Lab 10: Convolutional Neural Networks

Today we'll experiment with CNNs. We'll start with a hand-coded CNN structure based on numpy, then we'll move to PyTorch.

Hand-coded CNN

This example is based on Ahmed Gad's tutorial (https://www.kdnuggets.com/2018/04/building-convolutional-neural-network-numpy-scratch.html).

We will implement a very simple CNN in numpy. The model will have just three layers, a convolutional layer (conv for short), a ReLU activation layer, and max pooling. The major steps involved are as follows.

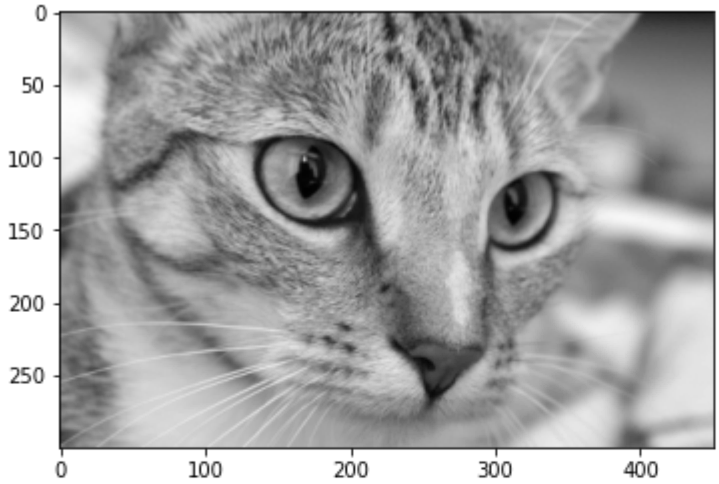
1. Reading the input image.
2. Preparing filters.
3. Conv layer: Convolving each filter with the input image.
4. ReLU layer: Applying ReLU activation function on the feature maps (output of conv layer).
5. Max Pooling layer: Applying the pooling operation on the output of ReLU layer.
6. Stacking the conv, ReLU, and max pooling layers.

Reading an input image

The following code reads an existing image using the SciKit-Image Python library and converts it into grayscale. You may need to `pip install scikit-image` .

In [1]:
```
import skimage.data
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

# Read image
img = skimage.data.chelsea()
print('Image dimensions:', img.shape)

# Convert to grayscale
img = skimage.color.rgb2gray(img)
plt.imshow(img, cmap='gray')
plt.show()
```

```
Image dimensions: (300, 451, 3)
```



Create some filters for the conv layer

Recall that a conv layer uses some number of convolution (actually cross correlation) filters, usually matching the number of channels in the input (1 in our case since the image is grayscale). Each kernel gives us one feature map (channel) in the result.

Let's make two 3×3 filters, using the horizontal and vertical Sobel edge filters:

In [2]:
```
l1_filters = np.zeros((2,3,3))
l1_filters[0, :, :] = np.array([[[-1, 0, 1],
                                 [-2, 0, 2],
                                 [-1, 0, 1]]])
l1_filters[1, :, :] = np.array([[[-1, -2, -1],
                                 [ 0,  0,  0],
                                 [ 1,  2,  1]]])
```

Conv layer feedforward step

Let's convolve the input image with our filters.

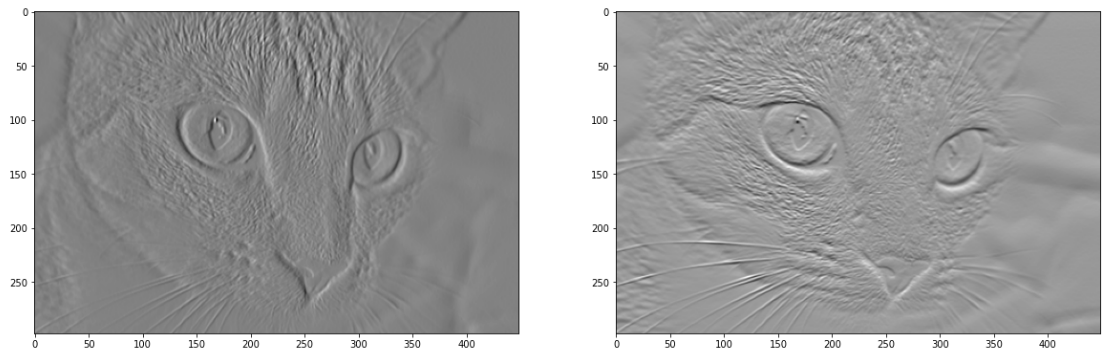
In [5]:
```
# Perform stride 1 cross correlation of an image and a filter. We
output the valid region only
# (no padding).
def convolve(img, conv_filter):
    stride = 1
    padding = 0
    filter_size = conv_filter.shape[1]
    results_dim = ((np.array(img.shape) - np.array(conv_filter.sh
ape) + (2*padding))/stride) + 1
    result = np.zeros((int(results_dim[0]), int(results_dim[1])))

    for r in np.arange(0, img.shape[0] - filter_size + 1):
        for c in np.arange(0, img.shape[1]-filter_size + 1):
            curr_region = img[r:r+filter_size,c:c+filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result)
            result[r, c] = conv_sum

    return result

# Perform convolution with a set of filters and return the result
def conv(img, conv_filters):
    # Check shape of inputs
    if len(img.shape) != len(conv_filters.shape) - 1:
        raise Exception("Error: Number of dimensions in conv filt
er and image do not match.")

    # Ensure filter depth is equal to number of channels in input
    if len(img.shape) > 2 or len(conv_filters.shape) > 3:
        if img.shape[-1] != conv_filters.shape[-1]:
            raise Exception("Error: Number of channels in both im
age and filter must match.")

    # Ensure filters are square
    if conv_filters.shape[1] != conv_filters.shape[2]:
        raise Exception('Error: Filter must be square (number of
rows and columns must match).')

    # Ensure filter dimensions are odd
    if conv_filters.shape[1]%2==0:
        raise Exception('Error: Filter must have an odd size (num
ber of rows and columns must be odd).')

    # Prepare output
    feature_maps = np.zeros((img.shape[0]-conv_filters.shape[1]+
1,
                             img.shape[1]-conv_filters.shape[1]+
1,
                             conv_filters.shape[0]))

    # Perform convolutions
    for filter_num in range(conv_filters.shape[0]):
        curr_filter = conv_filters[filter_num, :]
        # Our convolve function only handles 2D convolutions. If
the input has multiple channels, we
        # perform the 2D convolutions for each input channel sepa
rately then add them. If the input
```

```
        # has just a single channel, we do the convolution direct
ly.
        if len(curr_filter.shape) > 2:
            conv_map = convolve(img[:, :, 0], curr_filter[:, :,
0])
            for ch_num in range(1, curr_filter.shape[-1]):
                conv_map = conv_map + convolve(img[:, :, ch_num],
                                               curr_filter[:, :, ch_num])
        else:
            conv_map = convolve(img, curr_filter)
        feature_maps[:, :, filter_num] = conv_map

    return feature_maps
```

Let's give it a try:

In [6]:
```
features = conv(img, l1_filters)
%timeit conv(img,l1_filters)

print('Convolutional feature maps shape:', features.shape)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
ax1.imshow(features[:,:,0], cmap='gray')
ax2.imshow(features[:,:,1], cmap='gray')
plt.show()
```

```
1.33 s ± 7.88 ms per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)
Convolutional feature maps shape: (298, 449, 2)
```



See the time, what is different? :-)

Cool, right? A couple observations:

1. We've hard coded the values in the filters, so they are sensible to us. In a real CNN, we'd be tuning the filters to minimize loss on the training set, so we wouldn't expect such perfectly structured results.
2. Naive implementation of 2D convolutions requires 4 nested loops, which is super slow in Python. In the code above, we've replaced the two inner loops with an element-by-element matrix multiplication for the kernel and the portion of the image applicable for the current indices into the convoution result.

Exercise (15 points)

The semi-naive implementation of the convolution function above could be sped up with the use of a fast low-level 2D convolution routine that makes the best possible use of the CPU's optimized instructions, pipelining of operations, etc. Take a look at [Laurent Perrinet's blog on 2D convolution implementations (https://laurentperrinet.github.io/sciblog/posts/2017-09-20-the-fastest-2d-convolution-in-the-world.html)](https://laurentperrinet.github.io/sciblog/posts/2017-09-20-the-fastest-2d-convolution-in-the-world.html) and see how the two fastest implementations, scikit and numpy, outperform other methods and should vastly outperform the Python loop above. Reimplement the `convolve()` function above to be `convole2()` and compare the times taken by the naive and optimized versions of your convolution operation for the cat image. In your report, briefly describe the experiment and the results you obtained.

- Do faster CNN (10 points)
- Describe the experiment and the results (5 points)

Hint:

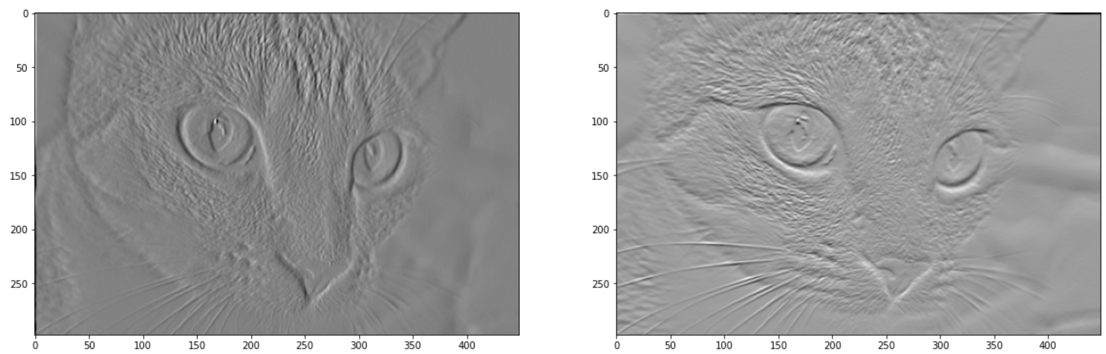
In [12]:
```
from numpy.fft  import fft2, ifft2
def convolve2(img, conv_filter):
    output = None
    ### BEGIN SOLUTION
    B = np.flip(conv_filter,axis=(0,1))
    fr = fft2(img[B.shape[0]-1:,B.shape[0]-1:])
    fr2 = fft2(B,s=fr.shape)
    output = np.real(ifft2(fr*fr2))
    ### END SOLUTION
    return output

def conv2(img, conv_filters):
    # Check shape of inputs
    if len(img.shape) != len(conv_filters.shape) - 1:
        raise Exception("Error: Number of dimensions in conv filt
er and image do not match.")

    # Ensure filter depth is equal to number of channels in input
    if len(img.shape) > 2 or len(conv_filters.shape) > 3:
        if img.shape[-1] != conv_filters.shape[-1]:
            raise Exception("Error: Number of channels in both im
age and filter must match.")

    # Ensure filters are square
    if conv_filters.shape[1] != conv_filters.shape[2]:
        raise Exception('Error: Filter must be square (number of
rows and columns must match).')

    # Ensure filter dimensions are odd
    if conv_filters.shape[1]%2==0:
        raise Exception('Error: Filter must have an odd size (num
ber of rows and columns must be odd).')

    # Prepare output
    feature_maps = np.zeros((img.shape[0]-conv_filters.shape[1]+
1,
                             img.shape[1]-conv_filters.shape[1]+
1,
                             conv_filters.shape[0]))

    # Perform convolutions
    ### BEGIN SOLUTION
    for filter_num in range(conv_filters.shape[0]):
        curr_filter = conv_filters[filter_num, :]
        # Our convolve function only handles 2D convolutions. If
the input has multiple channels, we
        # perform the 2D convolutions for each input channel sepa
rately then add them. If the input
        # has just a single channel, we do the convolution direct
ly.
        if len(curr_filter.shape) > 2:
            conv_map = convolve2(img[:, :, 0], curr_filter[:, :,
0])
            for ch_num in range(1, curr_filter.shape[-1]):
                conv_map = conv_map + convolve2(img[:, :, ch_nu
m],
                                                curr_filter[:, :, ch_num])
```

```
        else:
            conv_map = convolve2(img, curr_filter)
        feature_maps[:, :, filter_num] = conv_map
    ### END SOLUTION

    return feature_maps
```

In [17]:

```
import datetime
start = datetime.datetime.now()
features = conv2(img, l1_filters)
stop = datetime.datetime.now()
%timeit conv2(img,l1_filters)

c = stop - start
elapsed = c.microseconds / 1000 # millisec

print('Convolutional feature maps shape:', features.shape)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
ax1.imshow(features[:,:,0], cmap='gray')
ax2.imshow(features[:,:,1], cmap='gray')
plt.show()

# Test function: Do not remove
assert elapsed < 200, "Convolution is too slow, try again"
print("success!")
# End Test function
```

```
31.2 ms ± 324 µs per loop (mean ± std. dev. of 7 runs, 10 loops e
ach)
Convolutional feature maps shape: (298, 449, 2)
```




```
success!
```

Describe the experiment and the results here!

YOUR TEXT HERE

Pooling and relu

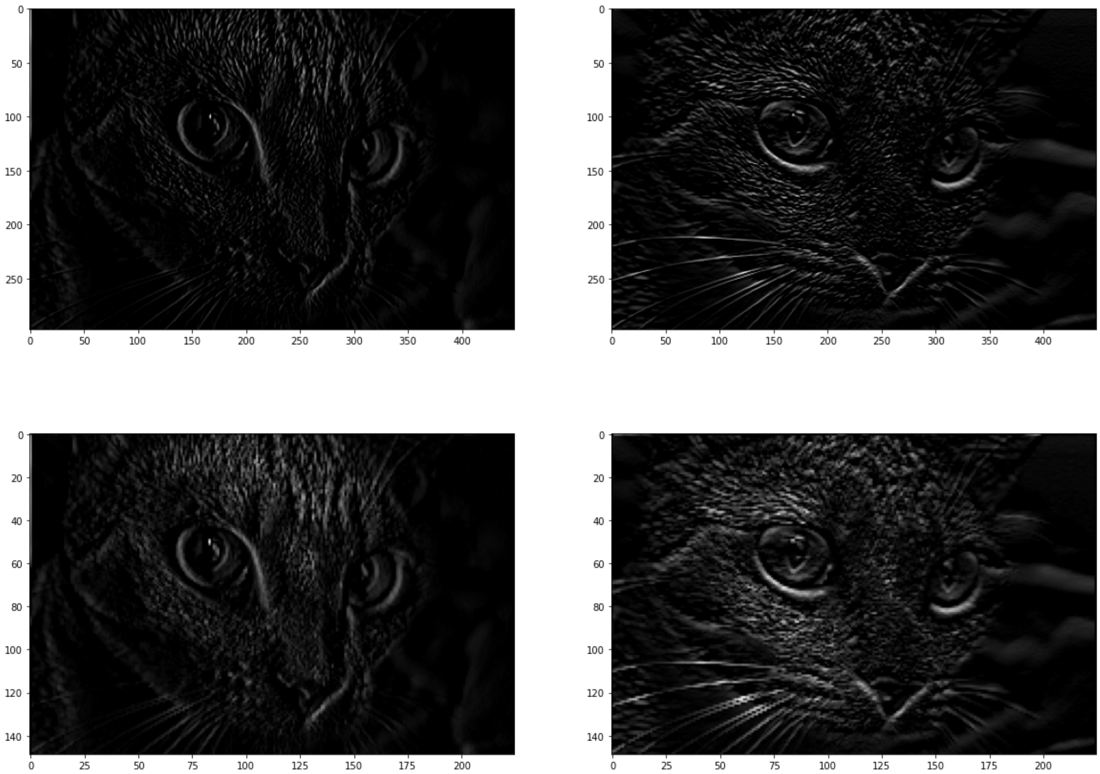
Next, consider the feedforward pooling and ReLU operations.

In [18]:
```
# Pooling layer with particular size and stride

def pooling(feature_map, size=2, stride=2):
    pool_out = np.zeros((np.uint16((feature_map.shape[0]-size+1)/
stride+1),
                         np.uint16((feature_map.shape[1]-size+1)/
stride+1),
                         feature_map.shape[-1]))
    for map_num in range(feature_map.shape[-1]):
        r2 = 0
        for r in np.arange(0,feature_map.shape[0]-size+1, strid
e):
            c2 = 0
            for c in np.arange(0, feature_map.shape[1]-size+1, st
ride):
                pool_out[r2, c2, map_num] = np.max([feature_map
[r:r+size,  c:c+size, map_num]])
                c2 = c2 + 1
            r2 = r2 +1
    return pool_out

# ReLU activation function

def relu(feature_map):
    relu_out = np.zeros(feature_map.shape)
    for map_num in range(feature_map.shape[-1]):
        for r in np.arange(0,feature_map.shape[0]):
            for c in np.arange(0, feature_map.shape[1]):
                relu_out[r, c, map_num] = np.max([feature_map[r,
c, map_num], 0])
    return relu_out
```

Now let's try ReLU and pooling:

In [19]:

```
relued_features = relu(features)
pooled_features = pooling(relued_features)

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(20, 1
5))
ax1.imshow(relued_features[:,:,0], cmap='gray')
ax2.imshow(relued_features[:,:,1], cmap='gray')
ax3.imshow(pooled_features[:,:,0], cmap='gray')
ax4.imshow(pooled_features[:,:,1], cmap='gray')
plt.show()
```



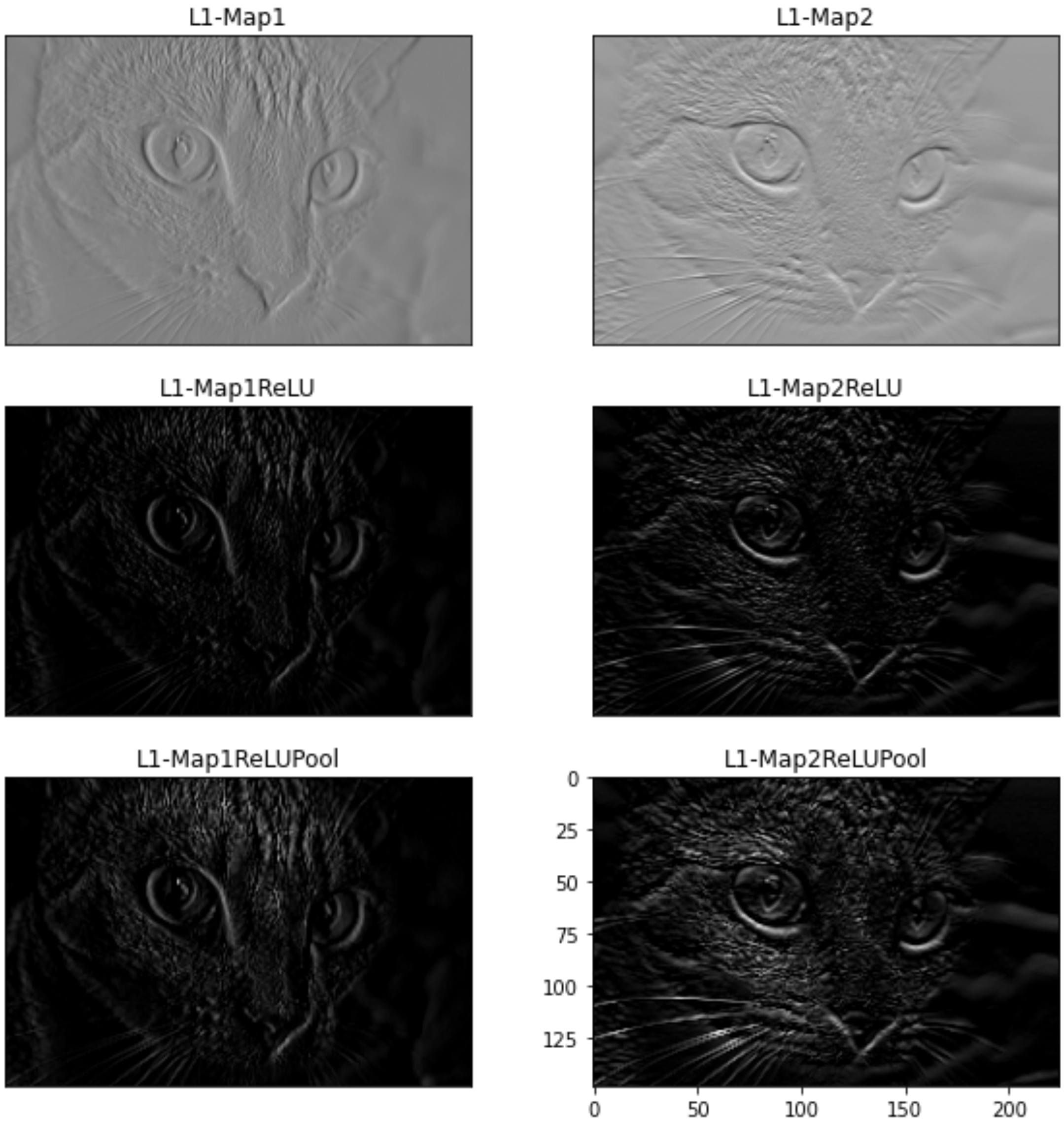
Let's visualize all of the feature maps in the model...

In [20]:
```
# First conv layer

import sys

np.set_printoptions(threshold=sys.maxsize)

print("conv layer 1...")
l1_feature_maps = conv(img, l1_filters)
l1_feature_maps_relu = relu(l1_feature_maps)
l1_feature_maps_relu_pool = pooling(l1_feature_maps_relu, 2, 2)

# Second conv layer

print("conv layer 2...")
l2_filters = np.random.rand(3, 5, 5, l1_feature_maps_relu_pool.sh
ape[-1])
l2_feature_maps = conv(l1_feature_maps_relu_pool, l2_filters)
l2_feature_maps_relu = relu(l2_feature_maps)
l2_feature_maps_relu_pool = pooling(l2_feature_maps_relu, 2, 2)
#print(l2_feature_maps)

# Third conv layer

print("conv layer 3...")
l3_filters = np.random.rand(1, 7, 7, l2_feature_maps_relu_pool.sh
ape[-1])
l3_feature_maps = conv(l2_feature_maps_relu_pool, l3_filters)
l3_feature_maps_relu = relu(l3_feature_maps)
l3_feature_maps_relu_pool = pooling(l3_feature_maps_relu, 2, 2)
```

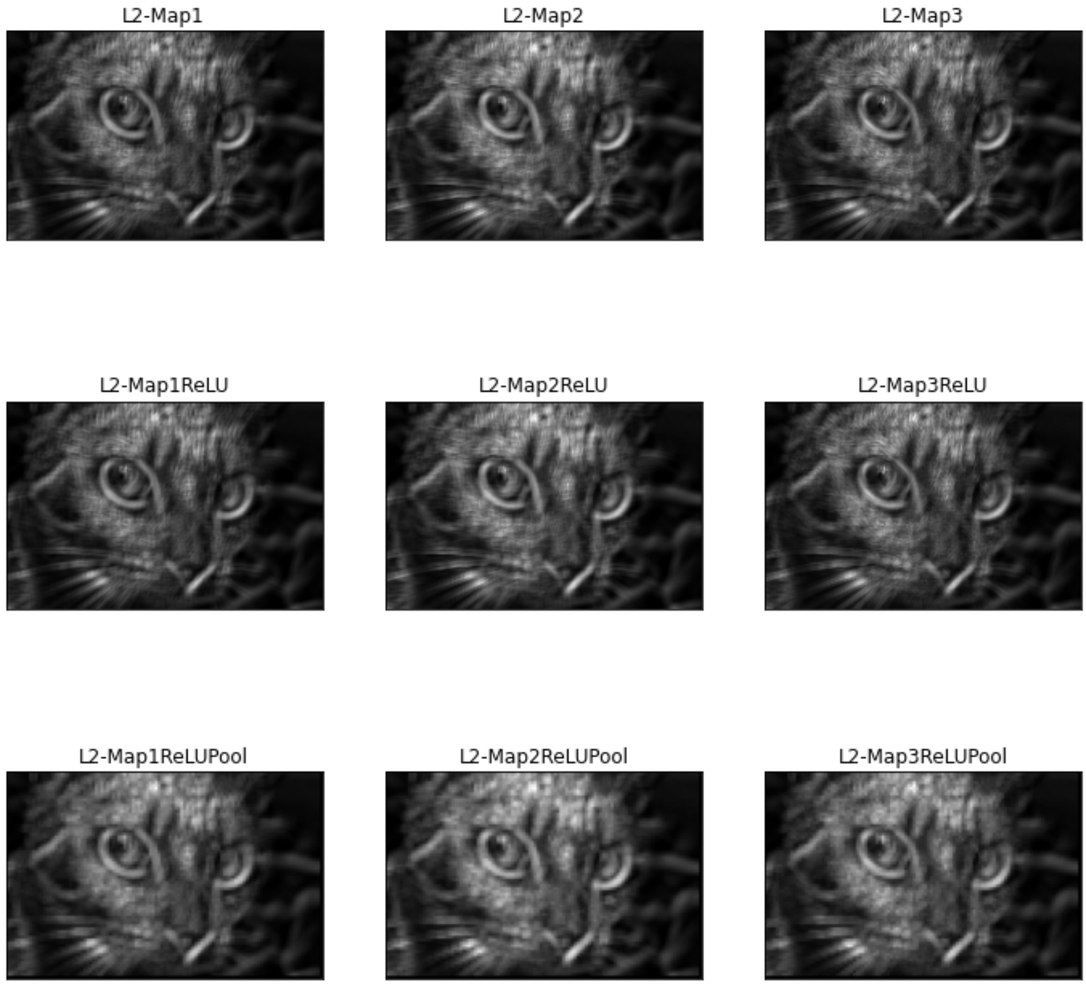
```
conv layer 1...
conv layer 2...
conv layer 3...
```

In [21]:
```
# Show results

fig0, ax0 = plt.subplots(nrows=1, ncols=1)
ax0.imshow(img).set_cmap("gray")
ax0.set_title("Input Image")
ax0.get_xaxis().set_ticks([])
ax0.get_yaxis().set_ticks([])
plt.show()
```

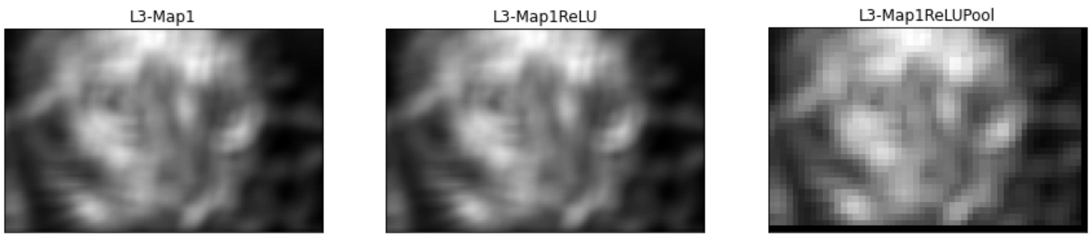
In [22]:
```
# Layer 1
fig1, ax1 = plt.subplots(nrows=3, ncols=2)
fig1.set_figheight(10)
fig1.set_figwidth(10)
ax1[0, 0].imshow(l1_feature_maps[:, :, 0]).set_cmap("gray")
ax1[0, 0].get_xaxis().set_ticks([])
ax1[0, 0].get_yaxis().set_ticks([])
ax1[0, 0].set_title("L1-Map1")

ax1[0, 1].imshow(l1_feature_maps[:, :, 1]).set_cmap("gray")
ax1[0, 1].get_xaxis().set_ticks([])
ax1[0, 1].get_yaxis().set_ticks([])
ax1[0, 1].set_title("L1-Map2")

ax1[1, 0].imshow(l1_feature_maps_relu[:, :, 0]).set_cmap("gray")
ax1[1, 0].get_xaxis().set_ticks([])
ax1[1, 0].get_yaxis().set_ticks([])
ax1[1, 0].set_title("L1-Map1ReLU")

ax1[1, 1].imshow(l1_feature_maps_relu[:, :, 1]).set_cmap("gray")
ax1[1, 1].get_xaxis().set_ticks([])
ax1[1, 1].get_yaxis().set_ticks([])
ax1[1, 1].set_title("L1-Map2ReLU")

ax1[2, 0].imshow(l1_feature_maps_relu_pool[:, :, 0]).set_cmap("gr
ay")
ax1[2, 0].get_xaxis().set_ticks([])
ax1[2, 0].get_yaxis().set_ticks([])
ax1[2, 0].set_title("L1-Map1ReLUPool")

ax1[2, 1].imshow(l1_feature_maps_relu_pool[:, :, 1]).set_cmap("gr
ay")
ax1[2, 0].get_xaxis().set_ticks([])
ax1[2, 0].get_yaxis().set_ticks([])
ax1[2, 1].set_title("L1-Map2ReLUPool")

plt.show()
```


L1-Map1
L1-Map2
L1-Map1ReLU
L1-Map2ReLU
L1-Map1ReLUPool
L1-Map2ReLUPool
0
25
50
75
100
125
0
50
100
150
200

In [23]:
```
# Layer 2
fig2, ax2 = plt.subplots(nrows=3, ncols=3)
fig2.set_figheight(12)
fig2.set_figwidth(12)
ax2[0, 0].imshow(l2_feature_maps[:, :, 0]).set_cmap("gray")
ax2[0, 0].get_xaxis().set_ticks([])
ax2[0, 0].get_yaxis().set_ticks([])
ax2[0, 0].set_title("L2-Map1")

ax2[0, 1].imshow(l2_feature_maps[:, :, 1]).set_cmap("gray")
ax2[0, 1].get_xaxis().set_ticks([])
ax2[0, 1].get_yaxis().set_ticks([])
ax2[0, 1].set_title("L2-Map2")

ax2[0, 2].imshow(l2_feature_maps[:, :, 2]).set_cmap("gray")
ax2[0, 2].get_xaxis().set_ticks([])
ax2[0, 2].get_yaxis().set_ticks([])
ax2[0, 2].set_title("L2-Map3")

ax2[1, 0].imshow(l2_feature_maps_relu[:, :, 0]).set_cmap("gray")
ax2[1, 0].get_xaxis().set_ticks([])
ax2[1, 0].get_yaxis().set_ticks([])
ax2[1, 0].set_title("L2-Map1ReLU")

ax2[1, 1].imshow(l2_feature_maps_relu[:, :, 1]).set_cmap("gray")
ax2[1, 1].get_xaxis().set_ticks([])
ax2[1, 1].get_yaxis().set_ticks([])
ax2[1, 1].set_title("L2-Map2ReLU")

ax2[1, 2].imshow(l2_feature_maps_relu[:, :, 2]).set_cmap("gray")
ax2[1, 2].get_xaxis().set_ticks([])
ax2[1, 2].get_yaxis().set_ticks([])
ax2[1, 2].set_title("L2-Map3ReLU")

ax2[2, 0].imshow(l2_feature_maps_relu_pool[:, :, 0]).set_cmap("gr
ay")
ax2[2, 0].get_xaxis().set_ticks([])
ax2[2, 0].get_yaxis().set_ticks([])
ax2[2, 0].set_title("L2-Map1ReLUPool")

ax2[2, 1].imshow(l2_feature_maps_relu_pool[:, :, 1]).set_cmap("gr
ay")
ax2[2, 1].get_xaxis().set_ticks([])
ax2[2, 1].get_yaxis().set_ticks([])
ax2[2, 1].set_title("L2-Map2ReLUPool")

ax2[2, 2].imshow(l2_feature_maps_relu_pool[:, :, 2]).set_cmap("gr
ay")
ax2[2, 2].get_xaxis().set_ticks([])
ax2[2, 2].get_yaxis().set_ticks([])
ax2[2, 2].set_title("L2-Map3ReLUPool")
plt.show()
```


L2-Map1
L2-Map2
L2-Map3
L2-Map1ReLU
L2-Map2ReLU
L2-Map3ReLU
L2-Map1ReLUPool
L2-Map2ReLUPool
L2-Map3ReLUPool

In [24]:
```
# Layer 3

fig3, ax3 = plt.subplots(nrows=1, ncols=3)
fig3.set_figheight(15)
fig3.set_figwidth(15)
ax3[0].imshow(l3_feature_maps[:, :, 0]).set_cmap("gray")
ax3[0].get_xaxis().set_ticks([])
ax3[0].get_yaxis().set_ticks([])
ax3[0].set_title("L3-Map1")

ax3[1].imshow(l3_feature_maps_relu[:, :, 0]).set_cmap("gray")
ax3[1].get_xaxis().set_ticks([])
ax3[1].get_yaxis().set_ticks([])
ax3[1].set_title("L3-Map1ReLU")

ax3[2].imshow(l3_feature_maps_relu_pool[:, :, 0]).set_cmap("gra
y")
ax3[2].get_xaxis().set_ticks([])
ax3[2].get_yaxis().set_ticks([])
ax3[2].set_title("L3-Map1ReLUPool")
plt.show()
```



We can see that at progressively higher layers of the network, we get coarser representations of the input. Since the filters at the later layers are random, they are not very structured, so we get a kind of blurring effect. These visualizations would be more meaningful in model with learned filters.

Exercise 2 (15 points)

Modify CNN 3 layer above with your `conv2()` function. Check the result and explain what you did and what is the different result.

In [26]:
```
### BEGIN SOLUTION
np.set_printoptions(threshold=sys.maxsize)

print("conv layer 1...")
l1_feature_maps = conv2(img, l1_filters)
l1_feature_maps_relu = relu(l1_feature_maps)
l1_feature_maps_relu_pool = pooling(l1_feature_maps_relu, 2, 2)

# Second conv layer

print("conv layer 2...")
l2_filters = np.random.rand(3, 5, 5, l1_feature_maps_relu_pool.sh
ape[-1])
l2_feature_maps = conv2(l1_feature_maps_relu_pool, l2_filters)
l2_feature_maps_relu = relu(l2_feature_maps)
l2_feature_maps_relu_pool = pooling(l2_feature_maps_relu, 2, 2)
#print(l2_feature_maps)

# Third conv layer

print("conv layer 3...")
l3_filters = np.random.rand(1, 7, 7, l2_feature_maps_relu_pool.sh
ape[-1])
l3_feature_maps = conv2(l2_feature_maps_relu_pool, l3_filters)
l3_feature_maps_relu = relu(l3_feature_maps)
l3_feature_maps_relu_pool = pooling(l3_feature_maps_relu, 2, 2)

# Show results

fig0, ax0 = plt.subplots(nrows=1, ncols=1)
ax0.imshow(img).set_cmap("gray")
ax0.set_title("Input Image")
ax0.get_xaxis().set_ticks([])
ax0.get_yaxis().set_ticks([])
plt.show()
### END SOLUTION
```

```
conv layer 1...
conv layer 2...
conv layer 3...
```

Input Image

Explanation

YOUR TEXT HERE

CNNs in PyTorch

Now we'll do a more complete CNN example using PyTorch. We'll use the MNIST digits again. The example is based on [Anand Saha's PyTorch tutorial (https://github.com/anandsaha/deep.learning.with.pytorch)](https://github.com/anandsaha/deep.learning.with.pytorch).

PyTorch has a few useful modules for us:

1. cuda: GPU-based tensor computations
2. nn: Neural network layer implementations and backpropagation via autograd
3. torchvision: datasets, models, and image transformations for computer vision problems.

torchvision itself includes several useful elements:

1. datasets: Datasets are subclasses of torch.utils.data.Dataset. Some of the common datasets available are "MNIST," "COCO," and "CIFAR." In this example we will see how to load MNIST dataset using a custom subclass of the datasets class.
2. transforms - Transforms are used for image transformations. The MNIST dataset from torchvision is in PIL image. To convert MNIST images to tensors, we will use `transforms.ToTensor()` .

In [27]:
```
import torch
import torch.cuda as cuda
import torch.nn as nn

from torch.autograd import Variable

from torchvision import datasets
from torchvision import transforms

# The functional module contains helper functions for defining ne
ural network layers as simple functions
import torch.nn.functional as F
```

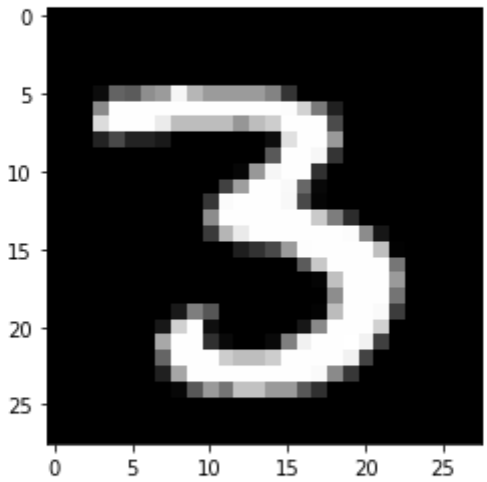
Load the MNIST data

First, let's load the data and transfrom the input elements (pixels) so that their mean over the entire training dataset is 0 and its standard deviation is 1.

In [28]:
```
# Desired mean and standard deviation

mean = 0.0
stddev = 1.0

# Transform input image

transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((mean), (stdde
v))])

mnist_train = datasets.MNIST('./data', train=True, download=True,
transform=transform)
mnist_valid = datasets.MNIST('./data', train=False, download=Tru
e, transform=transform)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ub
yte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/
MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ub
yte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/
MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-uby
te.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/M
NIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-uby
te.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/M
NIST/raw
Processing...
Done!
```

In [29]:
```
img = mnist_train[12][0].numpy()
plt.imshow(img.reshape(28, 28), cmap='gray')
plt.show()
```



In [30]:
```
label = mnist_train[12][1]
print('Label of image above:', label)

# Reduce batch size if you get out-of-memory error

batch_size = 1024
mnist_train_loader = torch.utils.data.DataLoader(mnist_train, bat
ch_size=batch_size, shuffle=True, num_workers=1)
mnist_valid_loader = torch.utils.data.DataLoader(mnist_valid, bat
ch_size=batch_size, shuffle=True, num_workers=1)
```

```
Label of image above: 3
```

Define the NN model

We use 2 convolutional layers followed by 2 fully connected layers. The input size of each image is (28,28,1). We will use stide of size 1 and padding of size 0.

For first convolution layer we will apply 20 filters of size (5,5). CNN output formula

$$\text{output size} = \frac{W - F + 2P}{S} + 1$$

where W - input, F - filter size, P - padding size and S - stride size.

We get $\frac{(28,28,1)-(5,5,1)+(2*0)}{1} + 1$ for each filter, so for 10 filters we get output size of (24,24,10).

The ReLU activation function is applied to the output of the first convolutional layer.

For the second convolutional layer, we apply 20 filters of size (5,5), giving us output of size of (20,20,20). Maxpooling with a size of 2 is applied to the output of the second convolutional layer, thereby giving us an output size of of (10,10,20). The ReLU activation function is applied to the output of the maxpooling layer.

Next we have two fully connected layers. The input of the first fully connected layer is flattened output of $10 * 10 * 20 = 2000$, with 50 nodes. The second layer is the output layer and has 10 nodes.

In [31]:
```
class CNN_Model(nn.Module):

    def __init__(self):
        super().__init__()

        # NOTE: All Conv2d layers have a default padding of 0 and
stride of 1,
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)      # 24 x
24 x 20  (after 1st convolution)
        self.relu1 = nn.ReLU()                            # Same
as above

        # Convolution Layer 2
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)     # 20 x
20 x 20  (after 2nd convolution)
        #self.conv2_drop = nn.Dropout2d(p=0.5)            # Dropo
ut is a regularization technqiue we discussed in class
        self.maxpool2 = nn.MaxPool2d(2)                   # 10 x
10 x 20  (after pooling)
        self.relu2 = nn.ReLU()                            # Same
as above

        # Fully connected layers
        self.fc1 = nn.Linear(2000, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):

        # Convolution Layer 1
        x = self.conv1(x)
        x = self.relu1(x)

        # Convolution Layer 2
        x = self.conv2(x)
        #x = self.conv2_drop(x)
        x = self.maxpool2(x)
        x = self.relu2(x)

        # Switch from activation maps to vectors
        x = x.view(-1, 2000)

        # Fully connected layer 1
        x = self.fc1(x)
        x = F.relu(x)
        #x = F.dropout(x, training=True)

        # Fully connected layer 2
        x = self.fc2(x)

        return x
```

Create the objects

In [32]:
```
# The model
net = CNN_Model()

if cuda.is_available():
    net = net.cuda()

# Our loss function
criterion = nn.CrossEntropyLoss()

# Our optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, m
omentum=0.9)
```

Training loop

In [33]:
```
num_epochs = 20

train_loss = []
valid_loss = []
train_accuracy = []
valid_accuracy = []

for epoch in range(num_epochs):

    ############################
    # Train
    ############################

    iter_loss = 0.0
    correct = 0
    iterations = 0

    net.train()                  # Put the network into training 
mode

    for i, (items, classes) in enumerate(mnist_train_loader):

        # Convert torch tensor to Variable
        items = Variable(items)
        classes = Variable(classes)

        # If we have GPU, shift the data to GPU
        if cuda.is_available():
            items = items.cuda()
            classes = classes.cuda()

        optimizer.zero_grad()     # Clear off the gradients from 
any past operation
        outputs = net(items)      # Do the forward pass
        loss = criterion(outputs, classes) # Calculate the loss
        iter_loss += loss.item() # Accumulate the loss
        loss.backward()           # Calculate the gradients with 
help of back propagation
        optimizer.step()          # Ask the optimizer to adjust t
he parameters based on the gradients

        # Record the correct predictions for training data
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == classes.data).sum()
        iterations += 1

    # Record the training loss
    train_loss.append(iter_loss/iterations)
    # Record the training accuracy
    train_accuracy.append((100 * correct / float(len(mnist_train_
loader.dataset))))


    ############################
    # Validate - How did we do on the unseen dataset?
    ############################
```

```
    loss = 0.0
    correct = 0
    iterations = 0

    net.eval()                    # Put the network into evaluate mode

    for i, (items, classes) in enumerate(mnist_valid_loader):

        # Convert torch tensor to Variable
        items = Variable(items)
        classes = Variable(classes)

        # If we have GPU, shift the data to GPU
        if cuda.is_available():
            items = items.cuda()
            classes = classes.cuda()

        outputs = net(items)      # Do the forward pass
        loss += criterion(outputs, classes).item() # Calculate the loss

        # Record the correct predictions for training data
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == classes.data).sum()

        iterations += 1

    # Record the validation loss
    valid_loss.append(loss/iterations)
    # Record the validation accuracy
    correct_scalar = np.array([correct.clone().cpu()])[0]
    valid_accuracy.append(correct_scalar / len(mnist_valid_loader.dataset) * 100.0)

    print ('Epoch %d/%d, Tr Loss: %.4f, Tr Acc: %.4f, Val Loss: %.4f, Val Acc: %.4f'
           %(epoch+1, num_epochs, train_loss[-1], train_accuracy[-1],
             valid_loss[-1], valid_accuracy[-1]))
```

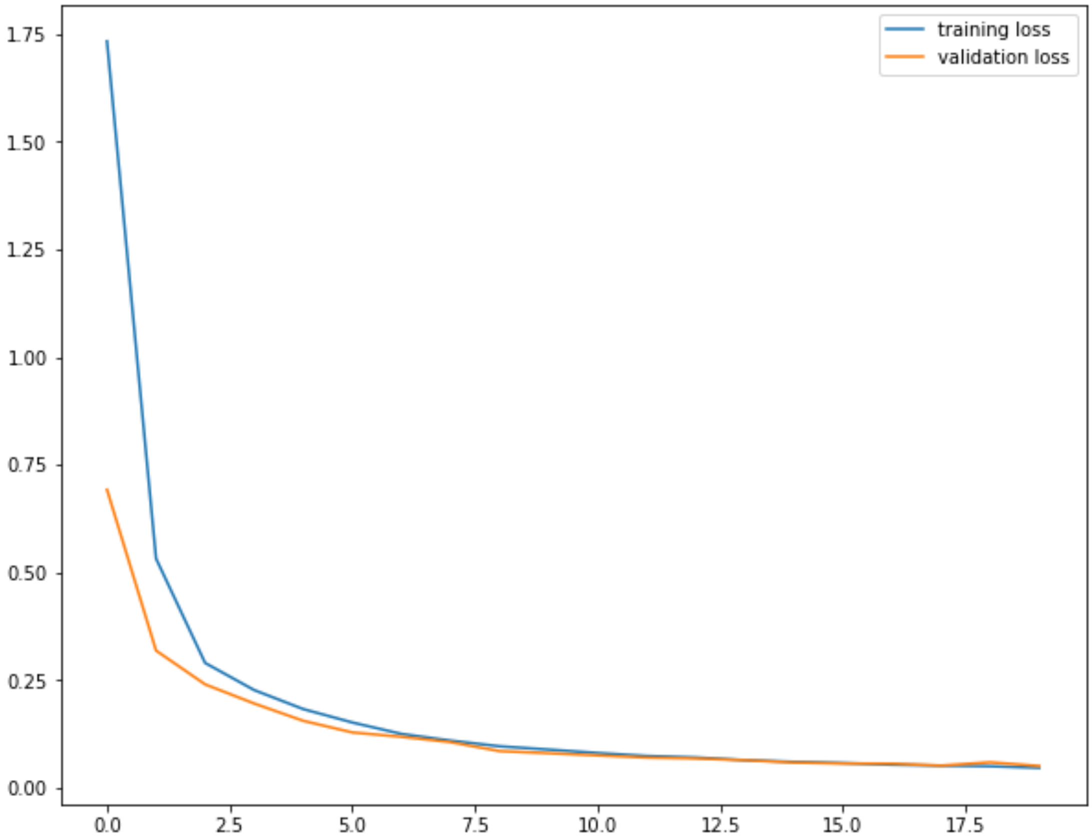
```
Epoch 1/20, Tr Loss: 1.7338, Tr Acc: 46.3733, Val Loss: 0.6924, V
al Acc: 79.6900
Epoch 2/20, Tr Loss: 0.5323, Tr Acc: 84.6383, Val Loss: 0.3186, V
al Acc: 90.6000
Epoch 3/20, Tr Loss: 0.2899, Tr Acc: 91.2700, Val Loss: 0.2402, V
al Acc: 92.9600
Epoch 4/20, Tr Loss: 0.2268, Tr Acc: 93.2067, Val Loss: 0.1957, V
al Acc: 94.3000
Epoch 5/20, Tr Loss: 0.1827, Tr Acc: 94.5150, Val Loss: 0.1555, V
al Acc: 95.3900
Epoch 6/20, Tr Loss: 0.1516, Tr Acc: 95.3767, Val Loss: 0.1285, V
al Acc: 96.0200
Epoch 7/20, Tr Loss: 0.1251, Tr Acc: 96.2300, Val Loss: 0.1184, V
al Acc: 96.4800
Epoch 8/20, Tr Loss: 0.1096, Tr Acc: 96.7733, Val Loss: 0.1060, V
al Acc: 96.5800
Epoch 9/20, Tr Loss: 0.0963, Tr Acc: 97.1517, Val Loss: 0.0852, V
al Acc: 97.4300
Epoch 10/20, Tr Loss: 0.0887, Tr Acc: 97.2633, Val Loss: 0.0803,
Val Acc: 97.5600
Epoch 11/20, Tr Loss: 0.0807, Tr Acc: 97.5483, Val Loss: 0.0752,
Val Acc: 97.6700
Epoch 12/20, Tr Loss: 0.0738, Tr Acc: 97.7783, Val Loss: 0.0701,
Val Acc: 97.9400
Epoch 13/20, Tr Loss: 0.0704, Tr Acc: 97.8100, Val Loss: 0.0681,
Val Acc: 97.8400
Epoch 14/20, Tr Loss: 0.0642, Tr Acc: 98.0517, Val Loss: 0.0640,
Val Acc: 98.0600
Epoch 15/20, Tr Loss: 0.0601, Tr Acc: 98.1833, Val Loss: 0.0587,
Val Acc: 98.1700
Epoch 16/20, Tr Loss: 0.0574, Tr Acc: 98.2917, Val Loss: 0.0564,
Val Acc: 98.1400
Epoch 17/20, Tr Loss: 0.0535, Tr Acc: 98.4033, Val Loss: 0.0560,
Val Acc: 98.2800
Epoch 18/20, Tr Loss: 0.0514, Tr Acc: 98.4300, Val Loss: 0.0516,
Val Acc: 98.3900
Epoch 19/20, Tr Loss: 0.0500, Tr Acc: 98.4817, Val Loss: 0.0591,
Val Acc: 98.0600
Epoch 20/20, Tr Loss: 0.0462, Tr Acc: 98.6117, Val Loss: 0.0507,
Val Acc: 98.3800
```

We can see that the model is still learning something. We might want to train another 10 epochs or so to see if validation accuracy increases further. For now, though, we'll just save the model.

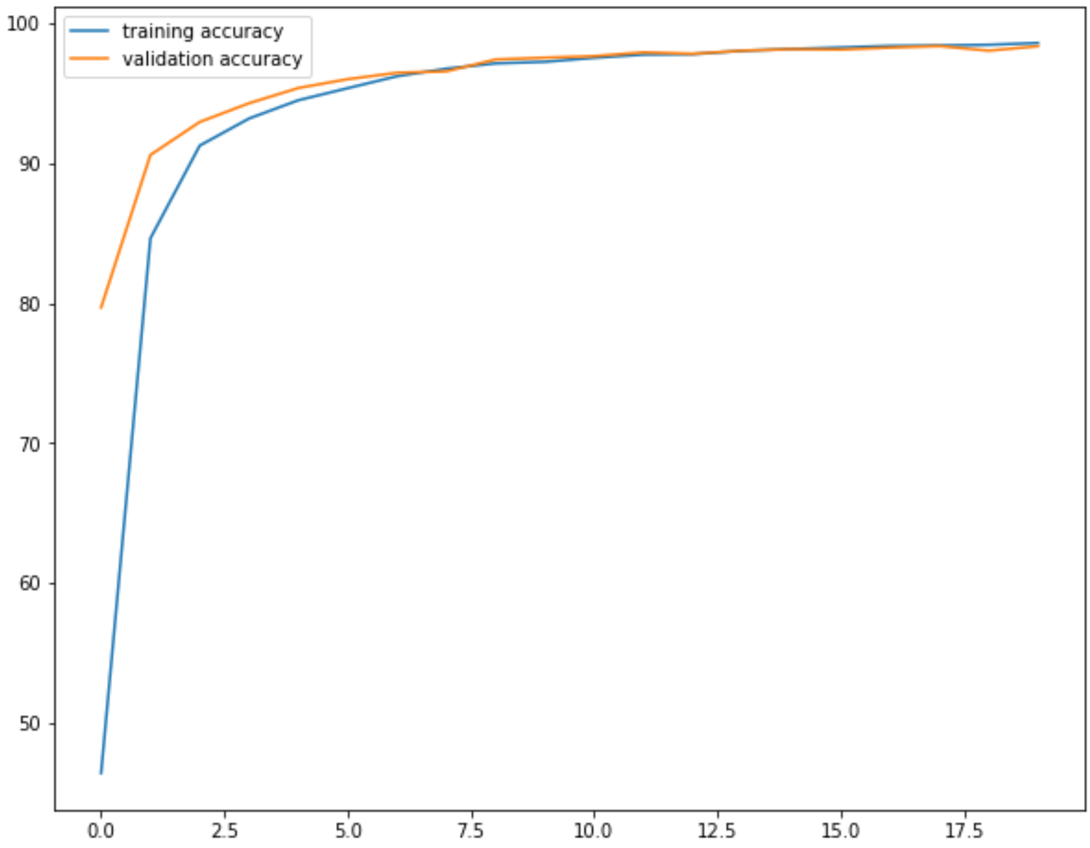
In [34]:
```
# save the model
torch.save(net.state_dict(), "./3.model.pth")
```

Next, let's visualize the loss and accuracy

In [35]:
```
# Plot loss curves

f = plt.figure(figsize=(10, 8))
plt.plot(train_loss, label='training loss')
plt.plot(valid_loss, label='validation loss')
plt.legend()
plt.show()
```



In [36]:
```
# Plot accuracy curves

f = plt.figure(figsize=(10, 8))
plt.plot(train_accuracy, label='training accuracy')
plt.plot(valid_accuracy, label='validation accuracy')
plt.legend()
plt.show()
```

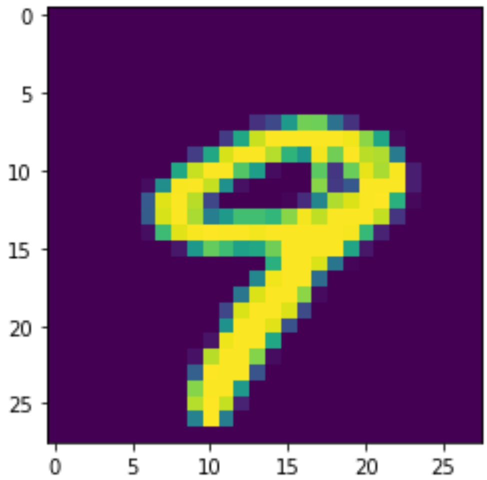


What can you conclude from the loss and accuracy curves?

1. We are not overfitting (at least not yet)
2. We should continue training, as validation loss is still improving
3. Validation accuracy is much higher than last week's fully connected models

Now let's test on a single image.

In [37]:
```
image_index = 9
img = mnist_valid[image_index][0].resize_((1, 1, 28, 28))
img = Variable(img)
label = mnist_valid[image_index][1]
plt.imshow(img[0,0])
net.eval()

if cuda.is_available():
    net = net.cuda()
    img = img.cuda()
else:
    net = net.cpu()
    img = img.cpu()

output = net(img)
```



In [38]:
```
output
```

Out[38]:
```
tensor([[ -8.7754,  -6.6059,  -3.1579,   2.1366,   5.0296,  -1.36
83, -10.7826,
          6.9277,   4.9248,  15.0195]], device='cuda:0',
       grad_fn=<AddmmBackward>)
```

In [39]:
```
_, predicted = torch.max(output.data, 1)
print("Predicted label:", predicted[0].item())
print("Actual label:", label)
```

```
Predicted label: 9
Actual label: 9
```

Take-home exercise (70 points)

Apply the tech you've learned up till now to take Kaggle's 2013 [Dogs vs. Cats Challenge (https://www.kaggle.com/c/dogs-vs-cats)](https://www.kaggle.com/c/dogs-vs-cats). Download the training and test datasets and try to build the best PyTorch CNN you can for this dataset. Describe your efforts and the results in a brief lab report.

In [ ]:
```
```