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

```
NAME = "Ayush Koirala"
ID = "st122802"
```

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.

CNN Explainer

Before doing this tutorial, take a look at [the CNN Explainer](). It gives beautiful illustrations of what's happening in a CNN at every level.

Connect to google drive for colab users

For the students who use google colab for run the system. It is better to upload your dataset or weight, model into your drive. The colab system will be reset everytime when the system has log off.

Ok, let's mount the drive

```
# for colab only
# from google.colab import drive
# drive.mount('/content/gdrive')

# Your root path in gdrive
root_path = 'gdrive/My Drive/'
```

Let's create folder

```
# for create the folder
import os

lab_path =  root_path + 'lab09/'
if not os.path.exists(lab_path):
  print("No folder ", lab_path, 'exist. Create the folder')
  os.mkdir(lab_path)
  print("Create directory finished")
else:
  print(lab_path, 'existed, do nothing')
```

For the person who want to use google drive as your memory you can check PyDrive. In this class, we do not talk about it.

Hand-coded CNN

This example is based on Ahmed Gad's tutorial.

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.

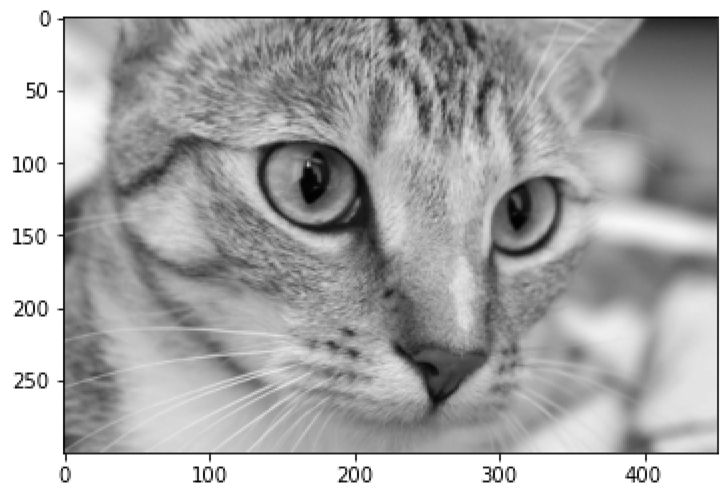
Reading an input image

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`.

```
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:

```
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.

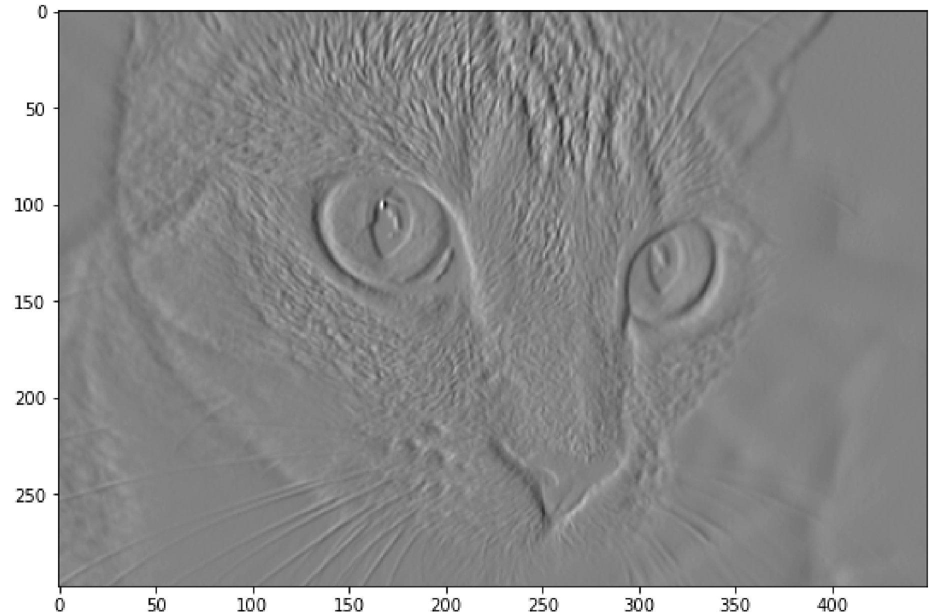
```
# Perform stride 1 cross correlation of an image and a filter. We output the valid region only
# (no padding).
def convolve(img, conv_filter):
    stride = 1
    padding = 0
    filter_size = conv_filter.shape[1]
    results_dim = ((np.array(img.shape) - np.array(conv_filter.shape) + (2*padding))/stride)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

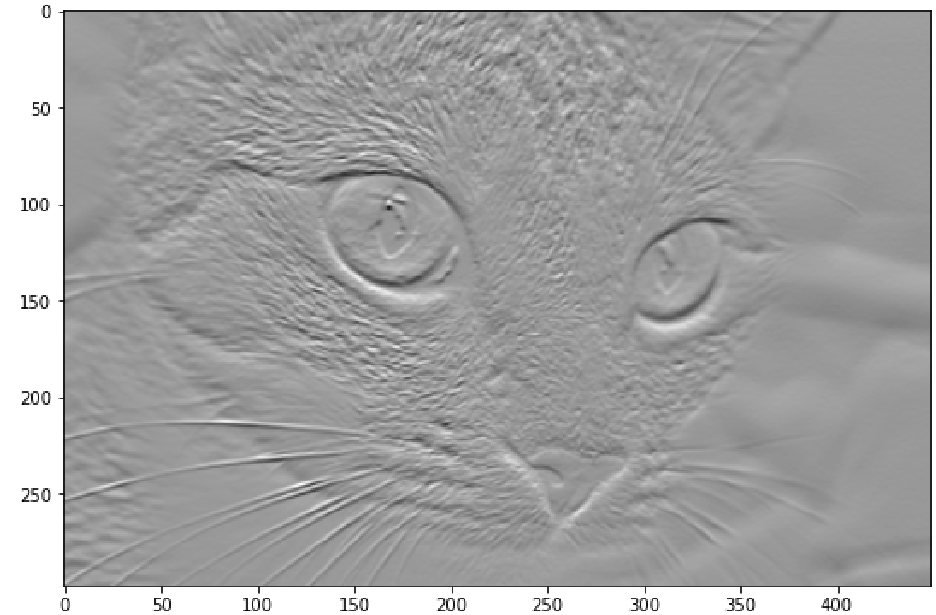
```
    # Ensure filter dimensions are odd
    if conv_filters.shape[1]%2==0:
        raise Exception('Error: Filter must have an odd size (number of rows and columns must be odd).')

    # 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:

```
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()
```

```
3.8 s ± 89.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
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]() 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:**

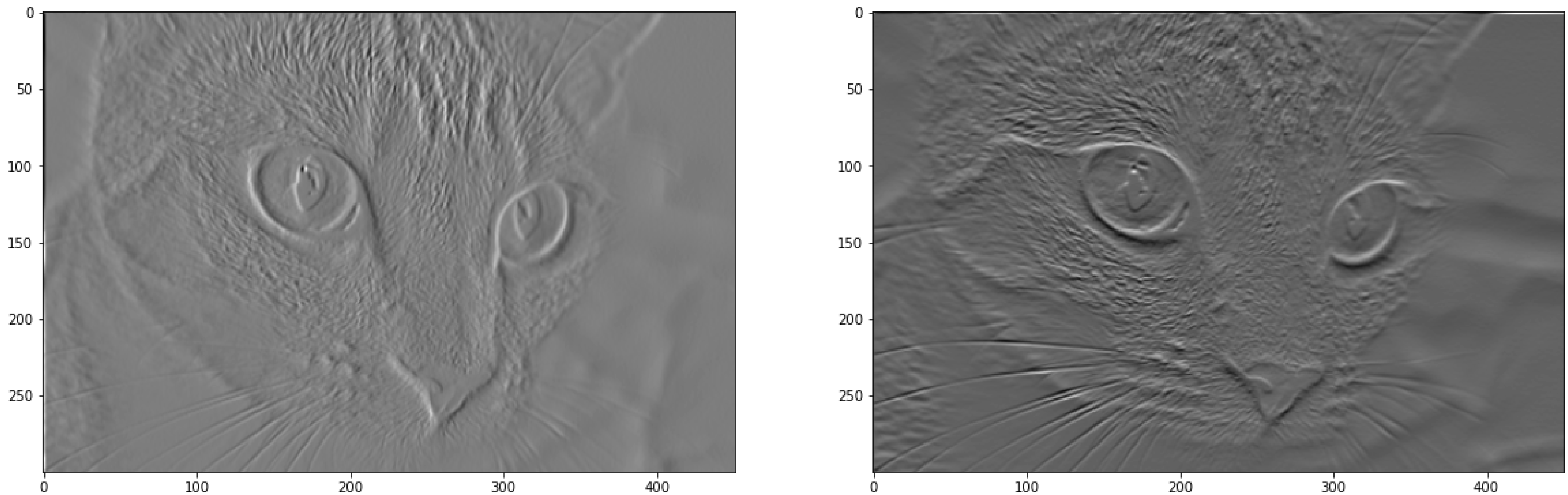
```
from numpy.fft import fft2, ifft2
def convolve2(img, conv_filter):
    output = None
    # YOUR CODE HERE

    ##if conv_filter is of shape (num_filters, k, k)
    if len(img.shape)<3:
        A = img.reshape(1, img.shape[0], img.shape[1])
    else:
        A = img

    if len(conv_filter.shape)<3:
        B = conv_filter.reshape(1, conv_filter.shape[0], conv_filter.shape[1])
    else:
        B = conv_filter

    f_B = np.zeros((B.shape[0], A.shape[-2], A.shape[-1]), dtype=np.complex128)
    for i_M in np.arange(B.shape[0]):
        f_B[i_M, :, :] = fft2(B[i_M, :, :], s=A.shape[-2:])

    C = np.zeros((A.shape[0], B.shape[0], A.shape[1], A.shape[2]))

    for i_N in np.arange(A.shape[0]):
            f_A = fft2(A[i_N, :, :])
            for i_M in np.arange(B.shape[0]):
                C[i_N, i_M, :, :] = np.real(ifft2(f_A*f_B[i_M, :, :]))

    output = C.squeeze()
#     print('output shape',output.shape)

    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],
                             img.shape[1],
                             conv_filters.shape[0]))

    # Perform convolutions
    # YOUR CODE HERE
    feature = convolve2(img, conv_filters) #shape (num_filters, img.shape[0], img.shape[1])

    #reshaping to (img.shape[0], img.shape[1], num_filters)
    for i in range(conv_filters.shape[0]):
        feature_maps[:,:,i] = feature[i,:,:]

    return feature_maps
```

```
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
```

```
29.4 ms ± 1.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Convolutional feature maps shape: (300, 451, 2)
```




```
success!
```

The time taken by the naive and the optimized versions of my convolution operation for the cat image are:

3.16 s ± 428 ms and 19.8 ms ± 223 μs respectively.

Experiment

In the optimized version, I used 2D Fast Fourier Transform function (fft2) and 2D Inverse Fast Fourier Tranform Function (ifft2). The technique or mathematics behind such use is - Fourier Transform of Convolution of Image and Kernel is equals to the multiplication of Fourier Transform of Image and Fourier Transform of Kernel, and the inverse Fourier tranform of the result is the Convolution of Image and Kernel. The convolution operation in spatial domain is multiplication operation in Frequency domain, which simplifies the convolution operation as shown in below equation:

```
A*B = /inv(F)(F(A)xF(B))
```

The output shape after the convolution is exactly same as the input image shape.

Pooling and relu

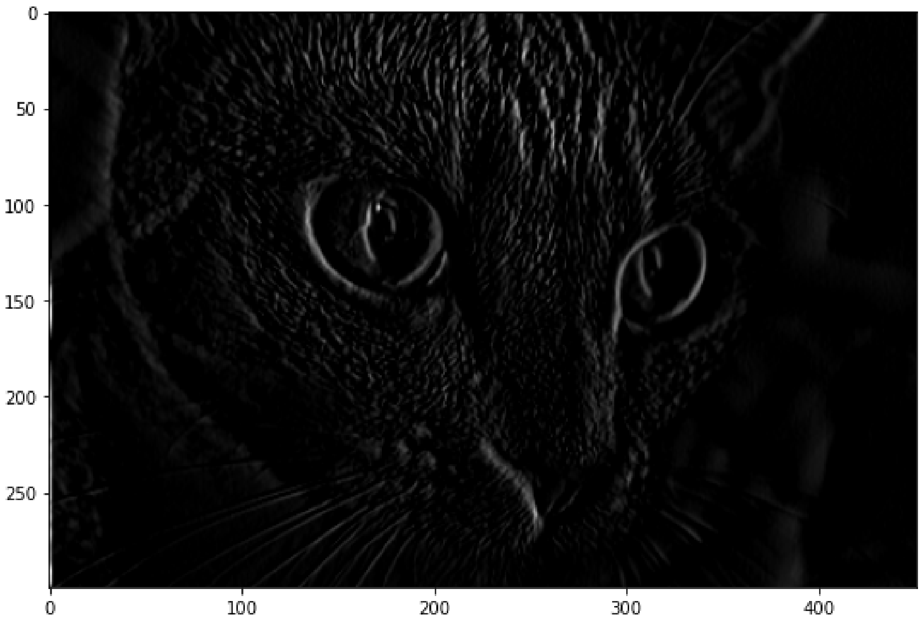
Next, consider the feedforward pooling and ReLU operations.

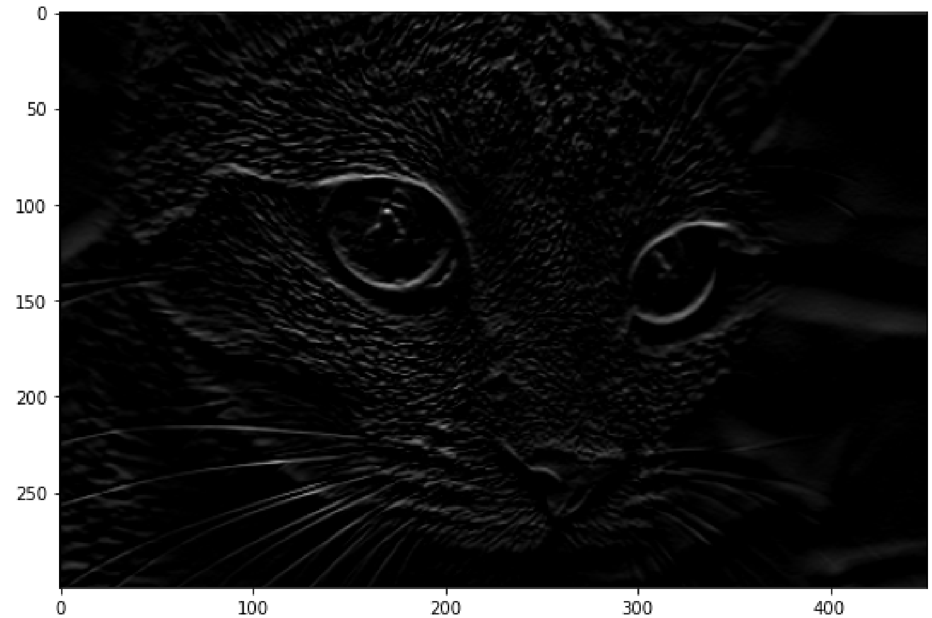
```
# Pooling layer with particular size and stride

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 ReLU and pooling:

```
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()
```





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


```
# 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)
```

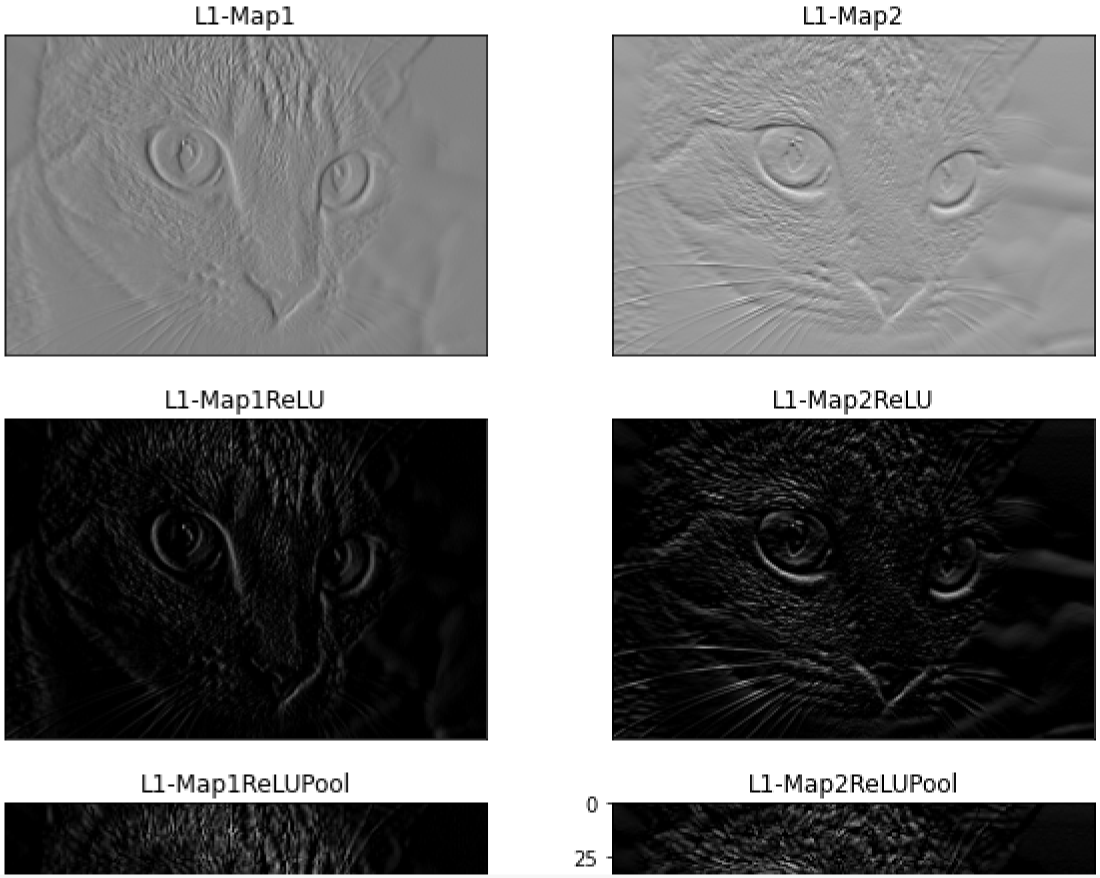
```
conv layer 1...
conv layer 2...
conv layer 3...
```

```
# 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()
```

Input Image

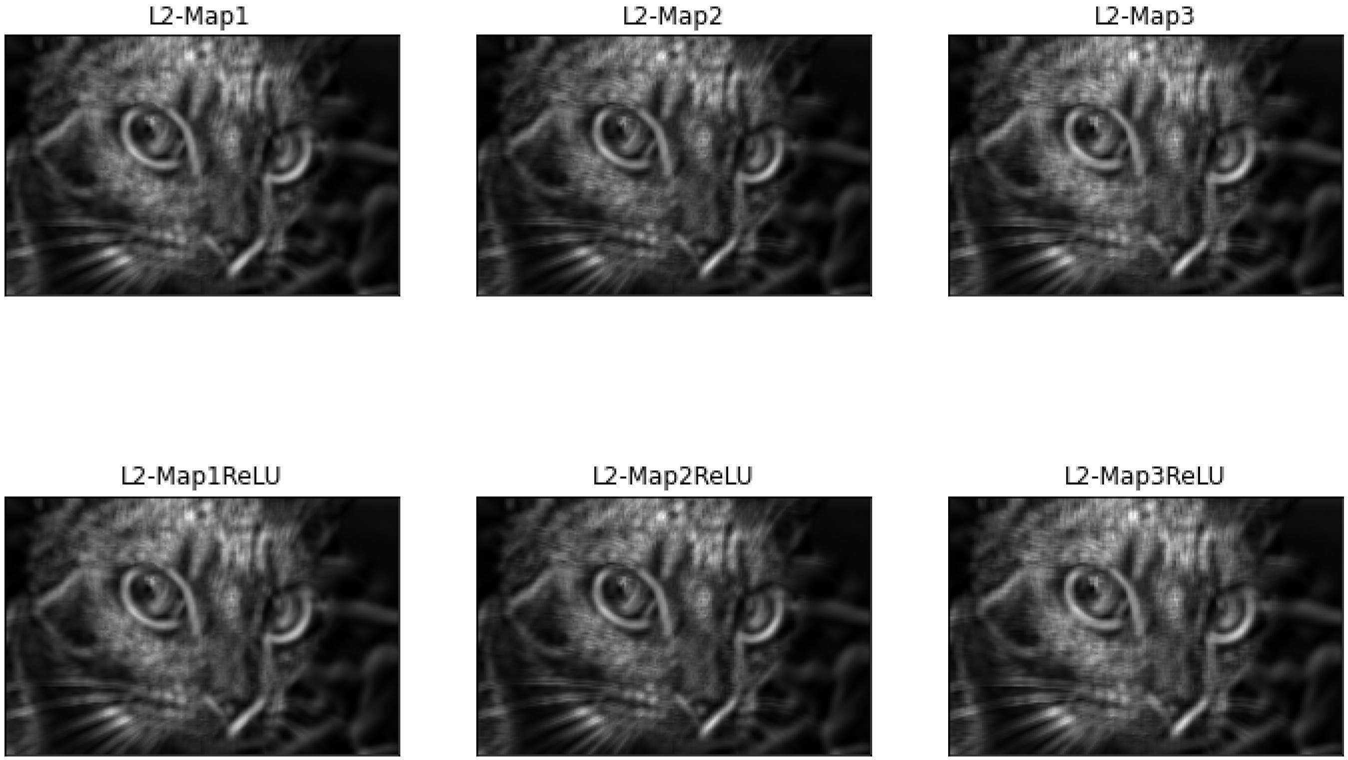
```
# Layer 1
fig1, ax1 = plt.subplots(nrows=3, ncols=2)
fig1.set_figheight(10)
```

```
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()
```




```
# 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()
```

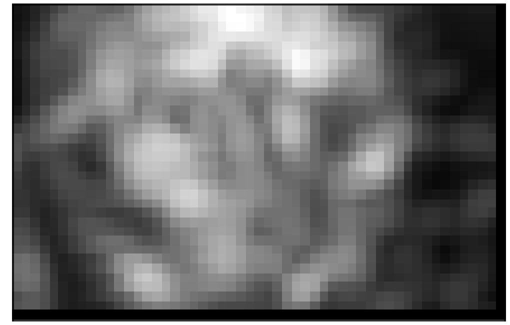



```
# Layer 3

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()
```

L3-Map1

L3-Map1ReLU

L3-Map1ReLUPool



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.

```
# YOUR CODE HERE
#raise NotImplementedError()
from numpy.fft import fft2, ifft2
def convolve2(img, conv_filter):
    output = None

    fft_img = fft2(img)
    fft_conv_filter = fft2(conv_filter, (img.shape[0], img.shape[1]))

    f_img_filter = np.multiply(fft_img, fft_conv_filter)
    output = ifft2(f_img_filter)


    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],
                             img.shape[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 = 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
```

```
#testing and understanding FFT
a = np.ones((2,3))
```

```
fft_a = fft2(a)
b = np.ones((2,3))
print(fft_a)
fft_b = fft2(b, (4,4))
print(fft_b)
```

```
[[6.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j]]
[[ 6.+0.j  0.-2.j  2.+0.j  0.+2.j]
 [ 3.-3.j -1.-1.j  1.-1.j  1.+1.j]
 [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 3.+3.j  1.-1.j  1.+1.j -1.+1.j]]
```

```
%timeit conv2(img,l1_filters)
%timeit conv(img,l1_filters)
```

```
33.7 ms ± 1.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
3.66 s ± 212 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

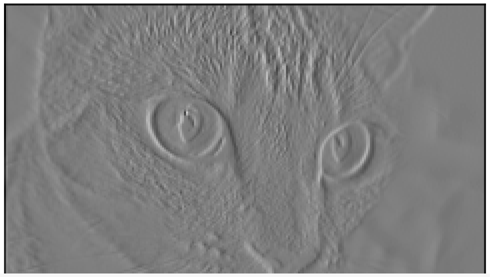
```
# First conv layer

import datetime
start = datetime.datetime.now()

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

# 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)
print(l2_feature_maps.shape)
l2_feature_maps_relu = relu(l2_feature_maps)
l2_feature_maps_relu_pool = pooling(l2_feature_maps_relu, 2, 2)
print(l2_feature_maps_relu_pool.shape)
#print(l2_feature_maps)
```

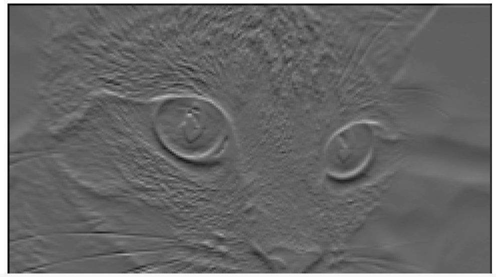
```python
# Third conv layer

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)
print(l3_feature_maps.shape)
l3_feature_maps_relu = relu(l3_feature_maps)
l3_feature_maps_relu_pool = pooling(l3_feature_maps_relu, 2, 2)
print(l3_feature_maps_relu_pool.shape)



stop = datetime.datetime.now()
c = stop - start
elapsed = c.microseconds / 1000 # millisec
print(f"time take by fft used CNN is: {elapsed} ms")
```

```
conv layer 1...
(300, 451, 2)
(150, 226, 2)
conv layer 2...
(150, 226, 3)
(75, 113, 3)
conv layer 3...
(75, 113, 1)
(38, 57, 1)
time take by fft used CNN is: 764.633 ms
```

```python
# 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([])
```

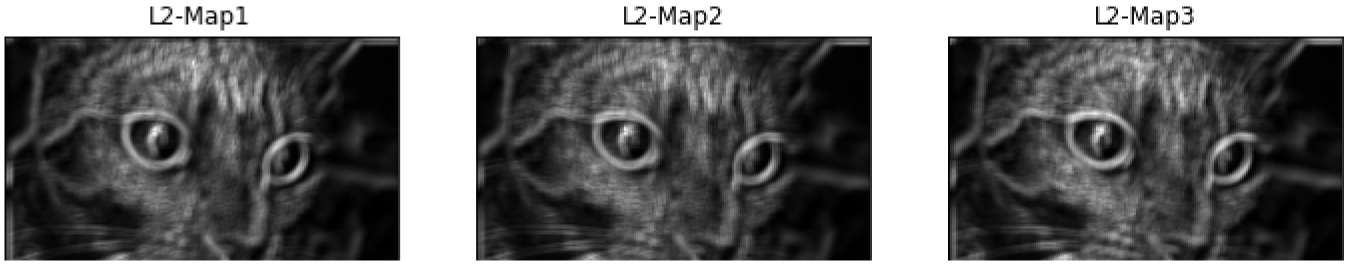
```
ax1[1, 0].get_yaxis().set_ticks([])
ax1[1, 0].set_title("L1-Map1ReLU")

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()
```





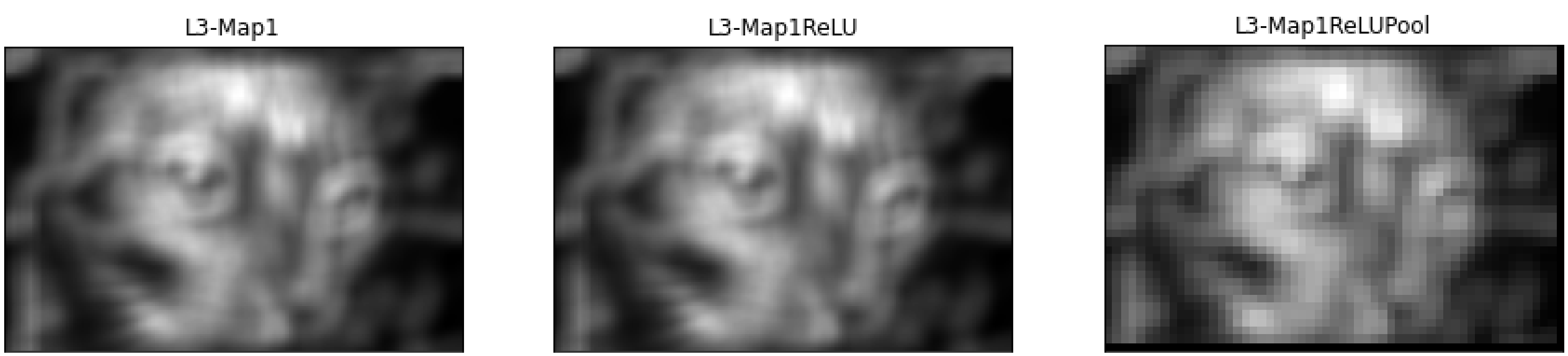

```
# Layer 2
fig2, ax2 = plt.subplots(nrows=3, ncols=3)
fig2.set_figheight(12)
fig2.set_figwidth(12)
ax2[0, 0].imshow(l2_feature_maps[:, :, 0]).set_cmap("gray")
ax2[0, 0].get_xaxis().set_ticks([])
ax2[0, 0].get_yaxis().set_ticks([])
ax2[0, 0].set_title("L2-Map1")

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()
```



Double-click (or enter) to edit

```
# 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()
```



Check the result, explain what you did, and take note of any differences in the result.

The output image from the Convolutional layer, relu and pooling layer are pretty much similar to previous one. Yes, of course the size of output from convolution layer is different from the previous one. Here I used same convolution. In here, I used simply performed inverse fourier transform of multiplication of fft of image and filter, only one corresponding layer at a time. Actually, I resized the fourier transform of the filter to match with the image size. Instead of this for better efficiency I could resize the fourier transform of the image to match with the filter size and perform inverse fourier transform of this result to be the shape of original image. That way we will lose some image quality because of the rejection of higher frequencies components, though image can be well represented by the few lower frequencies components as well.

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].

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()`.

```
import torch
import torch.cuda as cuda
import torch.nn as nn
import os

from torch.autograd import Variable
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
```

```
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 transfrom the input elements (pixels) so that their mean over the entire training dataset is 0 and its standard deviation is 1.

```
# Desired mean and standard deviation

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)
```

```
print(mnist_train)
print(mnist_valid)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Split: train
    Root Location: ./data
    Transforms (if any): Compose(
                             ToTensor()
                             Normalize(mean=[0.0], std=[1.0])
                         )
    Target Transforms (if any): None
```

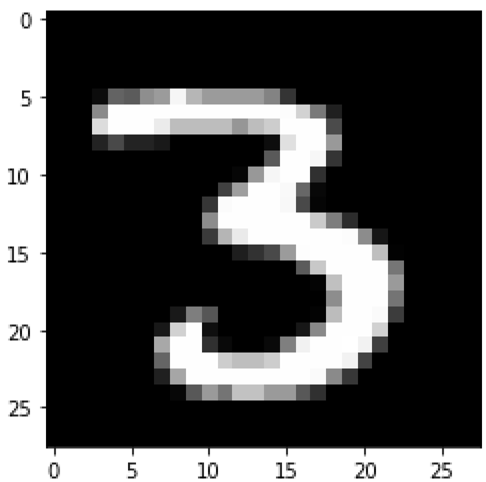
```
Dataset MNIST
    Number of datapoints: 10000
    Split: test
    Root Location: ./data
    Transforms (if any): Compose(
                             ToTensor()
                             Normalize(mean=[0.0], std=[1.0])
                         )
    Target Transforms (if any): None
```

```
print(mnist_train[2][0].shape)
```

```
torch.Size([1, 28, 28])
```

```
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)
```



Create **loader** that can connect to the dataset which you created

```
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, batch_size=batch_size, shuffle=True, num_workers=1)
mnist_valid_loader = torch.utils.data.DataLoader(mnist_valid, batch_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.

Important words (for PyTorch)

Tensor - any matrix arrays which use for calculation, you can call input, output, weight as any tensors. In CNNs, we use "input tensor" as input; i.e. image, and "output tensor" as output.

Kernel - filter tensor, or weight tensor. In computer vision, we might call mask tensor or mask matrix.

Channel - number of depth in tensor, so sometime we call **depth**.

Feature - Specific characteristic information for using in **dense layers** or **fully connect layers**.

Feature extraction - the process of transforming raw data into numerical features that can be processed while preserving the information in the original data set.

Stride - The jump necessary to go from one element to the next one in the specified dimension dim . A tuple of all strides is returned when no argument is passed in.

Padding - the zero array extends in both sides of tensor.

In PyTorch, the function of CNNs is no need to input size, but it needs to fill number of channels and kernel size, including operation in the layer. For dense layer or fully layer, we need to set input features number and output features number. Thus, it is necessary to understand how to calculate tensors and features size in each layer.

PyTorch model architecture

PyTorch deep learning models come in (at least) two possible styles:

1. The PyTorch Sequential API is very expressive when we have a straightforward sequence of operations to perform on the input.
2. The PyTorch Module allows more flexible transformations of inputs, combination of multiple inputs, generation of multiple outputs, and so on.

Number of parameters and output tensors size calculation

CNN parameters

Kernel size k in 1 layer for 1 channel output can be calculated by

$$k = k_w \times k_h \times i_c$$

when k_w is width of kernel, k_h is width of kernel, and i_c is input channels. We need to have o_c kernels for release o_c output channels. Therefore, for 1 layer of CNN, number of parameters can be calculated as

$$n_p = k \times o_c = (k_w \times k_h \times i_c) \times o_c$$

For bias in CNNs, it usually become all zeros, but we can assign bias CNNs in PyTorch. The **bias size** is equal to the **output tensor size**.

Fully connect parameters

Weight size s_w in 1 layers can be calculated by

$$s_w = i_f \times o_f$$

when i_f is input features, and o_f is output features. For bias, the size is equal to **output feature size**.

We can calculate that the total parameters number is the parameters of all layers, so the network size can be used from the parameters number. It is useful to tell how efficient of the network (How fast)

output tensors size

If we have an input tensor or image input size $w \times h$ which want to convolution with $k_w \times k_h$ kernel size with padding p and stride s, we can calculate output tensor size as:

$$output_{size} = \lfloor \frac{w + 2p - k_w}{s} + 1 \rfloor \times \lfloor \frac{h + 2p - k_h}{s} + 1 \rfloor$$

For example, input image in the first layer is 224×224. Using 11×11 of kernel size with padding 2 and stride 4. We calculate

$$output_{size} = \lfloor \frac{w + 2p - k_w}{s} + 1 \rfloor = \lfloor \frac{224 + 2(2) - 11}{4} + 1 \rfloor = \lfloor 55.25 \rfloor = 55$$

```
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)            # Dropout 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

```
# 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, momentum=0.9)
```

Print out the network

```
print(net)
```

```
CNN_Model(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (relu1): ReLU()
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc1): Linear(in_features=2000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Training loop

```
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 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 / 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]))
```

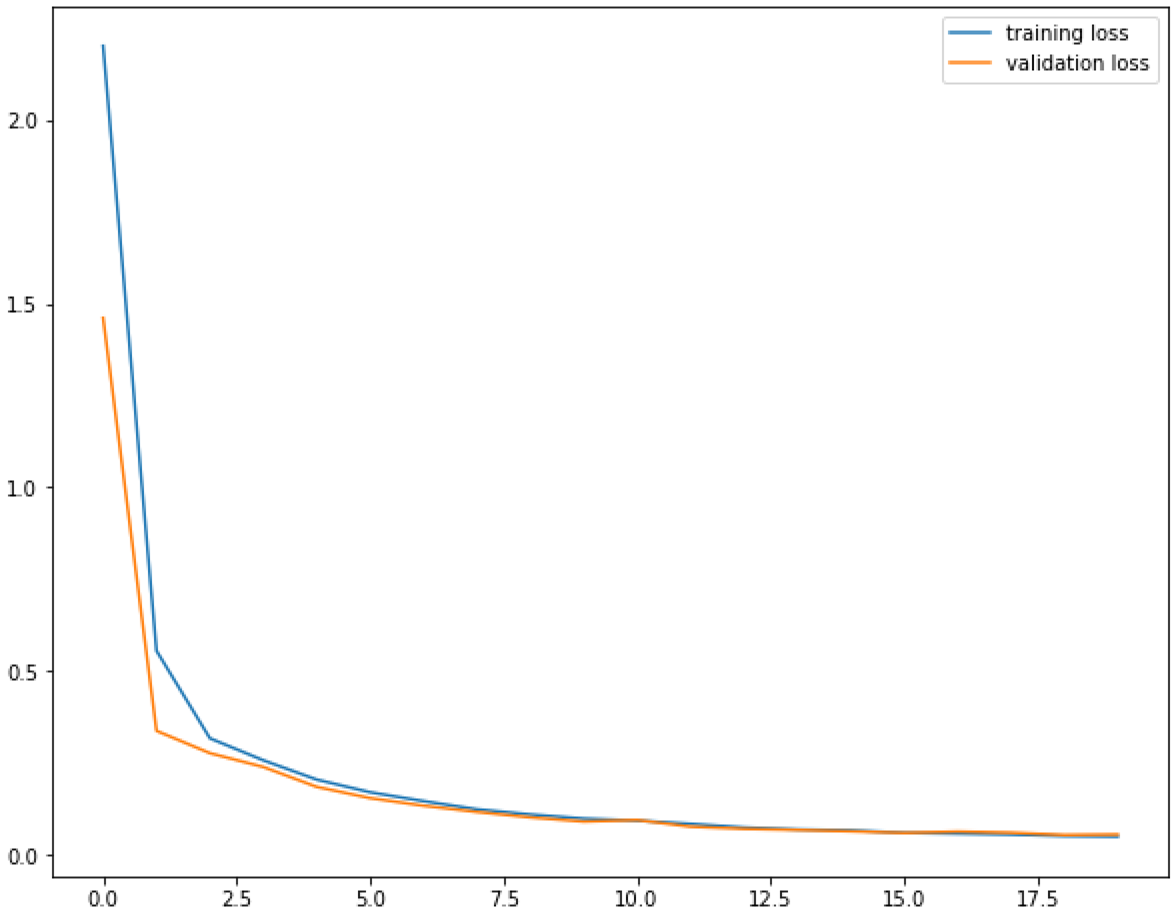
```
Epoch 1/20, Tr Loss: 2.2015, Tr Acc: 26.7450, Val Loss: 1.4613, Val Acc: 58.0500
Epoch 2/20, Tr Loss: 0.5556, Tr Acc: 83.7750, Val Loss: 0.3378, Val Acc: 90.1200
Epoch 3/20, Tr Loss: 0.3175, Tr Acc: 90.5033, Val Loss: 0.2771, Val Acc: 91.7900
Epoch 4/20, Tr Loss: 0.2577, Tr Acc: 92.3017, Val Loss: 0.2395, Val Acc: 92.9500
Epoch 5/20, Tr Loss: 0.2052, Tr Acc: 93.8650, Val Loss: 0.1854, Val Acc: 94.7200
Epoch 6/20, Tr Loss: 0.1708, Tr Acc: 94.8933, Val Loss: 0.1547, Val Acc: 95.4100
Epoch 7/20, Tr Loss: 0.1466, Tr Acc: 95.6117, Val Loss: 0.1341, Val Acc: 95.9900
Epoch 8/20, Tr Loss: 0.1243, Tr Acc: 96.2517, Val Loss: 0.1171, Val Acc: 96.4000
Epoch 9/20, Tr Loss: 0.1102, Tr Acc: 96.6900, Val Loss: 0.1023, Val Acc: 96.9000
Epoch 10/20, Tr Loss: 0.0990, Tr Acc: 96.9917, Val Loss: 0.0910, Val Acc: 97.2300
Epoch 11/20, Tr Loss: 0.0936, Tr Acc: 97.1667, Val Loss: 0.0951, Val Acc: 97.0300
Epoch 12/20, Tr Loss: 0.0847, Tr Acc: 97.4250, Val Loss: 0.0771, Val Acc: 97.6300
Epoch 13/20, Tr Loss: 0.0749, Tr Acc: 97.7417, Val Loss: 0.0712, Val Acc: 97.7700
Epoch 14/20, Tr Loss: 0.0694, Tr Acc: 97.8917, Val Loss: 0.0682, Val Acc: 97.9400
Epoch 15/20, Tr Loss: 0.0670, Tr Acc: 97.9283, Val Loss: 0.0645, Val Acc: 98.0100
Epoch 16/20, Tr Loss: 0.0603, Tr Acc: 98.1733, Val Loss: 0.0601, Val Acc: 98.1000
Epoch 17/20, Tr Loss: 0.0581, Tr Acc: 98.2167, Val Loss: 0.0637, Val Acc: 98.0200
Epoch 18/20, Tr Loss: 0.0557, Tr Acc: 98.2400, Val Loss: 0.0608, Val Acc: 98.1400
Epoch 19/20, Tr Loss: 0.0516, Tr Acc: 98.3833, Val Loss: 0.0545, Val Acc: 98.3200
Epoch 20/20, Tr Loss: 0.0502, Tr Acc: 98.4633, Val Loss: 0.0558, Val Acc: 98.2400
```

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.

```
# save the model
torch.save(net.state_dict(), "./3.model.pth")
```

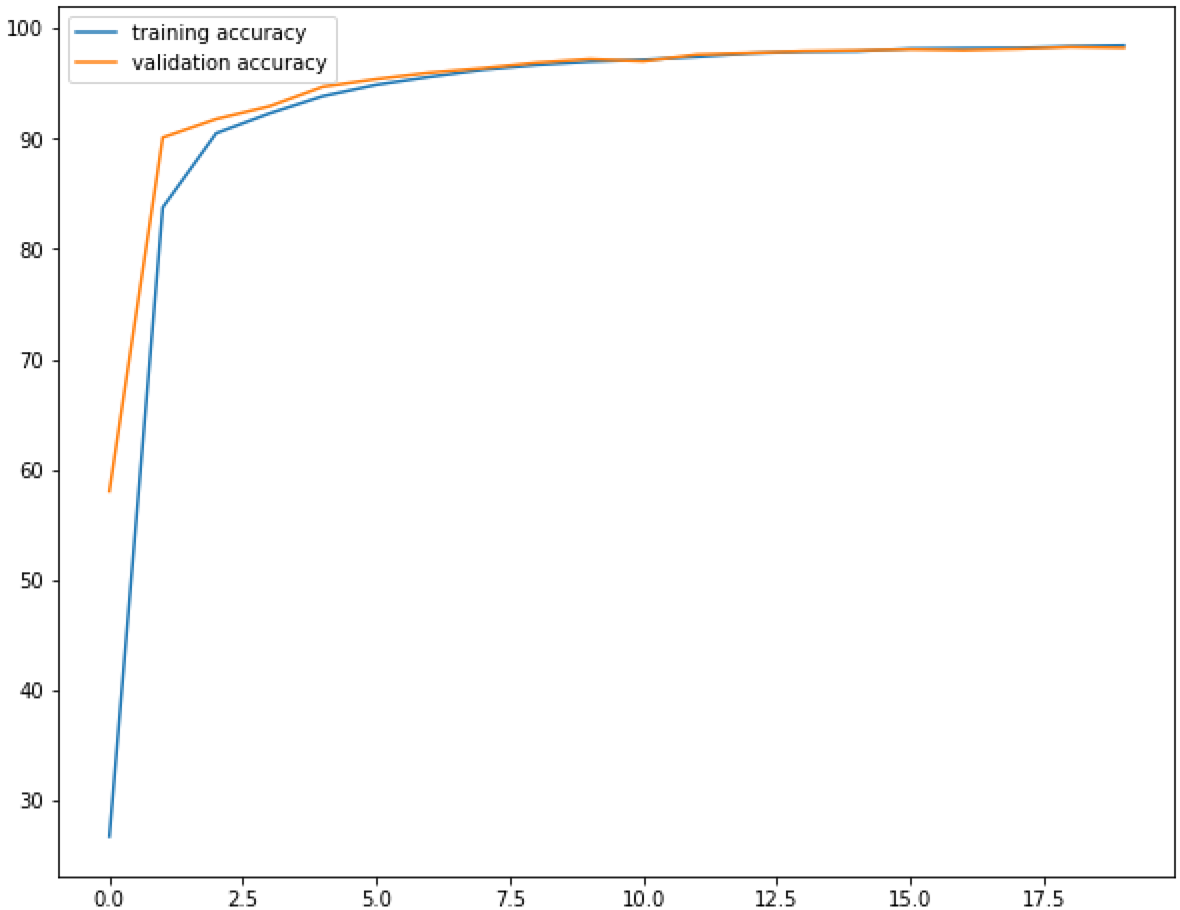
Next, let's visualize the loss and accuracy

```
# 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()
```




```
# 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()
```



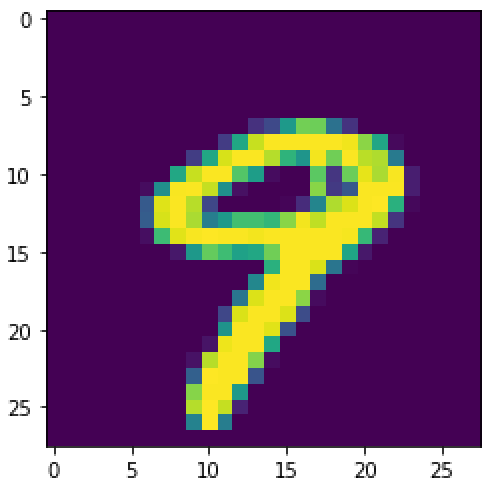
What can you conclude from the loss and accuracy curves?

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.

```
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)
```




```
output
```

```
tensor([[ -3.8013, -11.5048,  -4.7082,  -0.1845,   1.5572,   1.3673,  -7.2324,
           7.2792,   5.3050,  11.6134]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

```
_, 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. 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.

```
import numpy as np
import matplotlib.pyplot as plt
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
```

```
a = torch.tensor([2,3])
b = 2
c = [(a,b)]
```

```
from sklearn.model_selection import train_test_split

training_data_dir = "../ML2022-09-Deep learning II/train/"
filenames = [name for name in os.listdir(training_data_dir)]

label = []

for file in filenames:
    if 'dog' in file:
        label.append(1)
```

```python
    if 'cat' in file:
        label.append(0)

# validation_data_dir = "/home/Datasets/cats-and-dogs/test1/"
# validation_filenames = [name for name in os.listdir(validation_data_dir)]

train_filenames, valid_filenames, train_label, valid_label = train_test_split(filenames, label , test_size=0.2, train_size=0.8, random
```

```python
print(len(train_filenames), len(train_label))
print(len(valid_filenames), len(valid_label))
```

```
20000 20000
5000 5000
```

```python
from torch.utils.data import Dataset
from PIL import Image

class CatDogDataset(Dataset):

    def __init__(self, label, filenames, root_dir, size):
        self.label = label
        self.filenames = filenames
        self.root_dir = root_dir
        self.size = size

    def __getitem__(self, idx):

        file = self.filenames[idx]
        label = self.label[idx]

        with Image.open(self.root_dir+file) as im:
            image = im.resize(self.size)
            img_np = np.array(image)
            img_np = np.transpose(img_np, (2,0,1)) #(nc, h, w)
            tensor_img = torch.FloatTensor(img_np)
            tensor_img = tensor_img/tensor_img.max()

            sample = (tensor_img, label)

        return sample
```

```
    def __len__(self):
        return len(self.label)
```

```
size = (124,124)
catdog_train = CatDogDataset(label = train_label, filenames = train_filenames, root_dir = training_data_dir, size = size)
catdog_valid = CatDogDataset(label = valid_label, filenames = valid_filenames, root_dir = training_data_dir, size = size)
```

```
print(catdog_train[0])
print(len(catdog_train))
```

```
(tensor([[[0.4314, 0.2431, 0.2667,  ..., 0.4549, 0.4392, 0.3725],
         [0.6000, 0.5098, 0.3176,  ..., 0.4667, 0.4627, 0.4039],
         [0.6078, 0.6549, 0.6353,  ..., 0.4706, 0.4471, 0.4157],
         ...,
         [0.4588, 0.4353, 0.4235,  ..., 0.5882, 0.5922, 0.5882],
         [0.4392, 0.4157, 0.4471,  ..., 0.5843, 0.5922, 0.5725],
         [0.4078, 0.3686, 0.4549,  ..., 0.5922, 0.5882, 0.5725]],

        [[0.3804, 0.1882, 0.1922,  ..., 0.4588, 0.4314, 0.3412],
         [0.5176, 0.4275, 0.2235,  ..., 0.4588, 0.4235, 0.3451],
         [0.4902, 0.5373, 0.5098,  ..., 0.4549, 0.4078, 0.3569],
         ...,
         [0.4588, 0.4275, 0.4196,  ..., 0.5882, 0.5922, 0.5882],
         [0.4353, 0.4118, 0.4392,  ..., 0.5843, 0.5922, 0.5725],
         [0.4078, 0.3804, 0.4745,  ..., 0.5882, 0.5843, 0.5725]],

        [[0.4078, 0.1843, 0.1608,  ..., 0.4510, 0.4118, 0.3020],
         [0.5686, 0.4510, 0.2196,  ..., 0.4588, 0.4118, 0.3137],
         [0.5176, 0.5647, 0.5373,  ..., 0.4588, 0.4000, 0.3255],
         ...,
         [0.4667, 0.4549, 0.4627,  ..., 0.5882, 0.5922, 0.5882],
         [0.4549, 0.4471, 0.4941,  ..., 0.5882, 0.5922, 0.5725],
         [0.4039, 0.4039, 0.5255,  ..., 0.6275, 0.6078, 0.5804]]]), 1)
20000
```

DataLaoder

```
batch_size = 50
catdog_train_loader = torch.utils.data.DataLoader(catdog_train, batch_size=batch_size, shuffle=True, num_workers=2)
catdog_valid_loader = torch.utils.data.DataLoader(catdog_valid, batch_size=batch_size, shuffle=True, num_workers=2)
```

```
class CNN_Model(nn.Module):

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

        #input image = (3, 256, 256)
#       #nn.Conv2d(in_channels, out_channels (num of filters), kernel_size)
        # NOTE: All Conv2d layers have a default padding of 0 and stride of 1,
#         self.conv1 = nn.Conv2d(3, 10, kernel_size=5)      # 252 x 252 x 10  (after 1st convolution)
#         self.relu1 = nn.ReLU()                            # Same as above

#         # Convolution Layer 2
#         self.conv2 = nn.Conv2d(10, 20, kernel_size=5)     # 248 x 248 x 20  (after 2nd convolution)
#         #self.conv2_drop = nn.Dropout2d(p=0.5)            # Dropout is a regularization technqiue we discussed in class
#         self.maxpool2 = nn.MaxPool2d(2)                   # 124 x 124 x 20  (after pooling)
#         self.relu2 = nn.ReLU()                            # Same as above

        #Convolution layer 3
        self.conv3 = nn.Conv2d(3, 20, kernel_size=5)  # 120 x 120 x 20
        self.maxpool3 = nn.MaxPool2d(2)                # 60 x 60 x 20
        self.relu3 = nn.ReLU()

        #Convolution layer 4
        self.conv4 = nn.Conv2d(20, 20, kernel_size=5)  # 56 x 56 x 20
        self.maxpool4 = nn.MaxPool2d(2)                # 28 x 28 x 20
        self.relu4 = nn.ReLU()

        #Convolution layer 5
        self.conv5 = nn.Conv2d(20, 20, kernel_size=5)  # 24 x 24 x 20
        self.maxpool5 = nn.MaxPool2d(2)                # 12 x 12 x 20
        self.relu5 = nn.ReLU()

        # Fully connected layers
        self.fc1 = nn.Linear(2880, 50)
        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)
#         x = self.maxpool2(x)
#         x = self.relu2(x)

        # Convolution Layer 3
        x = self.conv3(x)
        x = self.maxpool3(x)
        x = self.relu3(x)

        # Convolution Layer 4
        x = self.conv4(x)
        x = self.maxpool4(x)
        x = self.relu4(x)

        # Convolution Layer 5
        x = self.conv5(x)
        x = self.maxpool5(x)
        x = self.relu5(x)

        # Switch from activation maps to vectors
        x = x.view(-1, 2880)

        # 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
```

Optimizers, Loss Functions, Model object instantiation

```
# The model
net = CNN_Model()
```

```
device = 'cuda:0'

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

# Our loss function
criterion = nn.CrossEntropyLoss()

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

```
len(catdog_train_loader.dataset)
```

```
20000
```

Training with validation

```
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(catdog_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(device)
        classes = classes.cuda(device)

    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(catdog_train_loader.dataset))))

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


############################
# 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(catdog_valid_loader):

    # Convert torch tensor to Variable
```

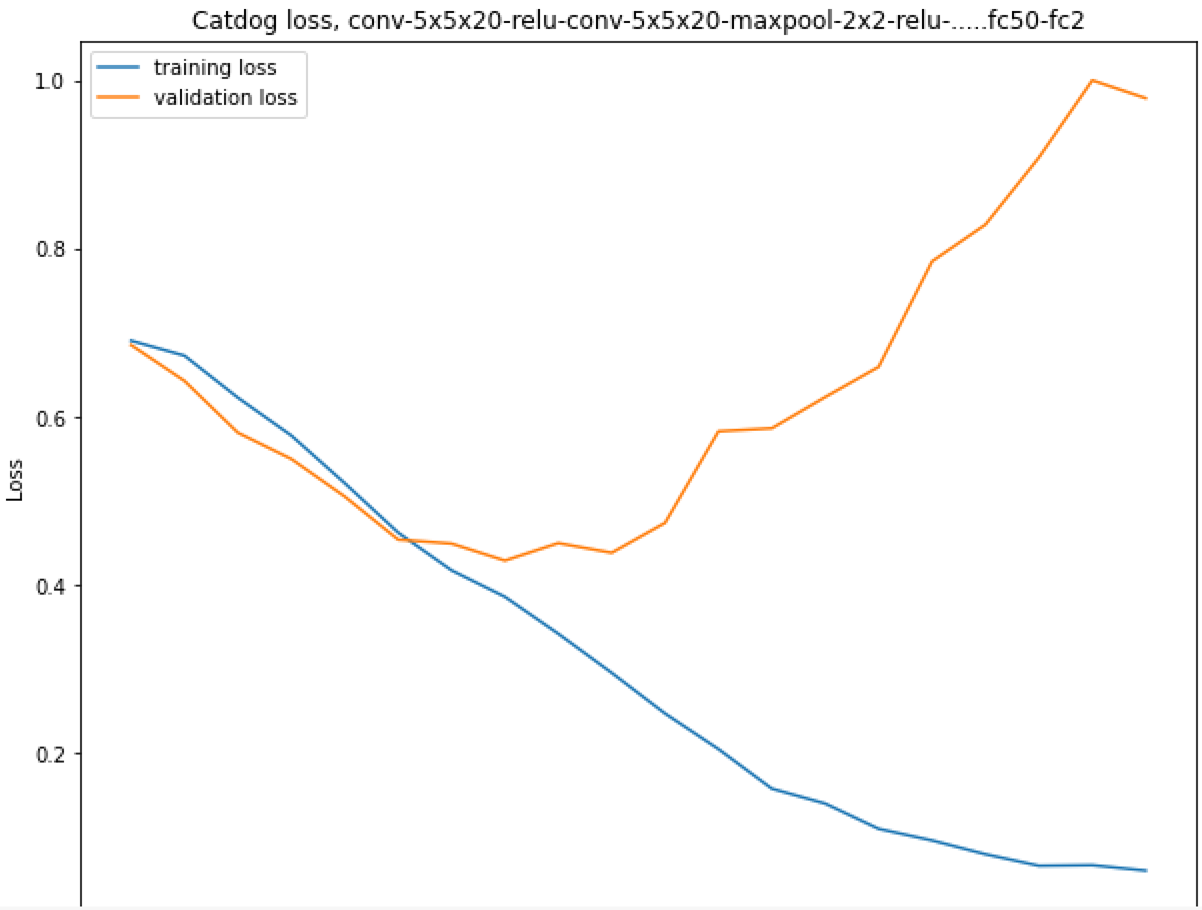
```
        items = Variable(items)
        classes = Variable(classes)

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

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

        # Record the correct predictions for validation 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(catdog_valid_loader.dataset) * 100.0)

    valid_accuracy.append((100 * correct.item() / float(len(catdog_valid_loader.dataset))))

    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: 0.6911, Tr Acc: 51.6250
Epoch 1/20, Tr Loss: 0.6911, Tr Acc: 51.6250, Val Loss: 0.6860, Val Acc: 54.5400
Epoch 2/20, Tr Loss: 0.6733, Tr Acc: 58.2350
Epoch 2/20, Tr Loss: 0.6733, Tr Acc: 58.2350, Val Loss: 0.6433, Val Acc: 63.0000
Epoch 3/20, Tr Loss: 0.6233, Tr Acc: 65.1900
Epoch 3/20, Tr Loss: 0.6233, Tr Acc: 65.1900, Val Loss: 0.5816, Val Acc: 69.5600
Epoch 4/20, Tr Loss: 0.5783, Tr Acc: 69.8150
Epoch 4/20, Tr Loss: 0.5783, Tr Acc: 69.8150, Val Loss: 0.5504, Val Acc: 72.3600
Epoch 5/20, Tr Loss: 0.5216, Tr Acc: 73.8400
Epoch 5/20, Tr Loss: 0.5216, Tr Acc: 73.8400, Val Loss: 0.5056, Val Acc: 74.8600
Epoch 6/20, Tr Loss: 0.4627, Tr Acc: 78.0200
Epoch 6/20, Tr Loss: 0.4627, Tr Acc: 78.0200, Val Loss: 0.4544, Val Acc: 78.7200
Epoch 7/20, Tr Loss: 0.4178, Tr Acc: 80.5950
Epoch 7/20, Tr Loss: 0.4178, Tr Acc: 80.5950, Val Loss: 0.4496, Val Acc: 78.7200
```

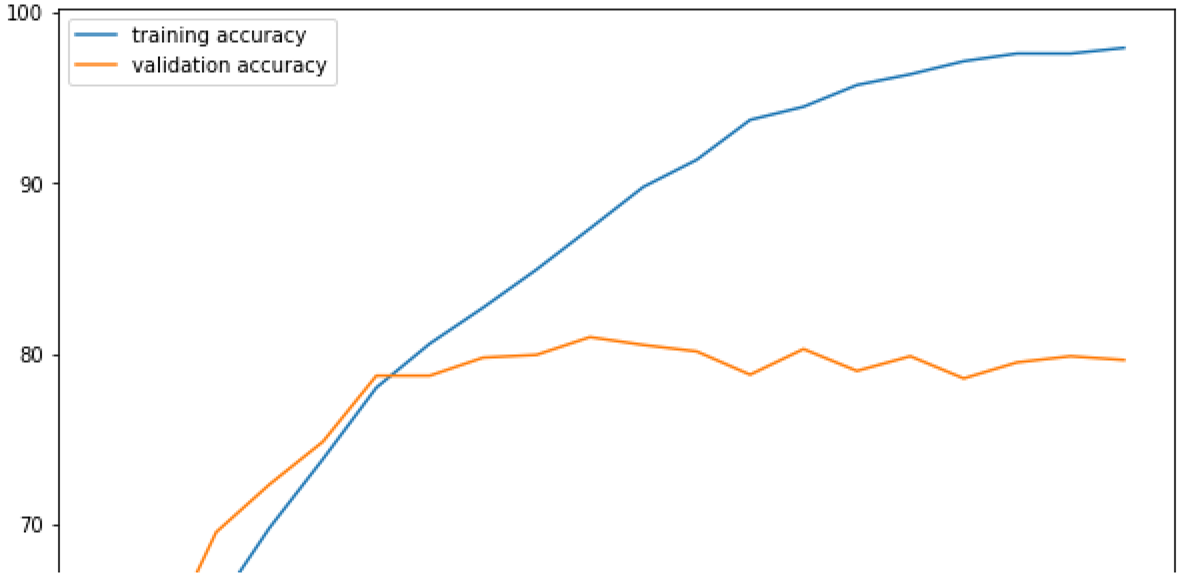
```
Epoch 8/20, Tr Loss: 0.3863, Tr Acc: 82.7050
Epoch 8/20, Tr Loss: 0.3863, Tr Acc: 82.7050, Val Loss: 0.4294, Val Acc: 79.7800
Epoch 9/20, Tr Loss: 0.3423, Tr Acc: 84.9350
Epoch 9/20, Tr Loss: 0.3423, Tr Acc: 84.9350, Val Loss: 0.4502, Val Acc: 79.9400
Epoch 10/20, Tr Loss: 0.2958, Tr Acc: 87.3350
Epoch 10/20, Tr Loss: 0.2958, Tr Acc: 87.3350, Val Loss: 0.4387, Val Acc: 80.9800
Epoch 11/20, Tr Loss: 0.2473, Tr Acc: 89.7700
Epoch 11/20, Tr Loss: 0.2473, Tr Acc: 89.7700, Val Loss: 0.4743, Val Acc: 80.5200
Epoch 12/20, Tr Loss: 0.2049, Tr Acc: 91.3600
Epoch 12/20, Tr Loss: 0.2049, Tr Acc: 91.3600, Val Loss: 0.5833, Val Acc: 80.1400
Epoch 13/20, Tr Loss: 0.1578, Tr Acc: 93.6950
Epoch 13/20, Tr Loss: 0.1578, Tr Acc: 93.6950, Val Loss: 0.5869, Val Acc: 78.7800
Epoch 14/20, Tr Loss: 0.1400, Tr Acc: 94.4650
Epoch 14/20, Tr Loss: 0.1400, Tr Acc: 94.4650, Val Loss: 0.6241, Val Acc: 80.2800
Epoch 15/20, Tr Loss: 0.1100, Tr Acc: 95.7350
Epoch 15/20, Tr Loss: 0.1100, Tr Acc: 95.7350, Val Loss: 0.6600, Val Acc: 79.0000
Epoch 16/20, Tr Loss: 0.0962, Tr Acc: 96.3650
Epoch 16/20, Tr Loss: 0.0962, Tr Acc: 96.3650, Val Loss: 0.7856, Val Acc: 79.8600
Epoch 17/20, Tr Loss: 0.0798, Tr Acc: 97.1300
Epoch 17/20, Tr Loss: 0.0798, Tr Acc: 97.1300, Val Loss: 0.8294, Val Acc: 78.5600
Epoch 18/20, Tr Loss: 0.0661, Tr Acc: 97.5800
Epoch 18/20, Tr Loss: 0.0661, Tr Acc: 97.5800, Val Loss: 0.9092, Val Acc: 79.5000
Epoch 19/20, Tr Loss: 0.0667, Tr Acc: 97.5750
Epoch 19/20, Tr Loss: 0.0667, Tr Acc: 97.5750, Val Loss: 1.0011, Val Acc: 79.8600
Epoch 20/20, Tr Loss: 0.0603, Tr Acc: 97.9100
Epoch 20/20, Tr Loss: 0.0603, Tr Acc: 97.9100, Val Loss: 0.9799, Val Acc: 79.6400
```

```
# save the model
torch.save(net.state_dict(), "./3.catdog_model.pth")
```

```
# 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('Catdog loss, conv-5x5x20-relu-conv-5x5x20-maxpool-2x2-relu-.....fc50-fc2')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




```
# 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()
```



Testing

```
image_index = 150
image_name = valid_filenames[image_index]
with Image.open(training_data_dir+image_name) as im:
    im.show()

img = catdog_valid[image_index][0].resize_((1, 3, 124, 124))
img = Variable(img)
label = catdog_valid[image_index][1]
net.eval()

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

output = net(img)
output = output.cpu()
print(output)
pred_label = torch.argmax(output)
if pred_label==1:
    print("Prediction is Dog.")
```

```
else:
    print("Prediction is Cat.")
```

```
tensor([[ 2.5603, -2.0267]], grad_fn=<ToCopyBackward0>)
Prediction is Cat.
```

Results:

(There are two model that I got in two training. One was before the upgrade of puffer and one was after the upgrade. So this report contains the result about the first model. But the output data and the plots are the result of second run. The previous model is in the models folder. I loaded that model to test the sample as shown in the second sample test in the last section "Loading the saved model and evaluating)

The model was trained on 80% of the total Training Datasets (25,000) and validated on remaining 20% (5,000), since there was no label for Test Datasets. The model is trained upto 20 epochs. The model had the least validation loss at 10th epoch which is 0.4508 and had best validation accuracy at 15th epoch which is 81.62%. The model that I used for sample test was of 20th epoch which has the validation accuracy of 80%. It means our model fails to classify 1000 images and successes to classify remaining 4,000 images. This is not so good and this can be improved by designing better Model, using dropouts if the new designed model is complex, training on larger datasets and many more.