Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = "Nopphawan Nurnuansuwan"
ID = "122410"
```

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.

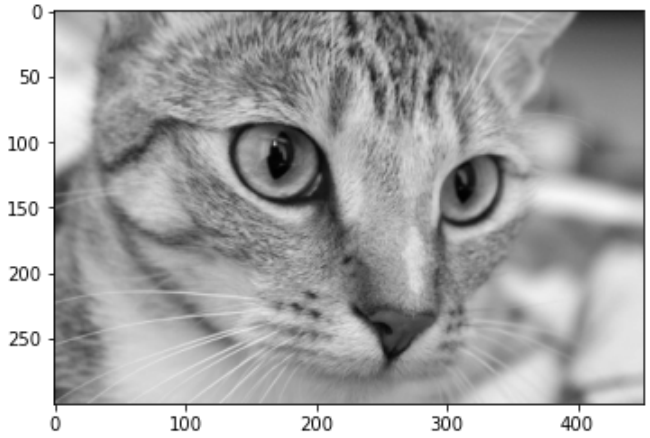
1. Reading input images.
2. Preparing filters.
3. Conv layer: Convolving each filter with the input image.
4. ReLU layer: Applying ReLU activation function on the feature maps (output of conv layer).
5. Max Pooling layer: Applying the pooling operation on the output of ReLU layer.
6. Stacking the conv, 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 [2]:
```
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)
print('Image dimensions (After transform to gray scale):', img.shape)
plt.imshow(img, cmap='gray')
plt.show()
```

```
Image dimensions: (300, 451, 3)
Image dimensions (After transform to gray scale): (300, 451)
```



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.

In this case, the convolution (cross correlation) computation can be written

$$\mathrm{X}'(i,j,k) = (\mathrm{X} \star \mathrm{K}_k)(i,j) = \sum_m \sum_n \sum_c \mathrm{X}(i+m, j+n, c)\mathrm{K}_k(m,n,c),$$

where X is an input tensor with three dimensions (H rows, W columns, and C channels), K_k is likewise an input tensor with three dimensions (M rows, N columns, and C channels), resulting in feature map $\mathrm{X}'(\cdot,\cdot,k)$, i.e., the kth channel of the overall output tensor. If our conv layer contains K kernels $\mathrm{K}_1, \ldots, \mathrm{K}_K$, after K of these convolution operations, the resulting tensor X' will contain K channels.

Let's make two $3 \times 3 \times 1$ filters, using the horizontal and vertical Sobel edge filters, flipped horizontally and vertically so that they give positive values for increasing intensity across an edge when used in cross correlation rather than convolution mode:

In [3]:
```
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

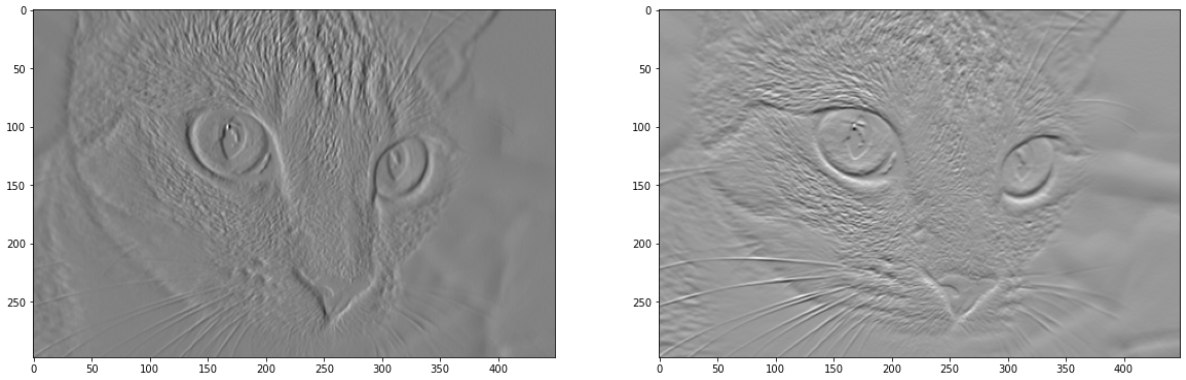
Let's convolve (correlate) an input image with our filters. The convolution will be stride 1 and valid region only.

In [4]:
```
# Convolution (cross correlation operation) with stride 1 for a single channel kernel,
# assuming the input image is also single channel.

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.shape) + (2*padding))/stride) + 1
    result = np.zeros((int(results_dim[0]), int(results_dim[1])))

    # r: row, c: column !!Actually, we should not use loop!!
    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 filter 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 image 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 (number 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 separately then add them. If the input
        # has just a single channel, we do the convolution directly.
        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 [5]:
```
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()
```

```
5 s ± 253 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Convolutional feature maps shape: (298, 449, 2)
```



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) 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.

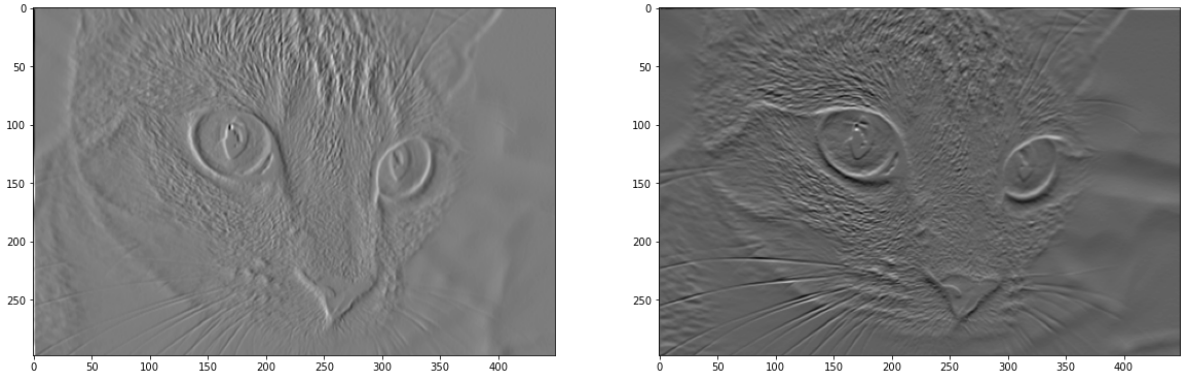
- Implement an optimized CNN convolution operation (10 points)
- Describe the experiment and the results (5 points)

Get a hint!

In [6]:
```
from numpy.fft import fft2, ifft2

def convolve2(img, conv_filter):
    stride = 1
    padding = 0
    w_img, b_img = img.shape[0], img.shape[1]
    filter_size = conv_filter.shape[1]
    output_size = (int(w_img - filter_size + (2*padding)/stride + 1),
                   int(b_img - filter_size + (2*padding)/stride + 1))

    fft_img = fft2(img, s = output_size)
    fft_filter = fft2(conv_filter, s = output_size)
    output = np.real(ifft2(np.multiply(fft_filter, fft_img)))

    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 filter 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 image 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 (number 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, :, :]
        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_num], curr_filter[:, :, ch_num])
        else:
            conv_map = convolve2(img, curr_filter)
        feature_maps[:, :, filter_num] = conv_map

    return feature_maps
```

In [7]:
```
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
```

```
139 ms ± 10.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Convolutional feature maps shape: (298, 449, 2)
```



```
success!
```

YOUR ANSWER HERE

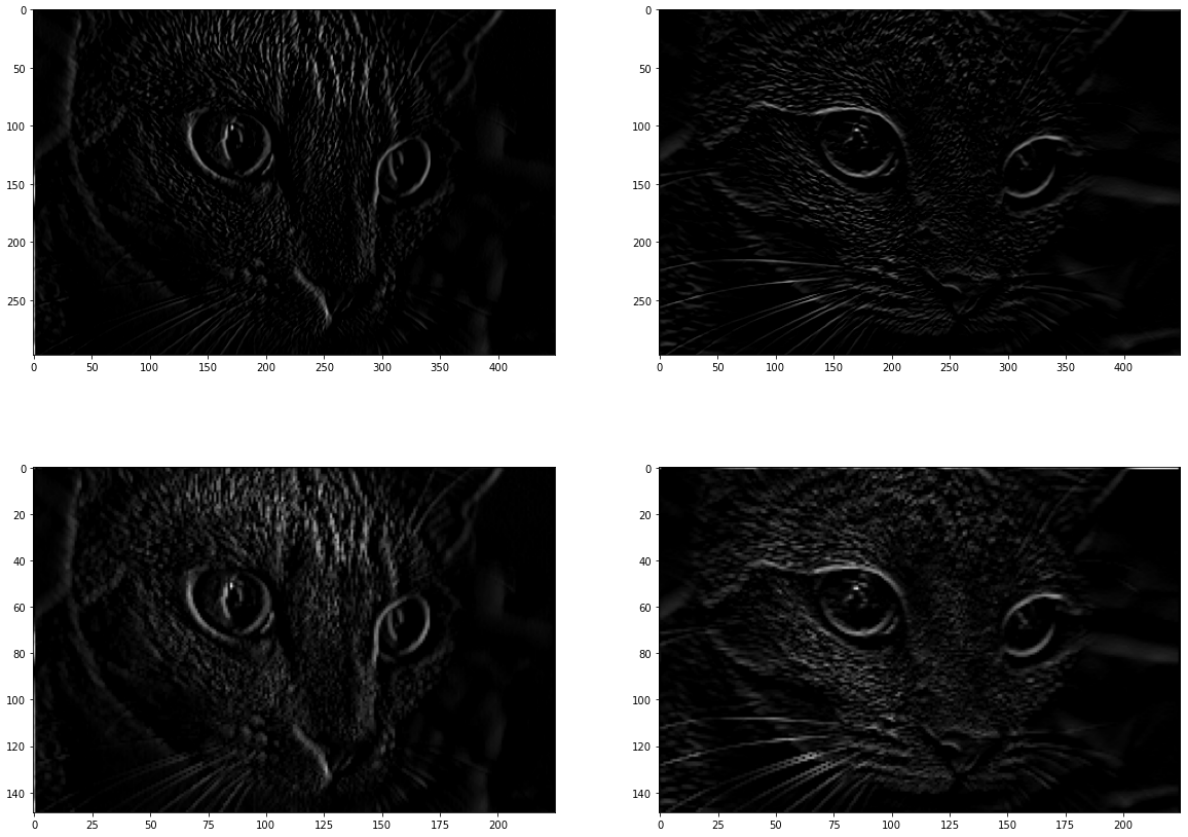
Pooling and relu

Next, we consider the feedforward pooling and ReLU operations.

In [8]:
```
# 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, stride):
            c2 = 0
            for c in np.arange(0, feature_map.shape[1]-size+1, stride):
                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 the ReLU and pooling operations on the result of the previous convolutions:

In [9]:
```
relued_features = relu(features)
pooled_features = pooling(relued_features)

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(20, 15))
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()
```



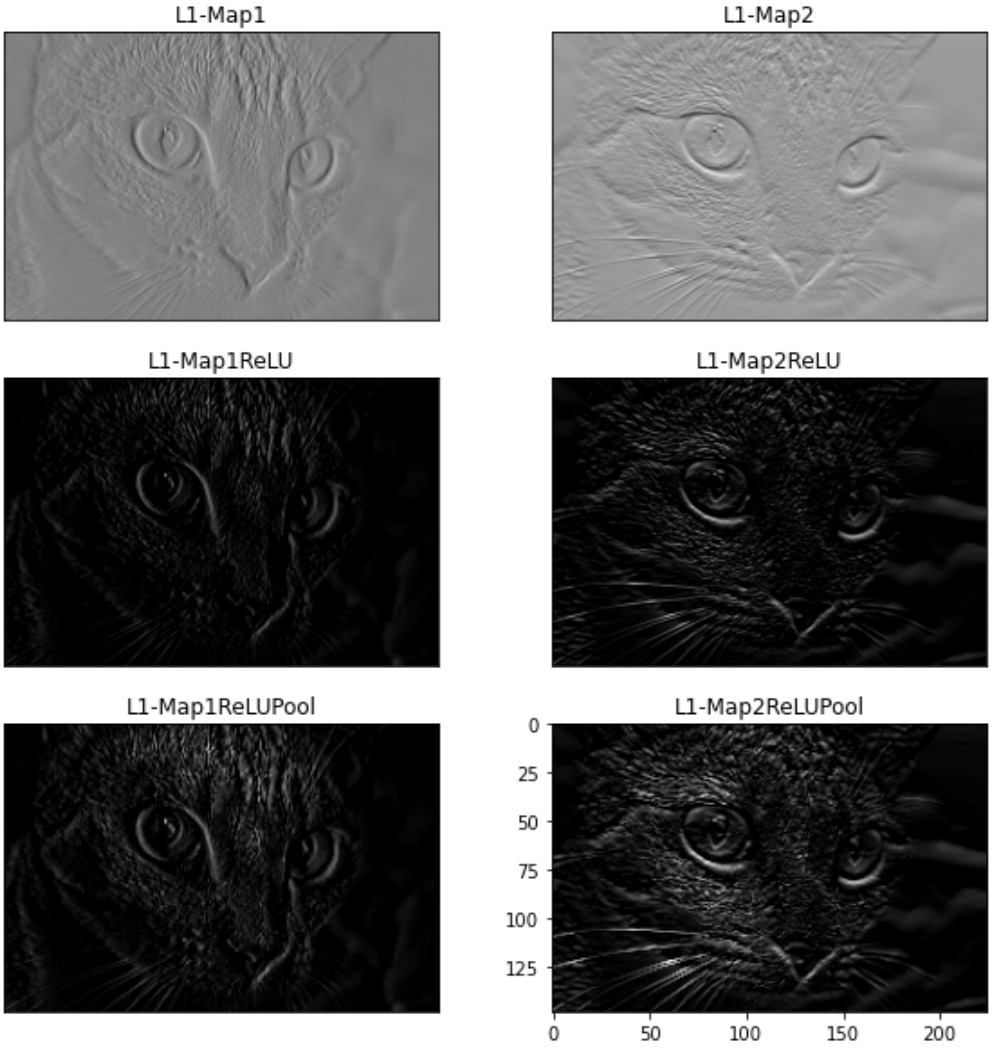
Next, let's visualize all of the feature maps in the model together.

In [10]:
```
# 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.shape[-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.shape[-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...
```

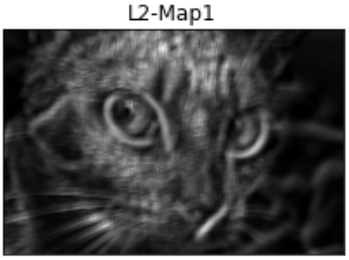
In [11]:
```
# Show results

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 [12]:
```
# 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("gray")
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("gray")
ax1[2, 0].get_xaxis().set_ticks([])
ax1[2, 0].get_yaxis().set_ticks([])
ax1[2, 1].set_title("L1-Map2ReLUPool")

plt.show()
```

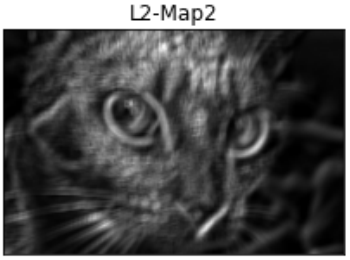

L1-Map1
L1-Map2
L1-Map1ReLU
L1-Map2ReLU
L1-Map1ReLUPool
L1-Map2ReLUPool
0
25
50
75
100
125
0
50
100
150
200

In [13]:
```
# 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("gray")
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("gray")
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("gray")
ax2[2, 2].get_xaxis().set_ticks([])
ax2[2, 2].get_yaxis().set_ticks([])
ax2[2, 2].set_title("L2-Map3ReLUPool")
plt.show()
```

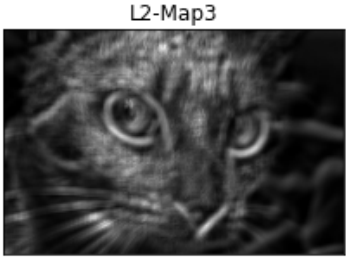
L2-Map1



L2-Map2



L2-Map3



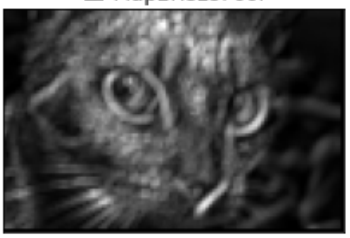
L2-Map1ReLU

L2-Map2ReLU

L2-Map3ReLU

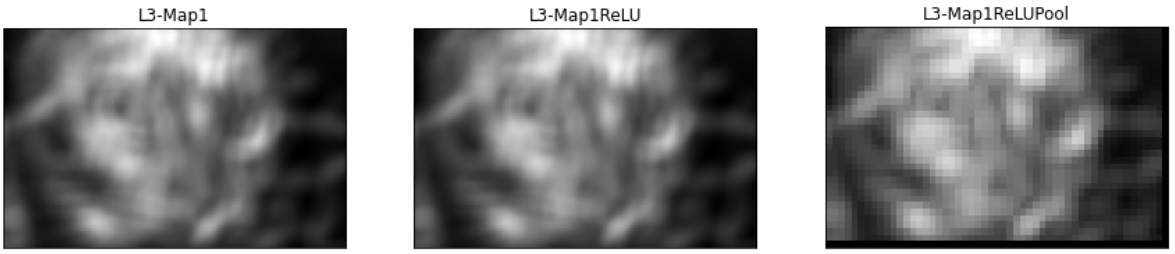
L2-Map1ReLUPool

L2-Map2ReLUPool



L2-Map3ReLUPool

In [14]:
```
# 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("gray")
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)

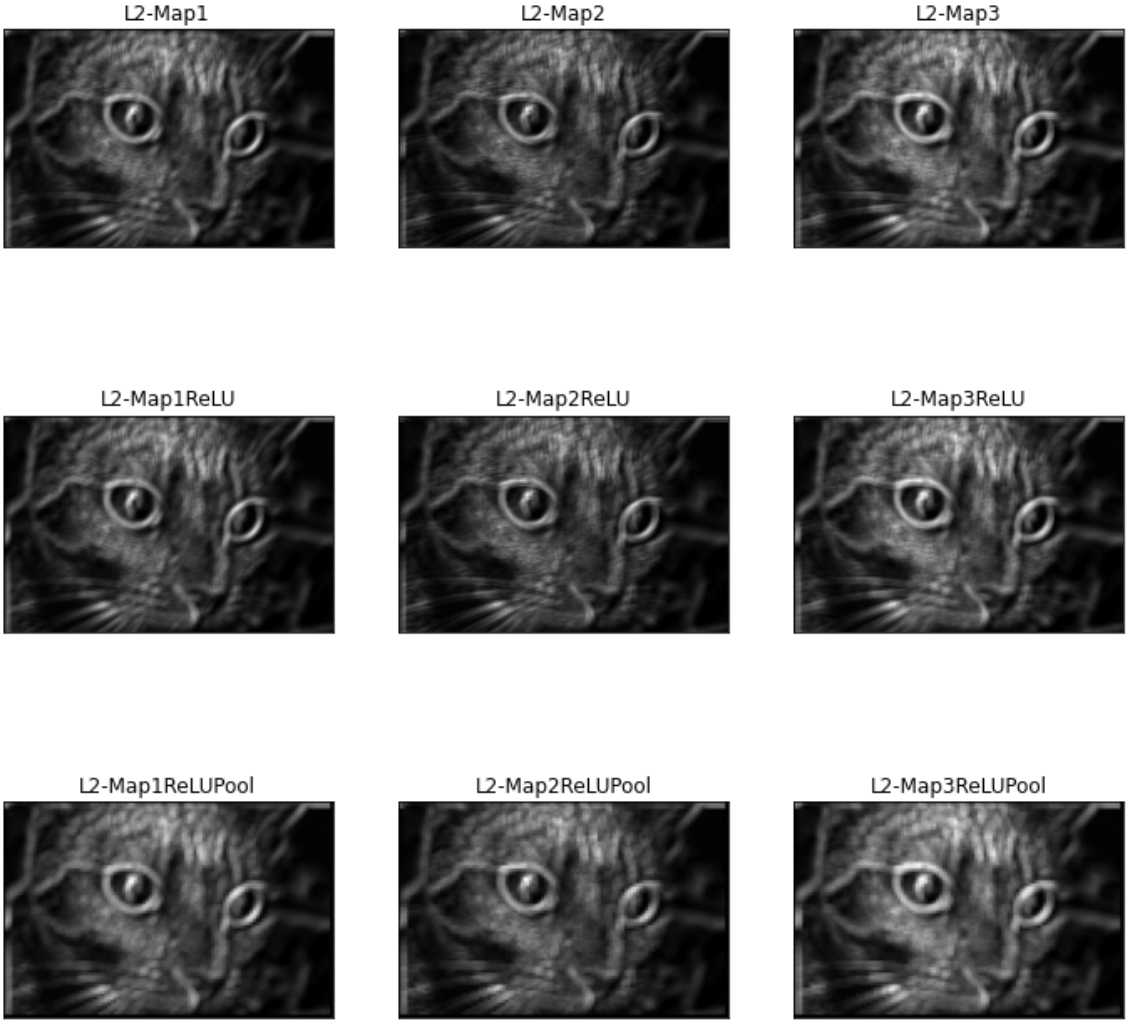
Modify the three-layer CNN above to use your conv2() function. Check the result, explain what you did, and take note of any differences in the result.

In [15]:
```
# First conv layer

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.shape[-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.shape[-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)
```

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

In [16]:
```
# 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("gray")
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("gray")
ax1[2, 0].get_xaxis().set_ticks([])
ax1[2, 0].get_yaxis().set_ticks([])
ax1[2, 1].set_title("L1-Map2ReLUPool")

plt.show()
```

In [17]:
```
# 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("gray")
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("gray")
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("gray")
ax2[2, 2].get_xaxis().set_ticks([])
ax2[2, 2].get_yaxis().set_ticks([])
ax2[2, 2].set_title("L2-Map3ReLUPool")
plt.show()
```



In [18]:
```
# 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("gray")
ax3[2].get_xaxis().set_ticks([])
ax3[2].get_yaxis().set_ticks([])
ax3[2].set_title("L3-Map1ReLUPool")
plt.show()
```

YOUR ANSWER HERE

CNNs in PyTorch

Now that you've seen layered convolutions in action, at least in feed-forward mode, using structured kernels and random kernels, you can probably imagine how we would backpropagate loss through the ReLU, pooling, and convolutional layers. Don't worry, we won't make you implement it. Let's use the compute graph and gradient calculation features of PyTorch to do the work for us.

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

As we learned last week, 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.

The TorchVision module is a separate module that builds on PyTorch and provides datasets and operations useful for computer vision problems::

1. datasets: TorchVision datasets are subclasses of `torch.utils.data.Dataset` . Some of the most commonly used torchvision datasets available are MNIST, CIFAR, and COCO. In this example we will see how to load the MNIST dataset using a custom subclass of the higher-level dataset class.
2. transforms: Transforms are used for image transformations. These transformations are useful for preparing images for input to our CNN and for so-called data augmentation, in which we perform randomized purturbations of input images prior to each training iteration, giving us more variety in the training data. The MNIST dataset from torchvision is in the PIL (Python Image Library) image format. To convert MNIST images to tensors, we will use `transforms.ToTensor()` .

In [19]:
```
import torch
import torch.cuda as cuda
import torch.nn as nn
import os

# Set proxy in case it's not already set in our environment before loading torchvision

os.environ['https_proxy'] = 'http://192.41.170.23:3128'
os.environ['http_proxy'] = 'http://192.41.170.23:3128'

import torchvision

from torch.autograd import Variable

from torchvision import datasets
from torchvision import transforms

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

Load the MNIST data

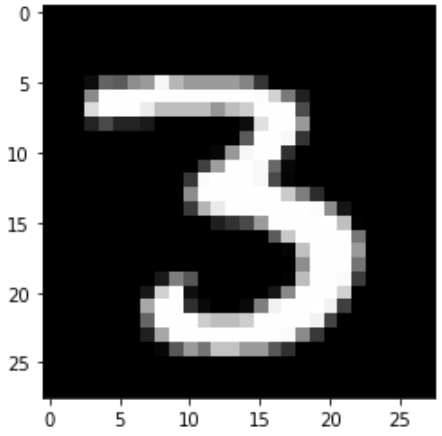
First, let's load the data and apply a transfromation of the input elements (pixel intensities) assuming a specific mean and standard deviation over the entire training dataset. Note that PIL automatically normalizes the incoming pixel data, which are integers between 0 and 255, to the floating point range $[0..1]$. For this dataset, we'll use a mean of 0 and a standard deviation of 1, which together imply *no* normalization. We could change `mean` and `stddev` to apply a normalizing transformation.

In [20]:
```
# Mean and standard deviation to use for normalization. For now, no transformation.

mean = 0.0
stddev = 1.0

# Transform to apply to input images

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

# Datasets

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

The elements of a torch Dataset can be accessed with array operations. The dataset has one array element for each input example. For supervised learning problems, each array element is usually a pair consisting of an input pattern and an output label (classification) or target value (regression). Let's grab one of the input images in the dataset. Note that a PyTorch tensor normally makes the feature map or image color channel the *first* dimension of the tensor, whereas it's usually the *last* dimension in an image.

In [21]:
```
img = mnist_train[12][0].numpy()
print('Input image 12 shape:', img.shape)
plt.imshow(img.reshape(28, 28), cmap='gray')
plt.show()
```

Input image 12 shape: (1, 28, 28)



Since we used a mean of 0 and standard deviation of 1 in our normalizing transform, the image intensities still vary from 0 to 1:

In [22]:
```
print('Min pixel intensity:', img.min(), 'max pixel intensity:', img.max())
```

Min pixel intensity: 0.0 max pixel intensity: 1.0

Let's see how to read one the labels, and also create a DataLoader for the dataset. A DataLoader goes through a Dataset , optionally shuffling the examples in the set, then loads examples in batches. The user of a loader applies an iterator to it to grab the batches one by one until the end of the dataset is reached.

In [23]:
```
label = mnist_train[12][1]
print('Label of image above:', label)

# Reduce batch size if you get out-of-memory errors!

batch_size = 1024
mnist_train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True, num_
mnist_valid_loader = torch.utils.data.DataLoader(mnist_valid, batch_size=batch_size, shuffle=True, num_
```

Label of image above: 3

Define the NN model

Now, let's use two convolutional layers followed by two fully connected layers. The input size of each image is (28,28,1). We will use a stride of 1 and padding of size 0 (valid convolution results only, which will shrink our input tensor).

For the first convolutional layer, let's apply 20 filters of size 5×5. The output tensor size formula is

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

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

In our case, we get $\frac{(28,28,1)-(5,5,1)+(2*0)}{1} + 1$ for each filter, so for 10 filters we get an output tensor 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 [29]:
```
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 10  (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)            # Dropout is a regularization technqiue we discusse
        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
```

Now let's create an instance of the neural network model, move it to the GPU, and set up our loss function and optimizer.

In [30]:
```
# The model
net = CNN_Model()

if cuda.is_available():
    net = net.cuda(2)


# Our loss function
criterion = nn.CrossEntropyLoss()

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

Now let's perform 20 epochs of training.

In [31]:
```
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(2)
            classes = classes.cuda(2)

        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 the 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.item() / 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(2)
```

```
            classes = classes.cuda(2)

        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.item() / 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]))
```

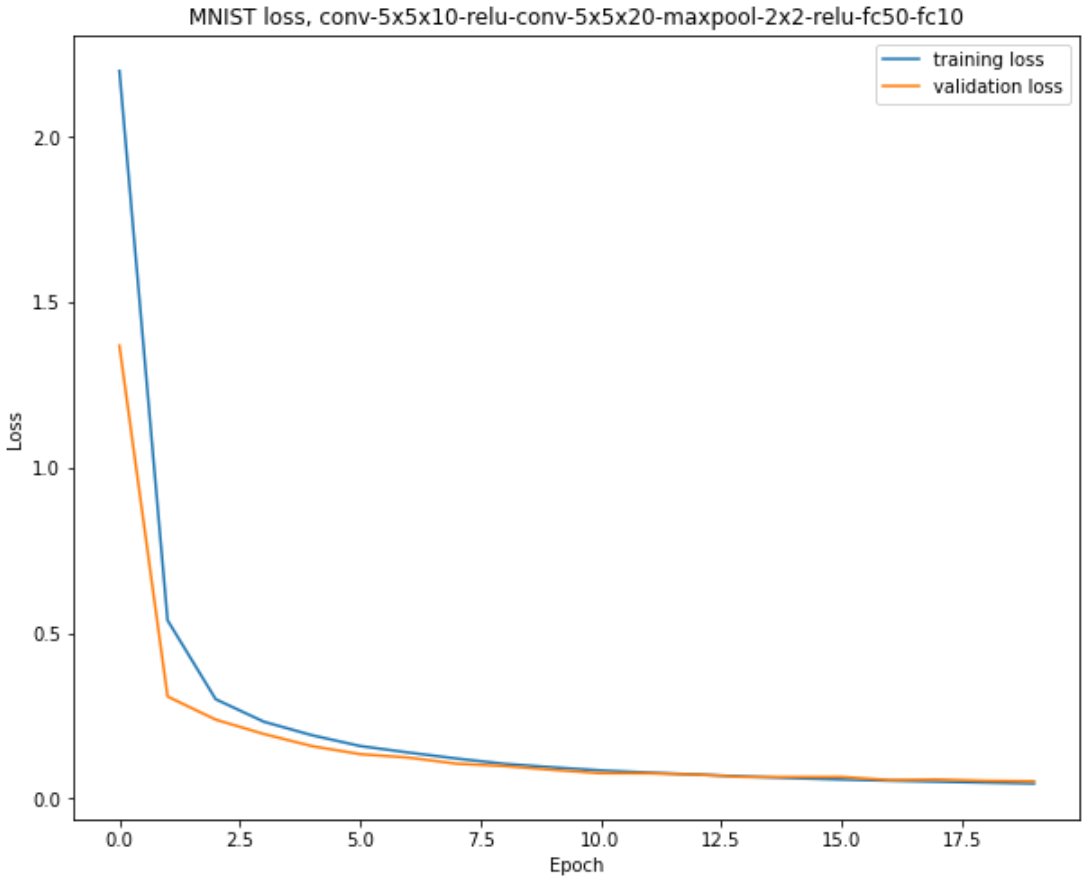
```
Epoch 1/20, Tr Loss: 2.1995, Tr Acc: 25.5067, Val Loss: 1.3689, Val Acc: 68.5000
Epoch 2/20, Tr Loss: 0.5387, Tr Acc: 84.0200, Val Loss: 0.3082, Val Acc: 90.7700
Epoch 3/20, Tr Loss: 0.3000, Tr Acc: 91.0033, Val Loss: 0.2383, Val Acc: 92.8300
Epoch 4/20, Tr Loss: 0.2320, Tr Acc: 93.1783, Val Loss: 0.1948, Val Acc: 94.2300
Epoch 5/20, Tr Loss: 0.1911, Tr Acc: 94.3750, Val Loss: 0.1582, Val Acc: 95.4400
Epoch 6/20, Tr Loss: 0.1584, Tr Acc: 95.3783, Val Loss: 0.1338, Val Acc: 96.0900
Epoch 7/20, Tr Loss: 0.1388, Tr Acc: 95.9150, Val Loss: 0.1236, Val Acc: 96.5200
Epoch 8/20, Tr Loss: 0.1207, Tr Acc: 96.4583, Val Loss: 0.1051, Val Acc: 96.8300
Epoch 9/20, Tr Loss: 0.1045, Tr Acc: 96.9333, Val Loss: 0.0980, Val Acc: 96.9800
Epoch 10/20, Tr Loss: 0.0941, Tr Acc: 97.1767, Val Loss: 0.0862, Val Acc: 97.3200
Epoch 11/20, Tr Loss: 0.0847, Tr Acc: 97.4100, Val Loss: 0.0761, Val Acc: 97.6500
Epoch 12/20, Tr Loss: 0.0780, Tr Acc: 97.6317, Val Loss: 0.0758, Val Acc: 97.6800
Epoch 13/20, Tr Loss: 0.0723, Tr Acc: 97.8133, Val Loss: 0.0720, Val Acc: 97.8500
Epoch 14/20, Tr Loss: 0.0661, Tr Acc: 98.0033, Val Loss: 0.0649, Val Acc: 98.0000
Epoch 15/20, Tr Loss: 0.0613, Tr Acc: 98.1100, Val Loss: 0.0650, Val Acc: 97.9800
Epoch 16/20, Tr Loss: 0.0571, Tr Acc: 98.2400, Val Loss: 0.0652, Val Acc: 98.0000
Epoch 17/20, Tr Loss: 0.0536, Tr Acc: 98.3617, Val Loss: 0.0557, Val Acc: 98.2400
Epoch 18/20, Tr Loss: 0.0507, Tr Acc: 98.4533, Val Loss: 0.0573, Val Acc: 98.2500
Epoch 19/20, Tr Loss: 0.0478, Tr Acc: 98.5733, Val Loss: 0.0533, Val Acc: 98.3100
Epoch 20/20, Tr Loss: 0.0451, Tr Acc: 98.6100, Val Loss: 0.0521, Val Acc: 98.3900
```

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.

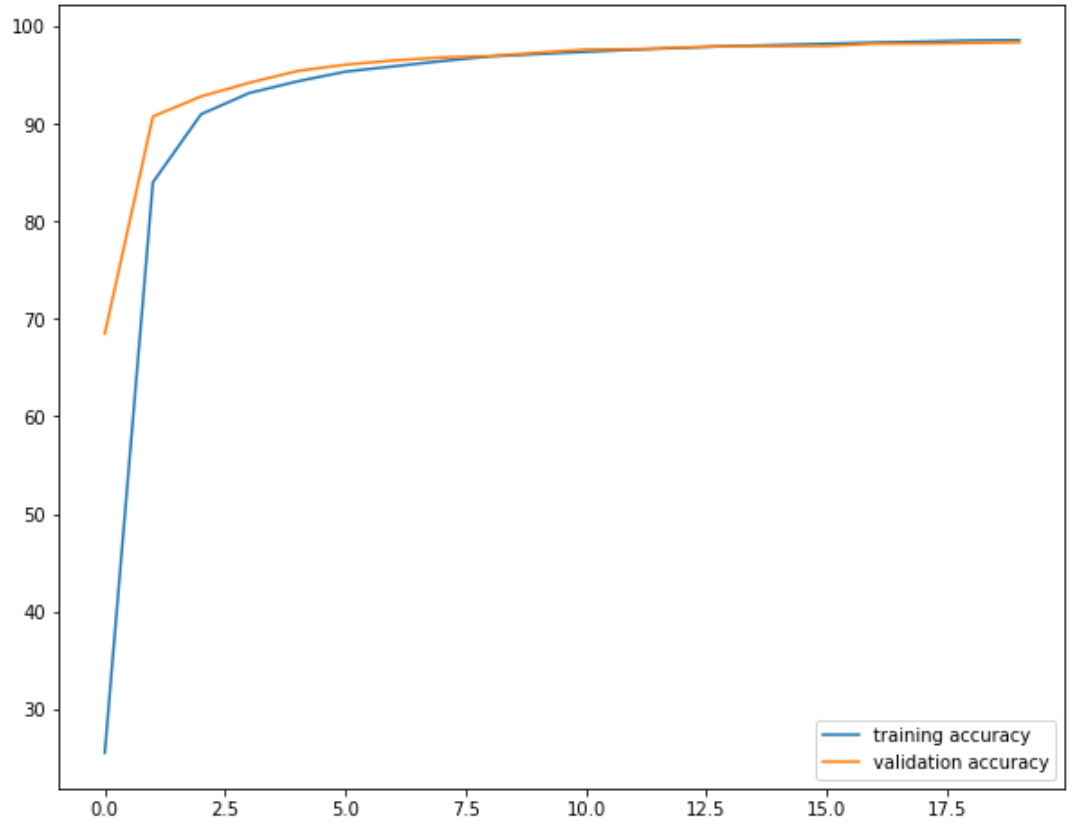
In [32]:
```
# save the model
torch.save(net.state_dict(), "./3.model.pth")
```

Next, let's visualize the loss and accuracy.

In [33]:
```
# Plot loss curves

f = plt.figure(figsize=(10, 8))
plt.plot(train_loss, label='training loss')
plt.plot(valid_loss, label='validation loss')
plt.title('MNIST loss, conv-5x5x10-relu-conv-5x5x20-maxpool-2x2-relu-fc50-fc10')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [34]:
```
# 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()
```

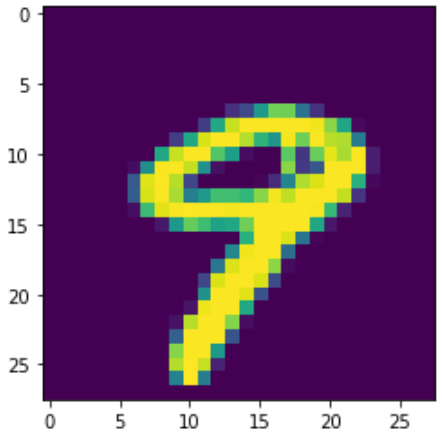


What can you conclude from the loss and accuracy curves?

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 [36]:
```
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(2)
    img = img.cuda(2)
else:
    net = net.cpu()
    img = img.cpu()

output = net(img)
```



In [37]:
```
output
```

Out[37]:
```
tensor([[ -5.9423, -10.0314,  -3.9764,   2.8745,   5.7845,  -0.8519,  -7.4996,
           6.8934,   8.0194,  15.4860]], device='cuda:2',
       grad_fn=<AddmmBackward0>)
```

In [38]:
```
_, 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). 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 [39]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
import sys

import torch
import torch.cuda as cuda
import torch.nn as nn

os.environ['https_proxy'] = 'http://192.41.170.23:3128'
os.environ['http_proxy'] = 'http://192.41.170.23:3128'

from torch.autograd import Variable
from torchvision import transforms
import torch.nn.functional as F

from time import time
```

In [40]:
```
resize = 64
transform = transforms.Compose([transforms.Resize((resize,resize)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,))])

train_path = '/home/Datasets/cats-and-dogs/train/'
dataset = []
img_label = []
start = time()
for path, _, filenames in os.walk(train_path):
    for i,filename in enumerate(filenames):
        dataset.append([])
        with Image.open(train_path+filename) as im:
            dataset[i].append(transform(im))

        if filename.split('.')[0] == 'cat':
            dataset[i].append(0)
        else:
            dataset[i].append(1)

        sys.stdout.write('\r Loading {:.2f} %'.format(i/len(filenames)*100))

load_time = time()-start
sys.stdout.write('\r Done! | Time: {:d} min {:.2f} sec'.format(int(load_time/60), load_time%60))
```

```
 Done! | Time: 4 min 49.50 sec
```

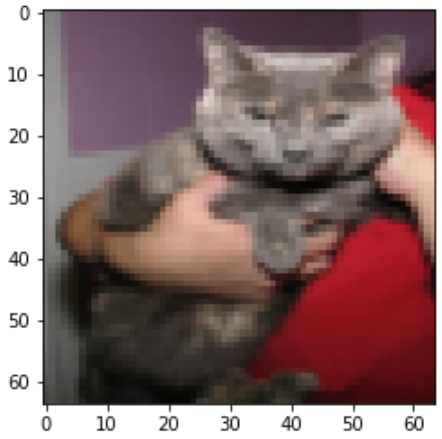
Out[40]: 31

In [41]:
```
import random
random.shuffle(dataset)

# Split train set and validation set
train_size = 0.8
m_train = int(train_size*len(dataset))
dataset_train = dataset[:m_train]
dataset_val = dataset[m_train:]
print("train samples:", len(dataset_train))
print("validation samples:", len(dataset_val))
```

```
train samples: 20000
validation samples: 5000
```

In [42]:
```
# Show a resized image
img = dataset[0][0].numpy()/2+0.5
img = np.rollaxis(img, 0, 3)
img = np.rollaxis(img, 1, 2)
if dataset[0][1] == 0:
    print('Class: cat')
else:
    print('Class: dog')
plt.imshow(img)
plt.show()
```

Class: cat



In [43]:
```
import torch
batch_size = 2000
train_loader = torch.utils.data.DataLoader(dataset_train, batch_size=batch_size, shuffle=False, num_work
val_loader = torch.utils.data.DataLoader(dataset_val, batch_size=batch_size, shuffle=False, num_workers
```

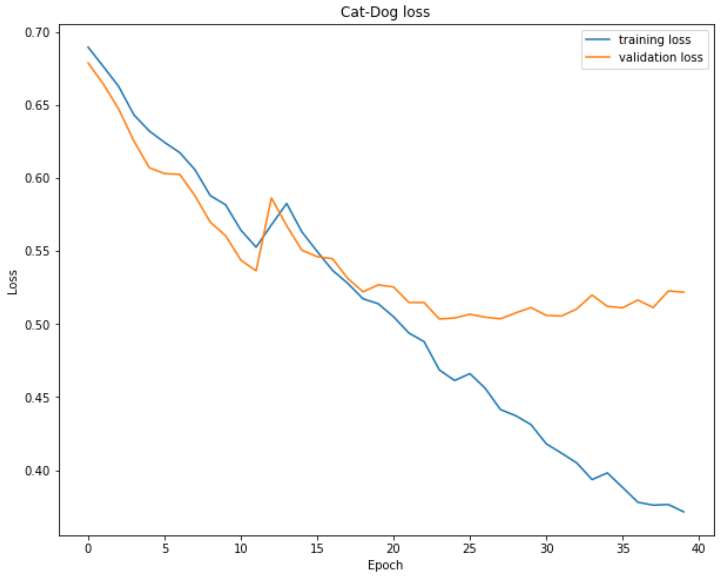
In [44]:
```
input_size = resize
conv_chanel = [3,10,20]
#conv_chanel = [3,10,20]
k = 7 # kernel size
p = 0 # padding
s = 1 # stride
mp = 2 # max pool

def conv_size_cal(in_size, k=5, p=0, s=1):
    return int((in_size+2*p-k)/s+1)

af_conv1 = conv_size_cal(input_size, k, p, s)
af_conv2 = conv_size_cal(af_conv1, k, p, s)
af_mp = int(af_conv2/mp)

print(af_conv1, af_conv2, af_mp)
```

58 52 26

In [45]:
```
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(conv_chanel[0], conv_chanel[1], kernel_size=k, padding=p, stride=s)      #
        self.relu1 = nn.ReLU()                          # Same as above

        # Convolution Layer 2
        self.conv2 = nn.Conv2d(conv_chanel[1], conv_chanel[2], kernel_size=k, padding=p, stride=s)     # 2
        self.conv2_drop = nn.Dropout2d(p=0.5)           # Dropout is a regularization technqiue we discussed
        if mp > 1:
            self.maxpool2 = nn.MaxPool2d(mp)                # 10 x 10 x 20  (after pooling)
        self.relu2 = nn.ReLU()                          # Same as above

        # Fully connected layers
        self.fc1 = nn.Linear(af_mp*af_mp*conv_chanel[2], 50)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(50, 2)

    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)
        if mp > 1:
            x = self.maxpool2(x)
        x = self.relu2(x)

        # Switch from activation maps to vectors
        x = x.view(-1, af_mp*af_mp*conv_chanel[2])

        # Fully connected layer 1
        x = self.fc1(x)
        x = self.relu3(x)
        #x = F.relu(x)
        #x = F.dropout(x, training=True)

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

        return x
```

In [46]:
```
core = 2
# The model
net = CNN_Model()

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

# Our loss function
criterion = nn.CrossEntropyLoss()

# Our optimizer
learning_rate = 0.1
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate, momentum=0.9)
```

This is train/validation losses related with number of epochs.
The validation losses are not improved about 20th epoch.
Therefore, I designed to use num_epoch = 20

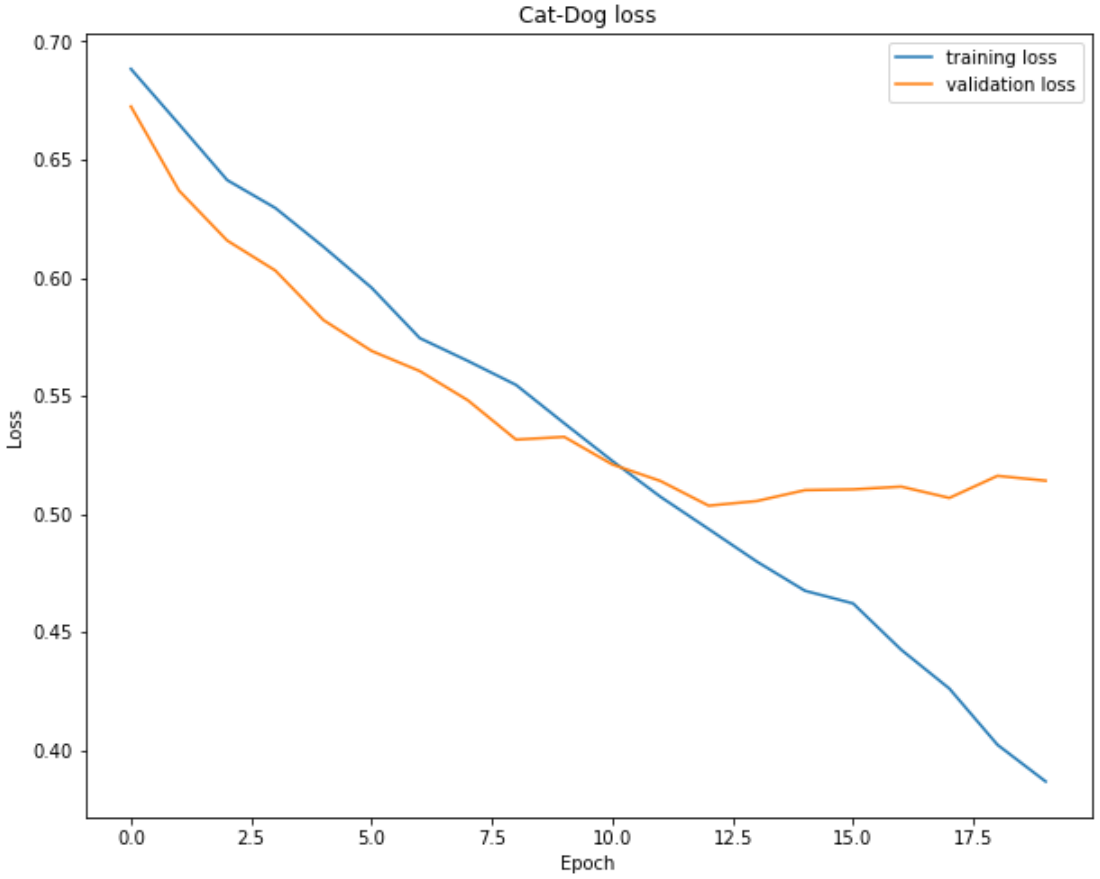

Cat-Dog loss
training loss
validation loss
Loss
Epoch
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
5
10
15
20
25
30
35
40

In [47]:
```
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(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(core)
            classes = classes.cuda(core)

        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 the 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.item() / float(len(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(val_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(core)
```

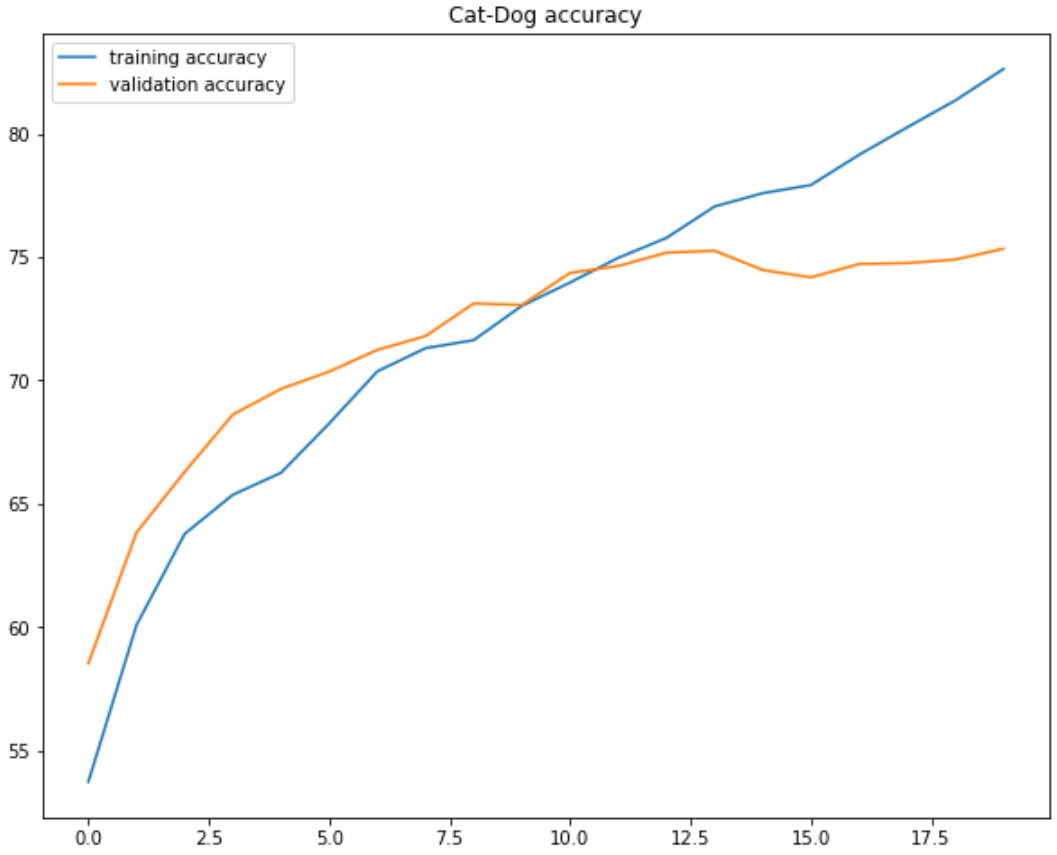
```
            classes = classes.cuda(core)

        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.item() / len(val_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: 0.6884, Tr Acc: 53.7400, Val Loss: 0.6725, Val Acc: 58.5400
Epoch 2/20, Tr Loss: 0.6650, Tr Acc: 60.0900, Val Loss: 0.6369, Val Acc: 63.8400
Epoch 3/20, Tr Loss: 0.6413, Tr Acc: 63.7850, Val Loss: 0.6158, Val Acc: 66.3000
Epoch 4/20, Tr Loss: 0.6295, Tr Acc: 65.3700, Val Loss: 0.6031, Val Acc: 68.6200
Epoch 5/20, Tr Loss: 0.6131, Tr Acc: 66.2600, Val Loss: 0.5821, Val Acc: 69.6600
Epoch 6/20, Tr Loss: 0.5958, Tr Acc: 68.2600, Val Loss: 0.5690, Val Acc: 70.3600
Epoch 7/20, Tr Loss: 0.5745, Tr Acc: 70.3700, Val Loss: 0.5605, Val Acc: 71.2400
Epoch 8/20, Tr Loss: 0.5648, Tr Acc: 71.3100, Val Loss: 0.5481, Val Acc: 71.8000
Epoch 9/20, Tr Loss: 0.5546, Tr Acc: 71.6350, Val Loss: 0.5315, Val Acc: 73.1200
Epoch 10/20, Tr Loss: 0.5384, Tr Acc: 73.0250, Val Loss: 0.5327, Val Acc: 73.0600
Epoch 11/20, Tr Loss: 0.5225, Tr Acc: 73.9700, Val Loss: 0.5210, Val Acc: 74.3600
Epoch 12/20, Tr Loss: 0.5073, Tr Acc: 74.9700, Val Loss: 0.5140, Val Acc: 74.6400
Epoch 13/20, Tr Loss: 0.4936, Tr Acc: 75.7750, Val Loss: 0.5035, Val Acc: 75.1800
Epoch 14/20, Tr Loss: 0.4799, Tr Acc: 77.0500, Val Loss: 0.5055, Val Acc: 75.2600
Epoch 15/20, Tr Loss: 0.4675, Tr Acc: 77.5900, Val Loss: 0.5101, Val Acc: 74.4800
Epoch 16/20, Tr Loss: 0.4621, Tr Acc: 77.9200, Val Loss: 0.5104, Val Acc: 74.1800
Epoch 17/20, Tr Loss: 0.4425, Tr Acc: 79.1450, Val Loss: 0.5116, Val Acc: 74.7200
Epoch 18/20, Tr Loss: 0.4261, Tr Acc: 80.2600, Val Loss: 0.5068, Val Acc: 74.7600
Epoch 19/20, Tr Loss: 0.4023, Tr Acc: 81.3500, Val Loss: 0.5162, Val Acc: 74.9000
Epoch 20/20, Tr Loss: 0.3868, Tr Acc: 82.6250, Val Loss: 0.5141, Val Acc: 75.3400
```

In [48]:
```
# Plot loss curves
f = plt.figure(figsize=(10, 8))
plt.plot(train_loss, label='training loss')
plt.plot(valid_loss, label='validation loss')
plt.title('Cat-Dog loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



In [49]:
```
# Plot accuracy curves
f = plt.figure(figsize=(10, 8))
plt.plot(train_accuracy, label='training accuracy')
plt.plot(valid_accuracy, label='validation accuracy')
plt.title('Cat-Dog accuracy')
plt.legend()
plt.show()
```



In [50]:
```
test_path = '/home/Datasets/cats-and-dogs/test1/'
test_dataset = []
test_real_img = []
start = time()

for path, _, filenames in os.walk(test_path):
    for i, filename in enumerate(filenames):
        with Image.open(test_path+filename) as im:
            test_real_img.append(im)
            test_dataset.append(transform(im))
        sys.stdout.write('\r Loading {:.2f} %'.format(i/len(filenames)*100))

load_time = time()-start
sys.stdout.write('\r Done! | Time: {:d} min {:.2f} sec'.format(int(load_time/60), load_time%60))
```

```
 Done! | Time: 3 min 44.14 sec
```

Out[50]: 31

In [51]:
```
core = 1
if cuda.is_available():
    net = net.cuda(core)
else:
    net = net.cpu()
net.eval()

pred_list = []
for i in range(len(test_dataset)):
    img = test_dataset[i]
    img = img[np.newaxis, :]
    if cuda.is_available():
        img = img.cuda(core)
    else:
        img = img.cpu()
    output = net(img)
    _, predicted = torch.max(output.data, 1)
    pred_list.append(predicted[0].item())
```

In [52]:
```
# show some test images

show_img = 20
st = random.randint(0, len(test_dataset)-show_img)

col = 5
row = int(np.ceil(show_img/col))
fig, axs = plt.subplots(row, col, figsize=(15/5*col, 13/4*row))
if row==1: axs = axs.reshape(1, -1)
for i in range(st, st+show_img):
    if pred_list[i]==0:
        clf_pred = "Cat"
    else:
        clf_pred = "Dog"
    r = int((i-st)/col)
    c = (i-st)%col
    axs[r,c].imshow(test_real_img[i].resize((600,600)))
    axs[r,c].set_title(clf_pred)
    axs[r,c].axis('off')
plt.show()
```

Explanation

- First, I have imported all training images via PIL.image library and transformed using torchvision.transforms .
- In transforms compose, there are resizing to 64x64, transforming the data to tensor, and normalizing the data.
- Second, I have shuffled and splited data to train and validation dataset.
- Next, I have pushed the datasets to torch.utils.data.DataLoader with batch size = 2000 samples.
- Third, I have made a CNN model class with following functions, respectively:
  - 1st 2D Convolutional function : input channel = 3 (RGB image), and output channel = 10
  - 1st ReLU function
  - 2nd 2D Convolutional function : input channel = 10 , and output channel = 20
  - Dropout some nodes with probability = 0.5
  - Max pool function with kernel size = 2
  - 2nd ReLU function
  - 1st Linear function : input features = width x height x output channel from 2nd Convolutional function, output features = 50
  - 2st Linear function : input features = output features from 1st Linear function, output features = number of classes (cat and dog) = 2
- I have calculated size of image in each step in CNN model to calculate size of image before go though 1st Linear function
- Next, I have set net = class CNN_model, the criterion is CrossEntropyLoss , and the optimizer is SGD with using learning_rate = 0.1 and momentum = 0.9
- I have tried to train the dateset with 40 epochs and found that the loss of validation set cannot improve/decrease after about epoch 20th. Therefore, I have trained the model using epochs = 20.
- The best accuracy validation set is about 75%
- Lastly, I have imported test images and predicted using the trained model.
- The result is quite good. As this result here: there are 3 miss-predicted images from 20 random samples, the accuracy of this random samples is 85%.