won't be changed. As you can see, the values of *a* has changed, while the values of *b* remained intact.

14. Let us study the reverse operation, converting NumPy arrays to a Torch tensor. Run the following code and examine the output.

In [51]:

```
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2.]

2
2
2
2
2
[torch.DoubleTensor of size 5]
```

In [52]:

```
b.shape
```

Out[52]:

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

```
In [53]:
```

```
a.shape
```

```
Out[53]:
```

```
(5,)
```

```
In [54]:
```

```
type(a)
```

```
Out[54]:
```

```
numpy.ndarray
```

```
In [55]:
```

```
type(b)
```

```
Out[55]:
```

```
torch.DoubleTensor
```

15. Torch tensors can be moved onto GPU device's memory using the .cuda method and back to CPU device with .cpu method. Run the following script

```
In [56]:
```

```
x = torch.randn(5, 3)
y = torch.randn(5, 3)
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
z=x+y
print(z)
if torch.cuda.is_available():
    print (z.cpu())
```

```
0.2076  0.4763 -0.1425
0.1606 -1.1519 -0.5523
0.0387 -1.4600  0.6628
-1.8752 -0.9028  0.6055
-0.9179  2.2516  1.5181
[torch.cuda.FloatTensor of size 5x3 (GPU 0)]
```

```
0.2076  0.4763 -0.1425
0.1606 -1.1519 -0.5523
0.0387 -1.4600  0.6628
-1.8752 -0.9028  0.6055
-0.9179  2.2516  1.5181
[torch.FloatTensor of size 5x3]
```

Interpret each of these instructions. Would this program produce the same result if it was run on a CPU-only architecture?

In [57]:

```
x = torch.randn(5, 3)
print(type(x))
print(id(x))
print(type(x.cuda()))
print(id(x.cuda()))

<class 'torch.FloatTensor'>
139912606523264
<class 'torch.cuda.FloatTensor'>
139912606523192
```

x = torch.randn(5, 3) creates a 5 by 3 tensor of random numbers. Then x = x.cuda() creates a cuda. Now that x and y are both cuda, z=x+y is done using the GPU. Next, we are converting z back to a tensor by using z.cpu(). Note that before using cuda() and cpu(), the first step is to determine whether the GPU should be used or not. A common pattern is to use Python's argparse module to read in user arguments, and have a flag that can be used to disable CUDA, in combination with is_available(). This is why the commands are written if torch.cuda.is_available().

GPU can render images much faster than CPUs. On the other hand, GPU results are all single precision – double precision is roughly 2 times slower on latest hardware; this can produce a slight different (difference in least significant digit) between results generated by CPU and GPU.

16. Run and interpret the result of the following two instructions

In [58]:

```
print(z.cpu().numpy())
print(type(z.cpu().numpy()))
print(z.numpy())

[[ 0.20759903  0.47631752 -0.14253587]
 [ 0.16056311 -1.1519263 -0.5523215 ]
 [ 0.03871652 -1.4599797  0.6627856 ]
 [-1.8751838 -0.9028124  0.60553503]
 [-0.9178691  2.2515786  1.5181298 ]]
<type 'numpy.ndarray'>
```

```
RuntimeErrorTraceback (most recent call last)
<ipython-input-58-b5989bf8ebb9> in <module>()
      1 print(z.cpu().numpy())
      2 print(type(z.cpu().numpy()))
----> 3 print(z.numpy())
```

```
RuntimeError: can't convert CUDA tensor to numpy (it doesn't support
GPU arrays). Use .cpu() to move the tensor to host memory first.
```

In order to be able to convert the results to numpy array, we should first move the data to cpu. This is because data in GPU (cuda) cannot be transformed to numpy arrays.

6. Autograd: automatic differentiation

17. Run the following script

In [59]:

```
import torch.autograd as ag

x = ag.Variable(torch.ones(2, 2), requires_grad=True)
print(x)
y=x+2
print(y)
```

Variable containing:

```
1 1
1 1
[torch.FloatTensor of size 2x2]
```

Variable containing:

```
3 3
3 3
[torch.FloatTensor of size 2x2]
```

What is the type of y?

In [60]:

```
type(y)
```

Out[60]:

```
torch.autograd.variable.Variable
```

18. Run the following script

In [61]:

```
z = y*y*3
f = z.mean()
print(z, f)
```

Variable containing:

```
27 27
27 27
[torch.FloatTensor of size 2x2]
Variable containing:
27
[torch.FloatTensor of size 1]
```

Inspect the results and write the equation linking f with the four entries x1, x2, x3, x4 of x: $f = f(x_1, x_2, x_3, x_4)$

Let's assume x is an m by n matrix. Then:

$$f(x_1, x_2, x_3, x_4) = \text{mean}(z) = \frac{1}{mn} \sum_i z_i = \frac{1}{mn} \sum_i 3y_i^2 = \frac{1}{mn} \sum_i 3(x_i + 2)^2 = \frac{1}{mn} \sum_{i=1}^{mn} 3(x_i + 2)^2$$

Thus,

$$f(x_1, x_2, x_3, x_4) = \frac{3}{4} \sum_1^4 (x_i + 2)^2$$

19. Because the variable f contains a single scalar, we can now back-propagate the gradient of f with respect to x into the variable x as

$$\begin{aligned}\partial f / \partial x_i &= \frac{3}{4} \cdot 2 \cdot (x_i + 2) = \frac{3}{2} (x_i + 2) \\ \partial f / \partial x_i &= \frac{3}{4} \cdot 2 \cdot (1 + 2) = 18/4 = 4.5\end{aligned}$$

In [62]:

```
f.backward() # That's it!
```

In [63]:

```
print(x.grad)
```

```
Variable containing:  
4.5000 4.5000  
4.5000 4.5000  
[torch.FloatTensor of size 2x2]
```

Report what the gradient is.

$$\begin{aligned}\nabla_x f(x) &= (\partial f / \partial x)^T = [\partial f / \partial x_i]_{2 \times 2} \\ \nabla_x f(x) &= [4.5, 4.5; 4.5, 4.5]\end{aligned}$$

20. Based on question 18, try to figure out mathematically if autograd produces the correct answer by deriving, yourself, the partial derivatives for each xi:

As shown in 18, 19 the produced gradient is correct.

7. MNIST Data preparation

21. Reuse the code from Assignment #2 to load and normalize MNIST testing and training data.

In [64]:

```
import MNISTtools
MNISTtools.load
xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
from __future__ import division
def normalize_MNIST_images(x):
    x = x.astype(np.float64)
    min_x = min(x.flatten())
    range_x = max(x.flatten()) - min_x
    x = 2*(x - min_x) / range_x - 1
    return x
xtrain = normalize_MNIST_images(xtrain)
```

In [65]:

```
xtrain.shape
```

Out[65]:

```
(784, 60000)
```

In [66]:

```
ltrain.shape
```

Out[66]:

```
(60000,)
```

In [67]:

```
np.amax(np.amax(xtrain, axis=0), axis=0)
```

Out[67]:

```
1.0
```

In [68]:

```
np.amin(np.amin(xtrain, axis=0), axis=0)
```

Out[68]:

```
-1.0
```

22. Reorganise the tensors xtrain and xtest.

In [69]:

```
xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")
```

In [70]:

```
xtest.shape
```

Out[70]:

```
(784, 10000)
```

In [71]:

```
ltest.shape
```

Out[71]:

```
(10000,)
```

In [72]:

```
npamax(npamax(xtest, axis=0), axis=0)
```

Out[72]:

```
255
```

In [73]:

```
namin(namin(xtest, axis=0), axis=0)
```

Out[73]:

```
0
```

In [74]:

```
from __future__ import division
def normalize_MNIST_images(x):
    x = x.astype(np.float64)
    min_x = min(x.flatten())
    range_x = max(x.flatten()) - min_x
    x = 2*(x - min_x) / range_x - 1
    return x
xtest = normalize_MNIST_images(xtest)
```

In [75]:

```
npamax(npamax(xtest, axis=0), axis=0)
```

Out[75]:

```
1.0
```

In [76]:

```
namin(namin(xtest, axis=0), axis=0)
```

Out[76]:

```
-1.0
```

In [77]:

```
xtrain = xtrain.reshape((28, 28, 1, 60000))
```

In [78]:

```
xtrain = np.moveaxis(xtrain, [0, 1, 2, 3], [-2, -1, -3, -4])
```

```
In [79]:
```

```
xtrain.shape
```

```
Out[79]:
```

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

```
In [80]:
```

```
xtest = xtest.reshape((28, 28, 1, 10000))
```

```
In [81]:
```

```
xtest = np.moveaxis(xtest, [0, 1, 2, 3], [-2, -1, -3, -4])
```

```
In [82]:
```

```
xtest.shape
```

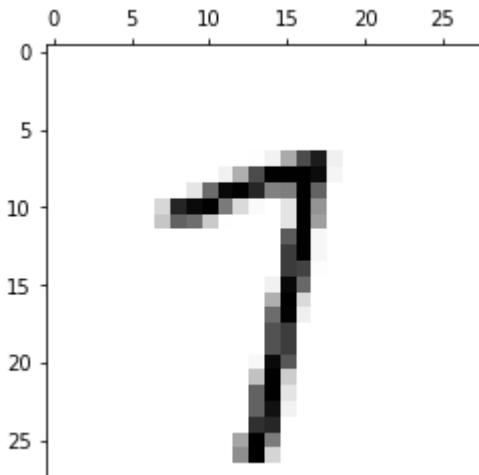
```
Out[82]:
```

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

23. Check that your data are well reorganized by making sure that display a digit that indeed corresponds to ltrain[42].

```
In [84]:
```

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



24. Now as studied in the previous section, we would like to convert these into autograd variable objects so that we can apply gradient propagation on them. Complete the following:

```
In [85]:
```

```
type(xtrain)
```

```
Out[85]:
```

```
numpy.ndarray
```

In [86]:

```
xtrain = ag.Variable(torch.from_numpy(xtrain), requires_grad=True)
ltrain = ag.Variable(torch.from_numpy(ltrain), requires_grad=False)
xtest = ag.Variable(torch.from_numpy(xtest), requires_grad=False)
```

8. Convolutional Neural Network (CNN) for MNIST classification

25. Determine the size of the feature maps after each convolution and maxpooling operation. How many input units have the third layer?

Input: xtrain: (60000, 1, 28, 28)

First convolutional layer:

6 feature maps, k = 5,

input images of size 28 by 28 ($W = H = 28$)

*output feature maps have size $[W - K + 1] \times [H - K + 1]$ which is **24 by 24**. Considering the default $L = 2$, the final output of the first layer is 6 feature maps of **12 by 12**.*

Second convolutional layer:

6 input channels to 16 output channels. k = 5,

6 input feature maps of 12 by 12.

*output is 16 channels of 8 by 8 features. Implementing Relu and maxpooling will result in **16 channels of 4 by 4**.*

*The third unit has 1616 input units = **256 input units**.*

connecting 16 feature maps to 120 output units.

26. Interpret and complete the following code initializing our LeNet network.

In [87]:

```
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).double()
        self.conv2 = nn.Conv2d(6, 16, 5).double()
        self.fc1 = nn.Linear(256, 120).double()
        self.fc2 = nn.Linear(120, 84).double()
        self.fc3 = nn.Linear(84, 10).double()

    # 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)
)
```

27. The learnable parameters of a model are returned by net.parameters(). Run the following and interpret the results:

In [88]:

```
params = list(net.parameters())
for i in range(len(params)):
    print(i, params[i].size())
```

```
0 torch.Size([6, 1, 5, 5])
1 torch.Size([6])
2 torch.Size([16, 6, 5, 5])
3 torch.Size([16])
4 torch.Size([120, 256])
5 torch.Size([120])
6 torch.Size([84, 120])
7 torch.Size([84])
8 torch.Size([10, 84])
9 torch.Size([10])
```

What are the parameters corresponding to indices 0 and 2? 4, 6, and 8? odd indices?

The parameters of even indices correspond to the layer dimensions [number of outputs, number of inputs, (convolution sizes if applicable)]. The parameters in the odd indices correspond to the number of output channels for each layer.

28. To run a forward pass of your initial network over your testing set, simply run:

In [89]:

```
yinit = net(xtest)
```

In [90]:

```
print(100 * np.mean(ltest == yinit.data.numpy().T.argmax(axis=0)))
```

13.36

This is the percentage of our initial outputs (corresponding to our test data) that match ltest(desired outputs). In other words, our initial performance without back propagation is around 10%.

29. We will use (Mini-Batch) Stochastic Gradient Descent (SGD) with cross-entropy and momentum. Complete the following section that sets up all parameters:

In [91]:

```
N = xtrain.size()[0] # Training set size
B = 100 # Minibatch size
NB = int(np.floor(N/B)) # Number of minibatches
T = 10 # Number of epochs
gamma = .001 # learning rate
rho = .9 # momentum
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(),
                            lr=gamma,
                            momentum=rho)
```

30. We have everything ready now to train our model with SGD and backprop. Complete each part in the piece of code below

In []:

```
for epoch in range(T):
    running_loss = 0.0
    idxminibatches = np.random.permutation(NB) # shuffling
    for k in range(NB):
        i = idxminibatches[k] # index of minibatch

        # Extract i-th minibatch from xtrain and ltrain
        idxsmp = np.arange(i*B, (i+1)*B-1) # indices of samples for i-th minibat
        ch
        inputs = xtrain[idxsmp]
        #print(inputs)
        labels = ltrain[idxsmp]
        #print(labels.shape)

        # Initialize the gradients to zero
        optimizer.zero_grad()

        # Forward propagation
        outputs = net(inputs)
        #print(outputs.shape)

        # Error evaluation
        loss = criterion(outputs, labels)

        # Back propagation
        loss.backward()
        #delta1 = xtrain.grad

        # Parameter update
        optimizer.step()

        # Print averaged loss per minibatch every 100 mini-batches
        running_loss += loss[0]
        if k%100==99:
            print( '[%d, %5d] loss: %.3f' %
                  (epoch + 1, k + 1, running_loss))
        running_loss = 0.0

print( 'Finished Training' )
```

31. Modify your code (Q24 and 26) such that all tensors and layers are moved to GPU and rerun. Make sure your code remains CPU compatible. The learning should run in less than 20 seconds!

In [92]:

```
import MNISTtools
MNISTtools.load
xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
from __future__ import division
def normalize_MNIST_images(x):
    x = x.astype(np.float64)
    min_x = min(x.flatten())
    range_x = max(x.flatten()) - min_x
    x = 2*(x - min_x) / range_x - 1
    return x
xtrain = normalize_MNIST_images(xtrain)
xtest, ltest = MNISTtools.load(dataset = "testing", path = "/datasets/MNIST")
from __future__ import division
def normalize_MNIST_images(x):
    x = x.astype(np.float64)
    min_x = min(x.flatten())
    range_x = max(x.flatten()) - min_x
    x = 2*(x - min_x) / range_x - 1
    return x
xtest = normalize_MNIST_images(xtest)
xtrain = xtrain.reshape((28, 28, 1, 60000))
xtrain = np.moveaxis(xtrain, [0, 1, 2, 3], [-2, -1, -3, -4])
xtest = xtest.reshape((28, 28, 1, 10000))
xtest = np.moveaxis(xtest, [0, 1, 2, 3], [-2, -1, -3, -4])
```

In [93]:

```
if torch.cuda.is_available():
    xtrain_cuda = ag.Variable(torch.from_numpy(xtrain).cuda(), requires_grad=True)
    ltrain_cuda = ag.Variable(torch.from_numpy(ltrain).cuda(), requires_grad=False)
    xtest_cuda = ag.Variable(torch.from_numpy(xtest).cuda(), requires_grad=False)
```

In [94]:

```
net_cuda = net.cuda()
```

In [95]:

```
criterion_cuda = criterion.cuda()
```

In [96]:

```
optimizer_cuda = torch.optim.SGD(net_cuda.parameters(),
                                  lr=gamma,
                                  momentum=rho)
```

In [97]:

```
if torch.cuda.is_available():
    params = list(net_cuda.parameters())
for i in range(len(params)):
    print(i, params[i].size())
```

```
0 torch.Size([6, 1, 5, 5])
1 torch.Size([6])
2 torch.Size([16, 6, 5, 5])
3 torch.Size([16])
4 torch.Size([120, 256])
5 torch.Size([120])
6 torch.Size([84, 120])
7 torch.Size([84])
8 torch.Size([10, 84])
9 torch.Size([10])
```

In [98]:

```
yinit_cuda = net_cuda(xtest_cuda)
```

In [99]:

```
if torch.cuda.is_available():
    yinit = yinit_cuda.cpu()
print(100 * np.mean(ltest == yinit.data.numpy().T.argmax(axis=0)))
```

13.36

In [100]:

```
loss_array = np.zeros((int(T*NB/B)))
loss_last = np.zeros((int(T*NB/B)))
for epoch in range(T):
    running_loss = 0.0
    idxminibatches = np.random.permutation(NB) # shuffling
    for k in range(NB):
        i = idxminibatches[k] # index of minibatch

        # Extract i-th minibatch from xtrain and ltrain
        idxsmp = np.arange(i*B, (i+1)*B-1) # indices of samples for i-th minibat
ch
        inputs = xtrain_cuda[idxsmp]
        #print(inputs)
        labels = ltrain_cuda[idxsmp]
        #print(labels.shape)

        # Initialize the gradients to zero
        if torch.cuda.is_available():
            inputs_cuda = inputs.cuda()
            labels_cuda = labels.cuda()

        if torch.cuda.is_available():
            optimizer_cuda.zero_grad()

        # Forward propagation
        outputs_cuda = net_cuda(inputs_cuda)
        #print(outputs.shape)

        # Error evaluation
        loss_cuda = criterion_cuda(outputs_cuda, labels_cuda)

        if torch.cuda.is_available():
            # Back propagation
            loss_cuda.backward()
            #delta1 = xtrain.grad

            # Parameter update
            optimizer_cuda.step()

        # Print averaged loss per minibatch every 100 mini-batches
        running_loss += loss_cuda[0]
        if k%100==99:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, k + 1, running_loss))
            #ind2 = int(epoch*6 + (k+1)/B)
            #loss_array[ind2-1] = running_loss
            #loss_last = np.copy(loss_array)
            running_loss = 0.0

print('Finished Training')
```

```
[1, 100] loss: 2.304
[1, 200] loss: 2.292
[1, 300] loss: 2.288
[1, 400] loss: 2.267
[1, 500] loss: 2.243
[1, 600] loss: 2.163
[2, 100] loss: 1.863
[2, 200] loss: 1.159
[2, 300] loss: 0.442
[2, 400] loss: 0.406
[2, 500] loss: 0.624
[2, 600] loss: 0.357
[3, 100] loss: 0.193
[3, 200] loss: 0.202
[3, 300] loss: 0.550
[3, 400] loss: 0.303
[3, 500] loss: 0.280
[3, 600] loss: 0.231
[4, 100] loss: 0.139
[4, 200] loss: 0.176
[4, 300] loss: 0.161
[4, 400] loss: 0.135
[4, 500] loss: 0.098
[4, 600] loss: 0.102
[5, 100] loss: 0.242
[5, 200] loss: 0.075
[5, 300] loss: 0.107
[5, 400] loss: 0.033
[5, 500] loss: 0.127
[5, 600] loss: 0.110
[6, 100] loss: 0.179
[6, 200] loss: 0.153
[6, 300] loss: 0.081
[6, 400] loss: 0.189
[6, 500] loss: 0.237
[6, 600] loss: 0.153
[7, 100] loss: 0.105
[7, 200] loss: 0.143
[7, 300] loss: 0.078
[7, 400] loss: 0.114
[7, 500] loss: 0.148
[7, 600] loss: 0.105
[8, 100] loss: 0.392
[8, 200] loss: 0.080
[8, 300] loss: 0.126
[8, 400] loss: 0.171
[8, 500] loss: 0.164
[8, 600] loss: 0.043
[9, 100] loss: 0.040
[9, 200] loss: 0.166
[9, 300] loss: 0.115
[9, 400] loss: 0.028
[9, 500] loss: 0.058
[9, 600] loss: 0.107
[10, 100] loss: 0.144
[10, 200] loss: 0.100
[10, 300] loss: 0.048
[10, 400] loss: 0.059
[10, 500] loss: 0.032
[10, 600] loss: 0.026
```

Finished Training

32. Re-evaluate the predictions of your trained network on the testing dataset. By how much did the accuracy improve?

In [101]:

```
y_last_cuda = net_cuda(xtest_cuda)

if torch.cuda.is_available():
    y_last = y_last_cuda.cpu()
print(100 * np.mean(ltest == y_last.data.numpy().T.argmax(axis=0)))
```

97.46000000000001

9. Bonus

33. Play around with the learning rate, momentum, batch size and number of epochs. How do they affect loss and training?

Learning rate:

In [102]:

```
B = 100 # Minibacth size
NB = int(np.floor(N/B)) # Number of minibatches
T = 10 # Number of epochs
gamma = .001 # learning rate
rho = .9 # momentum
```

In [103]:

```
# 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).double()
        self.conv2 = nn.Conv2d(6, 16, 5).double()
        self.fc1 = nn.Linear(256, 120).double()
        self.fc2 = nn.Linear(120, 84).double()
        self.fc3 = nn.Linear(84, 10).double()

    # 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 ()
```

In [104]:

```
list_of_gamma = [0.00001, 0.001, 0.01, 0.1, 1]
ind = 0
loss_array = np.zeros((len(list_of_gamma), int(T*N/B)))
loss_last = np.zeros((len(list_of_gamma), int(T*N/B)))
accuracy = np.zeros(len(list_of_gamma))
for gamma in list_of_gamma:
    net = LeNet()
    if torch.cuda.is_available():
        xtrain_cuda = ag.Variable(torch.from_numpy(xtrain).cuda(), requires_grad=True)
        ltrain_cuda = ag.Variable(torch.from_numpy(ltrain).cuda(), requires_grad=False)
        xtest_cuda = ag.Variable(torch.from_numpy(xtest).cuda(), requires_grad=False)

    net_cuda = net.cuda()
    criterion_cuda = criterion.cuda()
    optimizer_cuda = torch.optim.SGD(net_cuda.parameters(),
                                      lr=gamma,
                                      momentum=rho)

    if torch.cuda.is_available():
        params = list(net_cuda.parameters())

    yinit_cuda = net_cuda(xtest_cuda)

    # if torch.cuda.is_available():
    #     yinit = yinit_cuda.cpu()
    # error_initial = 100 * np.mean(ltest == yinit.data.numpy().T.argmax(axis=0))
    for epoch in range(T):
        running_loss = 0.0
        idxminibatches = np.random.permutation(NB) # shuffling
        for k in range(NB):
            i = idxminibatches[k] # index of minibatch

            # Extract i-th minibatch from xtrain and ltrain
            idxsmp = np.arange(i*B, (i+1)*B-1) # indices of samples for i-th minibatch
            inputs = xtrain_cuda[idxsmp]
            #print(inputs)
            labels = ltrain_cuda[idxsmp]
            #print(labels.shape)

            # Initialize the gradients to zero
            if torch.cuda.is_available():
                inputs_cuda = inputs.cuda()
                labels_cuda = labels.cuda()

            if torch.cuda.is_available():
                optimizer_cuda.zero_grad()

            # Forward propagation
            outputs_cuda = net_cuda(inputs_cuda)
            #print(outputs.shape)

            # Error evaluation
            loss_cuda = criterion_cuda(outputs_cuda, labels_cuda)

            if torch.cuda.is_available():
```

```
# Back propagation
loss_cuda.backward()
#delta1 = xtrain.grad

# Parameter update
optimizer_cuda.step()

# Print averaged loss per minibatch every 100 mini-batches
running_loss += loss_cuda[0]
if k%100==99:
    #print( '[%d, %5d] loss: %.3f' %
    #(epoch + 1, k + 1, running_loss))
    ind2 = int(epoch*6 + (k+1)/B)
    loss_array[ind,ind2-1] = running_loss
    loss_last[ind,:] = np.copy(loss_array[ind,:])

running_loss = 0.0

y_last_cuda = net_cuda(xtest_cuda)

if torch.cuda.is_available():
    y_last = y_last_cuda.cpu()
    accuracy[ind] = 100 * np.mean(ltest == y_last.data.numpy().T.argmax(axis
=0))
    ind = ind + 1
```

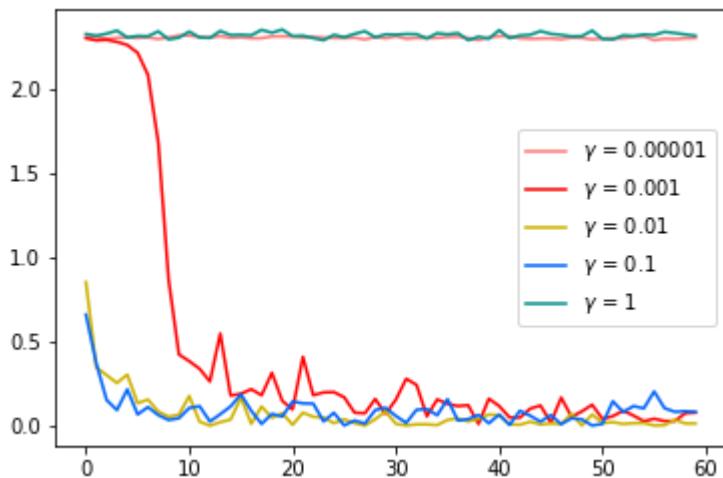
In [105]:

```
print(accuracy)
```

```
[11.17 97.81 98.94 97.97 10.1 ]
```

In [106]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
t = np.arange(0, T*NB/B)
ax.plot(t, loss_last[0,:], '#fe7b7c', label='$\gamma = 0.00001$')
ax.plot(t, loss_last[1,:], 'r', label='$\gamma = 0.001$')
ax.plot(t, loss_last[2,:], '#ceb301', label='$\gamma = 0.01$')
ax.plot(t, loss_last[3,:], '#0165fc', label='$\gamma = 0.1$')
ax.plot(t, loss_last[4,:], '#029386', label='$\gamma = 1$')
legend = ax.legend(loc='right')
plt.show()
```



As you can see, very low/high gamma will lead to high error and not converging at all. While, for gamma in a good interval, higher gamma will converge faster specially at first.

Momentum:

In [107]:

```
B = 100 # Minibatch size
NB = int(np.floor(N/B)) # Number of minibatches
T = 10 # Number of epochs
gamma = .001 # learning rate
rho = .9 # momentum
```

In [108]:

```
# 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).double()
        self.conv2 = nn.Conv2d(6, 16, 5).double()
        self.fc1 = nn.Linear(256, 120).double()
        self.fc2 = nn.Linear(120, 84).double()
        self.fc3 = nn.Linear(84, 10).double()

    # 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 ()
```

In [109]:

```
list_of_momentum = [0.1, 0.5, 0.7, 0.9, 1]
ind = 0
loss_array = np.zeros((len(list_of_momentum), int(T*NB/B)))
loss_last = np.zeros((len(list_of_momentum), int(T*NB/B)))
accuracy = np.zeros(len(list_of_momentum))
for rho in list_of_momentum:
    net = LeNet()
    if torch.cuda.is_available():
        xtrain_cuda = ag.Variable(torch.from_numpy(xtrain).cuda(), requires_grad=True)
        ltrain_cuda = ag.Variable(torch.from_numpy(ltrain).cuda(), requires_grad=False)
        xtest_cuda = ag.Variable(torch.from_numpy(xtest).cuda(), requires_grad=False)

    net_cuda = net.cuda()
    criterion_cuda = criterion.cuda()
    optimizer_cuda = torch.optim.SGD(net_cuda.parameters(),
                                      lr=gamma,
                                      momentum=rho)

    if torch.cuda.is_available():
        params = list(net_cuda.parameters())

    yinit_cuda = net_cuda(xtest_cuda)

#     if torch.cuda.is_available():
#         yinit = yinit_cuda.cpu()
#     error_initial = 100 * np.mean(ltest == yinit.data.numpy().T.argmax(axis=0))
    for epoch in range(T):
        running_loss = 0.0
        idxminibatches = np.random.permutation(NB) # shuffling
        for k in range(NB):
            i = idxminibatches[k] # index of minibatch

            # Extract i-th minibatch from xtrain and ltrain
            idxsmp = np.arange(i*B, (i+1)*B-1) # indices of samples for i-th minibatch
            inputs = xtrain_cuda[idxsmp]
            #print(inputs)
            labels = ltrain_cuda[idxsmp]
            #print(labels.shape)

            # Initialize the gradients to zero
            if torch.cuda.is_available():
                inputs_cuda = inputs.cuda()
                labels_cuda = labels.cuda()

            if torch.cuda.is_available():
                optimizer_cuda.zero_grad()

            # Forward propagation
            outputs_cuda = net_cuda(inputs_cuda)
            #print(outputs.shape)

            # Error evaluation
            loss_cuda = criterion_cuda(outputs_cuda, labels_cuda)

            if torch.cuda.is_available():
```

```

# Back propagation
loss_cuda.backward()
#delta1 = xtrain.grad

# Parameter update
optimizer_cuda.step()

# Print averaged loss per minibatch every 100 mini-batches
running_loss += loss_cuda[0]
if k%100==99:
    #print( '[%d, %5d] loss: %.3f' %
    #(epoch + 1, k + 1, running_loss))
    ind2 = int(epoch*6 + (k+1)/B)
    loss_array[ind,ind2-1] = running_loss
    loss_last[ind,:] = np.copy(loss_array[ind,:])

running_loss = 0.0

y_last_cuda = net_cuda(xtest_cuda)

if torch.cuda.is_available():
    y_last = y_last_cuda.cpu()
accuracy[ind] = 100 * np.mean(ltest == y_last.data.numpy().T.argmax(axis=0))
ind = ind + 1

```

In [110]:

```
print(accuracy)
```

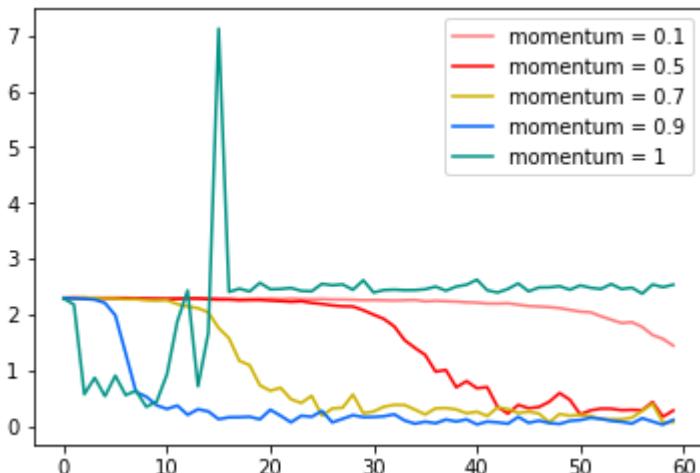
```
[76.47 91.75 94.52 97.73 10.28]
```

In [111]:

```

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
t = np.arange(0, T*NB/B)
ax.plot(t, loss_last[0,:], '#fe7b7c', label='momentum = 0.1')
ax.plot(t, loss_last[1,:], 'r', label='momentum = 0.5')
ax.plot(t, loss_last[2,:], '#ceb301', label='momentum = 0.7')
ax.plot(t, loss_last[3,:], '#0165fc', label='momentum = 0.9')
ax.plot(t, loss_last[4,:], '#029386', label='momentum = 1')
legend = ax.legend(loc='upper right')
plt.show()

```



For very high or low momentum, the error is very high. While for momentum in the proper interval, higher momentum will give us better accuracy and lower loss and faster convergence.

34. To prevent over-fitting, implement early stopping by using 10% of the training data for validation.

In []:

```
B = 100 # Minibatch size
NB = int(np.floor(N/B)) # Number of minibatches
T = 10 # Number of epochs
gamma = .001 # learning rate
rho = .9 # momentum
```

In []:

```
# 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).double()
        self.conv2 = nn.Conv2d(6, 16, 5).double()
        self.fc1 = nn.Linear(256, 120).double()
        self.fc2 = nn.Linear(120, 84).double()
        self.fc3 = nn.Linear(84, 10).double()

    # 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 ()
```

In []:

```
ind = 0
net = LeNet()
index_shuffle = np.random.permutation(N) # shuffling
for k in range(N):
    i = index_shuffle[k] # index of sample
    x_validation_sub = xtrain[0:int(N/10)-1]
    labels_validation_sub = ltrain[0:int(N/10)-1]
    x_train_sub = xtrain[int(N/10):N-1]
    labels_train_sub = ltrain[int(N/10):N-1]

if torch.cuda.is_available():
    xtrain_cuda = ag.Variable(torch.from_numpy(x_train_sub).cuda(), requires_grad=True)
    ltrain_cuda = ag.Variable(torch.from_numpy(labels_train_sub).cuda(), requires_grad=False)
    xtest_cuda = ag.Variable(torch.from_numpy(x_validation_sub).cuda(), requires_grad=False)

net_cuda = net.cuda()
criterion_cuda = criterion.cuda()
optimizer_cuda = torch.optim.SGD(net_cuda.parameters(),
                                  lr=gamma,
                                  momentum=rho)

if torch.cuda.is_available:
    params = list(net_cuda.parameters())

yinit_cuda = net_cuda(xtest_cuda)

#     if torch.cuda.is_available():
#         yinit = yinit_cuda.cpu()
#     error_initial = 100 * np.mean(ltest == yinit.data.numpy().T.argmax(axis=0))
accuracy = np.zeros((T))
for epoch in range(T):
    running_loss = 0.0
    idxminibatches = np.random.permutation(NB) # shuffling
    for k in range(NB):
        i = idxminibatches[k] # index of minibatch

        # Extract i-th minibatch from xtrain and ltrain
        idxsmp = np.arange(i*B, (i+1)*B-1) # indices of samples for i-th minibatch
        inputs = xtrain_cuda[idxsmp]
        #print(inputs)
        labels = ltrain_cuda[idxsmp]
        #print(labels.shape)

        # Initialize the gradients to zero
        if torch.cuda.is_available():
            inputs_cuda = inputs.cuda()
            labels_cuda = labels.cuda()

        if torch.cuda.is_available():
            optimizer_cuda.zero_grad()

        # Forward propagation
        outputs_cuda = net_cuda(inputs_cuda)
        #print(outputs.shape)
```

```

# Error evaluation
loss_cuda = criterion_cuda(outputs_cuda, labels_cuda)

if torch.cuda.is_available():
    # Back propagation
    loss_cuda.backward()
    #delta1 = xtrain.grad

    # Parameter update
    optimizer_cuda.step()

# Print averaged loss per minibatch every 100 mini-batches
running_loss += loss_cuda[0]
if k%100==99:
    #print( '[%d, %5d] loss: %.3f' %
        %(epoch + 1, k + 1, running_loss))
    ind2 = int(epoch*6 + (k+1)/B)
    loss_array[ind2-1] = running_loss
    loss_last[:] = np.copy(loss_array)

running_loss = 0.0
if epoch%2==0:
    y_last_cuda = net_cuda(xtest_cuda)
    if torch.cuda.is_available():
        y_last = y_last_cuda.cpu()
    accuracy[epoch] = 100 * np.mean(labels_validation_sub == y_last.
data.numpy().T.argmax(axis=0))
    if accuracy[epoch]<accuracy[epoch-1]:
        break

y_last_cuda = net_cuda(xtest_cuda)

if torch.cuda.is_available():
    y_last = y_last_cuda.cpu()
accuracy = 100 * np.mean(ltest == y_last.data.numpy().T.argmax(axis=0))

```