CSGI-GA 3033-076 Vision meets Machine Learning (Part 1)

Homework 1

In this assignment, you will be introduced to few Python libraries that will be needed for this course. This assumes that you can write basic Python programs. If you do not know how to code in Python, this would be a good time for you to learn.

The main goals of this assignment are to introduce how you could:

1. Do basic vector operations.
2. Use Matplotlib to plot a function, and to also display a collection of images.
3. Summarize a neural network architecture.

You may have already worked with these libraries before, in which case we hope this could be a nice review.

Also accompanying each section below, there are a few questions (**4 questions in total**). Please give your answers in the space provided.

Part 2 of this assignment is available in hw1_part2.pdf (**3 questions in total**). Please provide the code to those questions at the end of this notebook.

Part 1: Numpy

NumPy (https://numpy.org/) is a highly popular Python library within the scientific community. With NumPy, you can operate on n-dimensional arrays and perform a number of mathematical operations on them. For example, you can generate an array of random numbers (https://numpy.org/doc/stable/reference/random/index.html), compute element-wise sum/difference between 2 arrays (https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs), find the L2 norm of an array (https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html), etc.

NumPy is mostly C under the hood, and it's highly optimized. So it's highly recommended that you use NumPy whenever possible. We will show some basic operations here. This is by no means complete - NumPy is a huge library and it's impossible to cover everything.

First, let's import numpy:

In [1]:
```
import numpy as np
import torch
import torch.nn as nn
```

You can convert a 2d list of numbers into a numpy array using:

In [2]:
```
nums = [[1, 2, 3, 4], [5, 6, 7, 8]]
np.array(nums)
```

Out[2]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

`linspace(start, end, n)` can be used to generate an array of `n` evenly spaced numbers from `start` to `end` (both inclusive).

In [3]:
```
np.linspace(0, 10, 21)
```

Out[3]:
```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

You can multiply this array with a scalar:

In [4]:
```
2*np.linspace(0, 10, 21)
```

Out[4]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
       13., 14., 15., 16., 17., 18., 19., 20.])
```

A 1d array can also be reshaped into a 2d array. To see the dimensions of a numpy array `arr`, use `arr.shape`.

In [5]:
```
arr = np.linspace(0, 99, 100)
new_arr = arr.reshape((10, 10))
print("Original shape", arr.shape)
print("Reshaped array shape", new_arr.shape)
print("Reshaped array:\n", new_arr)
```

```
Original shape (100,)
Reshaped array shape (10, 10)
Reshaped array:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
 [50. 51. 52. 53. 54. 55. 56. 57. 58. 59.]
 [60. 61. 62. 63. 64. 65. 66. 67. 68. 69.]
 [70. 71. 72. 73. 74. 75. 76. 77. 78. 79.]
 [80. 81. 82. 83. 84. 85. 86. 87. 88. 89.]
 [90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```

You can learn more about NumPy from [the quickstart page (https://numpy.org/doc/stable/user/quickstart.html)](https://numpy.org/doc/stable/user/quickstart.html).

Question 1

Find the dot product of these 2 arrays using NumPy:

$$A = \begin{bmatrix} 19 & 7 & -15 \\ 12 & 59 & 27 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 \\ -7 \\ -13 \end{bmatrix}$$

In [6]:
```
# TODO:
A = np.array([[19,7,-15],[12,59,27]])
B = np.array([[3],[-7],[-13]])
# Compute dot product A.B and print it
dot_prod = np.dot(A,B)
print(dot_prod)
```

```
[[ 203]
 [-728]]
```

Question 2

Create 2 matrices:

1. C: [[1, 2, 3], [4, 5, 6]]
2. D: [[10, 11, 12], [13, 14, 15]]

and concatenate them to get this matrix:

[[1, 2, 3], [4, 5, 6], [10, 11, 12], [13, 14, 15]]

In [7]:
```
# TODO:
C = np.array( [[1, 2, 3], [4, 5, 6]])
D = np.array([[10, 11, 12], [13, 14, 15]])
# Concatenate and print the resulting matrix
new_arr = np.concatenate((C,D), axis=0)
print(new_arr)
new_arr.shape
```

```
[[ 1  2  3]
 [ 4  5  6]
 [10 11 12]
 [13 14 15]]
```

Out[7]: (4, 3)

Part 2: Plotting in Python

Matplotlib is the most commonly used plotting Python library. There are other libraries out there (such as seaborn, bokeh, etc.), but we will focus on Matplotlib.

Our first step would be to plot the sigmoid activation function:

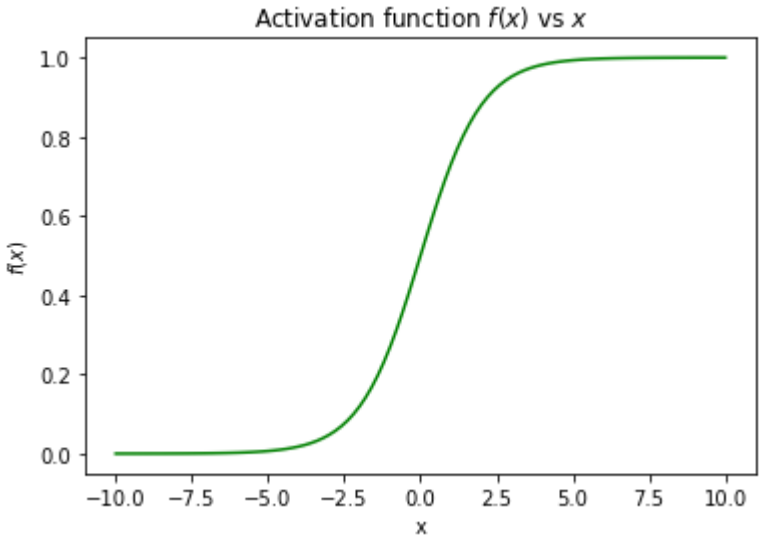
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

In [8]:
```
x = np.linspace(-10, 10, 101) # get 101 data points
sigma = 1/(1+ np.exp(-x))     # get f(x)
```

In [9]:
```
print(x.shape, sigma.shape)
```

```
(101,) (101,)
```

In [10]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(x, sigma, 'g-') # plot x, f(x)
ax.set_xlabel('x')
ax.set_ylabel('$f(x)$')
ax.set_title('Activation function $f(x)$ vs $x$')
plt.show()
```



Question 3

On the same graph:

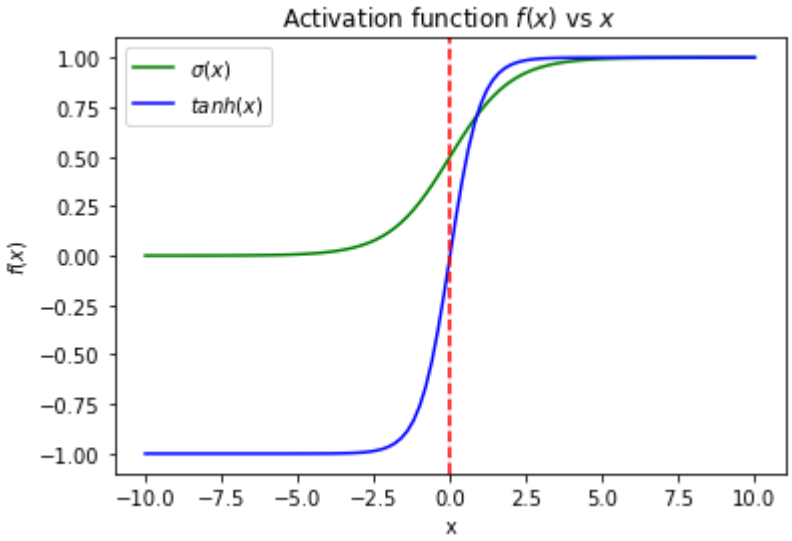
1. Plot the `tanh` activation function (using a blue line). For reference,
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
2. Draw a vertical dotted line at $x = 0$.
3. Add a legend indicating what the different lines are.

Here's an excellent article (although slightly advanced) that gives a quick intro to Matplotlib: https://pbpython.com/effective-matplotlib.html (https://pbpython.com/effective-matplotlib.html). Some sections use Pandas, which is another Python library used by the data science community. You don't need to understand how it works, but just know that in Pandas, you can visualize data by calling a plot method on the data object (known as a dataframe in Pandas vernacular). This effectively calls matplotlib in the background.

Matplotlib documentation: https://matplotlib.org/3.3.3/contents.html (https://matplotlib.org/3.3.3/contents.html).

In [11]:
```
# TODO:
# Your logic goes here
x = np.linspace(-10, 10, 101)
x1 = np.exp(x)
x2 = np.exp(-x)
f_x = (x1-x2)/(x1+x2)
ax.plot(x,f_x, 'b')
ax.axvline(x = 0, color ='r', linestyle = 'dashed')
ax.legend(['$\sigma(x)$','$tanh(x)$'])
fig
```

Out[11]:



Now let's try to display a group of images using Matplotlib. We have provided some sample images from Imagenet dataset in the `img` folder. This dataset is mainly used to train neural nets for object recognition task. For this, we will need a library called `PIL` (Python Imaging Library).

In [12]:
```
from PIL import Image # Reference: https://pillow.readthedocs.io/en/3.0.x/reference/Image.html
import glob     # Reference: https://docs.python.org/3/library/glob.html
img_paths = glob.glob('img/**.jpg')
n_images = len(img_paths)
```

In [13]:
```
img_paths
```

Out[13]:
```
['img/n02206856_2865_bee.jpg',
 'img/n02121808_1421_domestic_cat.jpg',
 'img/n02118333_27_fox.jpg',
 'img/n02802426_12131_basketball.jpg',
 'img/n07749582_16107_lemon.jpg',
 'img/n02324045_3738_rabbit.jpg',
 'img/n02509815_27612_red_panda.jpg',
 'img/n02787622_3030_banjo.jpg']
```

In [14]:
```
fig, ax = plt.subplots(n_images//2, 2, figsize=(10, 10))
fig.tight_layout()

for i in range(n_images):
    img_path = img_paths[i]
    img_data = Image.open(img_path)
    row, col = i//2, i%2
    label = img_path.split('_')[-1].split('.')[0]
    ax[row][col].imshow(img_data)
    ax[row][col].set_title(label)
    ax[row][col].axis('off')
```

Part 3: ResNet

In this course, we will use PyTorch to build and train neural networks. Although PyTorch will be introduced in a later assignment, we want to show here how you can see important model details.

PyTorch folks also maintain a separate module called `torchvision` (https://pytorch.org/vision/stable/index.html), which contains popular datasets, pretrained models relevant to computer vision. We will use ResNet-18 model from this module to demonstrate this. You don't need to know any details about ResNet, this is just for demonstration purposes. Here's a link to the paper in case you're interested: https://arxiv.org/pdf/1512.03385.pdf (https://arxiv.org/pdf/1512.03385.pdf)

There is a library called `torchsummary` (https://github.com/sksq96/pytorch-summary) that can be used to print model summary (this was created by an NYU student!). With this, we can see all the layers inside the model, the type of each layer, etc. In addition, it also prints overall statistics such as model size, number of parameters in the model, etc.

In [15]:
```
from torchvision import models
from torchsummary import summary

resnet18 = models.resnet18().cpu()

# Summary method requires the input image size. ResNet has been trained
# on image size (3, 224, 224)
print(summary(resnet18, (3, 224, 224)))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 112, 112]           9,408
       BatchNorm2d-2         [-1, 64, 112, 112]             128
              ReLU-3         [-1, 64, 112, 112]               0
         MaxPool2d-4           [-1, 64, 56, 56]               0
            Conv2d-5           [-1, 64, 56, 56]          36,864
       BatchNorm2d-6           [-1, 64, 56, 56]             128
              ReLU-7           [-1, 64, 56, 56]               0
            Conv2d-8           [-1, 64, 56, 56]          36,864
       BatchNorm2d-9           [-1, 64, 56, 56]             128
             ReLU-10           [-1, 64, 56, 56]               0
       BasicBlock-11           [-1, 64, 56, 56]               0
           Conv2d-12           [-1, 64, 56, 56]          36,864
      BatchNorm2d-13           [-1, 64, 56, 56]             128
             ReLU-14           [-1, 64, 56, 56]               0
           Conv2d-15           [-1, 64, 56, 56]          36,864
      BatchNorm2d-16           [-1, 64, 56, 56]             128
             ReLU-17           [-1, 64, 56, 56]               0
       BasicBlock-18           [-1, 64, 56, 56]               0
           Conv2d-19          [-1, 128, 28, 28]          73,728
      BatchNorm2d-20          [-1, 128, 28, 28]             256
             ReLU-21          [-1, 128, 28, 28]               0
           Conv2d-22          [-1, 128, 28, 28]         147,456
      BatchNorm2d-23          [-1, 128, 28, 28]             256
           Conv2d-24          [-1, 128, 28, 28]           8,192
      BatchNorm2d-25          [-1, 128, 28, 28]             256
             ReLU-26          [-1, 128, 28, 28]               0
       BasicBlock-27          [-1, 128, 28, 28]               0
           Conv2d-28          [-1, 128, 28, 28]         147,456
      BatchNorm2d-29          [-1, 128, 28, 28]             256
             ReLU-30          [-1, 128, 28, 28]               0
           Conv2d-31          [-1, 128, 28, 28]         147,456
      BatchNorm2d-32          [-1, 128, 28, 28]             256
             ReLU-33          [-1, 128, 28, 28]               0
       BasicBlock-34          [-1, 128, 28, 28]               0
           Conv2d-35          [-1, 256, 14, 14]         294,912
      BatchNorm2d-36          [-1, 256, 14, 14]             512
             ReLU-37          [-1, 256, 14, 14]               0
           Conv2d-38          [-1, 256, 14, 14]         589,824
      BatchNorm2d-39          [-1, 256, 14, 14]             512
           Conv2d-40          [-1, 256, 14, 14]          32,768
      BatchNorm2d-41          [-1, 256, 14, 14]             512
             ReLU-42          [-1, 256, 14, 14]               0
       BasicBlock-43          [-1, 256, 14, 14]               0
           Conv2d-44          [-1, 256, 14, 14]         589,824
      BatchNorm2d-45          [-1, 256, 14, 14]             512
             ReLU-46          [-1, 256, 14, 14]               0
           Conv2d-47          [-1, 256, 14, 14]         589,824
      BatchNorm2d-48          [-1, 256, 14, 14]             512
             ReLU-49          [-1, 256, 14, 14]               0
       BasicBlock-50          [-1, 256, 14, 14]               0
           Conv2d-51            [-1, 512, 7, 7]       1,179,648
      BatchNorm2d-52            [-1, 512, 7, 7]           1,024
```

```
              ReLU-53            [-1, 512, 7, 7]               0
            Conv2d-54            [-1, 512, 7, 7]       2,359,296
       BatchNorm2d-55            [-1, 512, 7, 7]           1,024
            Conv2d-56            [-1, 512, 7, 7]         131,072
       BatchNorm2d-57            [-1, 512, 7, 7]           1,024
              ReLU-58            [-1, 512, 7, 7]               0
        BasicBlock-59            [-1, 512, 7, 7]               0
            Conv2d-60            [-1, 512, 7, 7]       2,359,296
       BatchNorm2d-61            [-1, 512, 7, 7]           1,024
              ReLU-62            [-1, 512, 7, 7]               0
            Conv2d-63            [-1, 512, 7, 7]       2,359,296
       BatchNorm2d-64            [-1, 512, 7, 7]           1,024
              ReLU-65            [-1, 512, 7, 7]               0
        BasicBlock-66            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-67            [-1, 512, 1, 1]               0
            Linear-68                 [-1, 1000]         513,000
================================================================
Total params: 11,689,512
Trainable params: 11,689,512
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 62.79
Params size (MB): 44.59
Estimated Total Size (MB): 107.96
----------------------------------------------------------------
None
```

Question 4

There are other ResNet architectures - ResNet-34, ResNet-50, ResNet-101 and ResNet-152. These models mainly differ in the number of layers. They are available in the torchvision module. Using the example code above, find the model size (i.e. params size) and number of parameters for each of the **first three** models. Provide your answers in a tabular format.

Use the input image size (3, 224, 224).

In [16]:
```
d = [ ["ResNet-34", 83.15, 21797672],
      ["ResNet-50", 97.49, 25557032],
      ["ResNet-101", 169.94, 44549160]]

print ("{:<20} {:<30} {:<30}".format('Model Name','Model Size (params size)(MB)','Number of Parameters'))

for v in d:
    name, size, num = v
    print ("{:<20} {:<30} {:<30}".format( name, size, num))
```

```
Model Name           Model Size (params size)(MB)   Number of Parameters
ResNet-34            83.15                          21797672
ResNet-50            97.49                          25557032
ResNet-101           169.94                         44549160
```

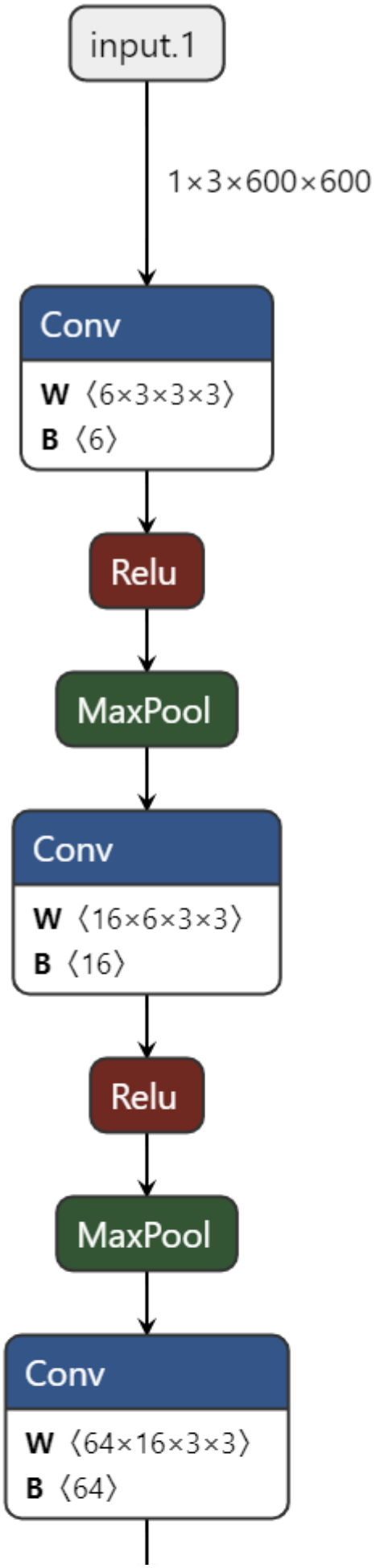
Part 2

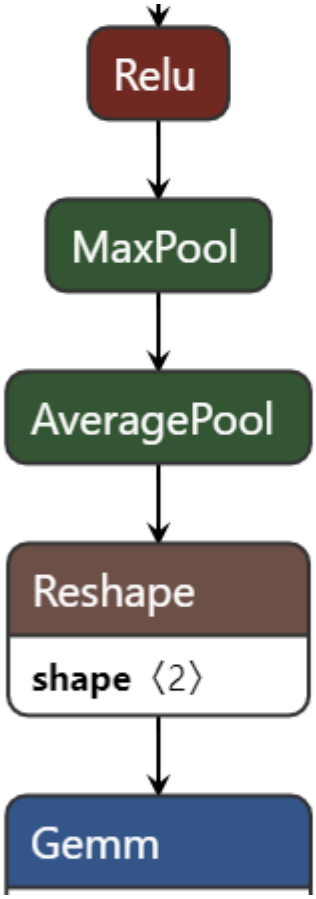
Question 1:

In [17]:
```
%reset -f
import torch
from torch.nn import Linear, ReLU, Conv2d, MaxPool2d, Module, Sequential, Adap
tiveAvgPool2d
from torchvision import models
from torchsummary import summary
import torch.onnx

class my_CNN(Module):
    def __init__(self):
        super(my_CNN, self).__init__()
        self.features = Sequential(
            # Layer 1
            Conv2d(in_channels=3, out_channels=6, kernel_size=3),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=3, stride=2),

            # Layer 2
            Conv2d(in_channels=6, out_channels=16, kernel_size=3),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=3, stride=2),

            # Layer 3
            Conv2d(in_channels=16, out_channels=64, kernel_size=3),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=3, stride=2)
        )

        # Average Pooling Layer
        self.avgpool = AdaptiveAvgPool2d((6, 6))

        # Linear Layer / Fully connected layer / Dense Layer
        self.Linear_Layers = Sequential(
            # Linear Layer 1 with ReLU
            Linear(in_features=64*6*6, out_features=100),
            ReLU(inplace=True),

            # Linear Layer 2 with ReLU
            Linear(in_features=100, out_features=10),
            ReLU(inplace=True)
        )

    def forward(self, input_image: torch.Tensor) -> torch.Tensor:
        input_image = self.features(input_image)
        input_image = self.avgpool(input_image)
        input_image = input_image.view(-1,64*6*6)
        input_image = self.Linear_Layers(input_image)
        return input_image


my_torch_model = my_CNN()
inputs = torch.randn(1, 3, 600, 600, requires_grad=True)
torch_out = my_torch_model(inputs)
summary(my_torch_model,(3, 600, 600))
```

```
torch.onnx.export(my_torch_model,inputs,"My_CNN_model.onnx", export_params=True, opset_version=10)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 6, 598, 598]             168
              ReLU-2          [-1, 6, 598, 598]               0
         MaxPool2d-3          [-1, 6, 298, 298]               0
            Conv2d-4         [-1, 16, 296, 296]             880
              ReLU-5         [-1, 16, 296, 296]               0
         MaxPool2d-6         [-1, 16, 147, 147]               0
            Conv2d-7         [-1, 64, 145, 145]           9,280
              ReLU-8         [-1, 64, 145, 145]               0
         MaxPool2d-9           [-1, 64, 72, 72]               0
AdaptiveAvgPool2d-10             [-1, 64, 6, 6]               0
           Linear-11                  [-1, 100]         230,500
             ReLU-12                  [-1, 100]               0
           Linear-13                   [-1, 10]           1,010
             ReLU-14                   [-1, 10]               0
================================================================
Total params: 241,838
Trainable params: 241,838
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 4.12
Forward/backward pass size (MB): 83.92
Params size (MB): 0.92
Estimated Total Size (MB): 88.96
----------------------------------------------------------------
```


input.1
1×3×600×600
Conv
W ⟨6×3×3×3⟩
B ⟨6⟩
Relu
MaxPool
Conv
W ⟨16×6×3×3⟩
B ⟨16⟩
Relu
MaxPool
Conv
W ⟨64×16×3×3⟩
B ⟨64⟩



Calculation of the number of learnable parameters

Layer-1 Input=3, Output=6, kernel=3 result = 9 params/input *3 inputs = 27 params/filter* 6 filters = 162 params + (6 filters * 1 bias/filter) = 162 + 6 + 0 (for Relu and maxpooling) = 168 parameters

Layer-2 Input=6, Output=16, kernel=3 result = 9 params/input *6 inputs = 54 params/filter* 16 filters = 864 params + (6 filters * 1 bias/filter) = 864 + 16 + 0 (for Relu and maxpooling) = 880 parameters

Layer-3 Input=16, Output=64, kernel=3 result = 9 params/input *16 inputs = 144 params/filter* 64 filters = 9216 params + (64 filters * 1 bias/filter) = 9216 + 64 + 0 (for Relu and maxpooling) = 9280 parameters

Linear Layer-1 Input=6466, Output=100 result = 6466 params/output_channel *100 output_channels = 230400 params + (100 channels* 1 bias/filter) = 230500 parameters

Linear Layer-2 Input=100, Output=10 result = 100 params/output_channel *10 output_channels = 1000 params + (10 channels* 1 bias/filter) = 1010 parameters

Total params = 168 + 880 + 9280 + 230500 + 1010 = 241838 parameters

Question 2:

In [18]:
```
%reset -f
import torch
import torch.nn as nn
import torch.nn.functional as F

def convolution(image_mat, kernel_mat, dilation):
    i_rows, i_cols =len(image_mat), len(image_mat[0])
    k_rows, k_cols = len(kernel_mat), len(kernel_mat[0])
    dil = dilation #Dilation
    output_rows = i_rows-(dil*(k_rows-1))
    output_cols = i_cols-(dil*(k_cols-1))
    #print(i_rows,i_cols, k_rows, k_cols,output_rows,output_cols)

    convoluted_mat=[]
    for i in range(output_rows):
        temp = []
        for j in range(output_cols):
            res = 0
            for k in range(k_rows):
                for l in range(k_cols):
                    res += image_mat[k+i][l+j]*kernel_mat[k][l]
            temp.append(res)
        convoluted_mat.append(temp)
    return (convoluted_mat)

# Using PyTorch

def using_pytorch(image_mat, kernel_mat, dil):
    image_2D = torch.tensor(image_mat).unsqueeze(0).unsqueeze(0)
    kernel = torch.tensor(kernel_mat).unsqueeze(0).unsqueeze(0)
    return (F.conv2d(image_2D, kernel, dilation = dil))

# Function call
image_mat = [[1,2,3,4], [5,6,7,8], [9,0,1,2], [3,4,5,6]]
kernel = [[1,2],[3,4]]
dilation = 1

print("The convoluted matrix with dialation is: ", convolution(image_mat, kern
el, dilation))
print()
print("The convoluted matrix using pytorch is: ")
print(using_pytorch(image_mat, kernel, dilation))
```

```
The convoluted matrix with dialation is:  [[44, 54, 64], [44, 24, 34], [34, 3
4, 44]]

The convoluted matrix using pytorch is:
tensor([[[[44, 54, 64],
          [44, 24, 34],
          [34, 34, 44]]]])
```

Question 3 (General form using dilation, will work for both Qs 2 and 3)

In [19]:
```
%reset -f
import torch
import torch.nn as nn
import torch.nn.functional as F

def convolution(image_mat, kernel_mat, dil):
    i_rows, i_cols =len(image_mat), len(image_mat[0])
    k_rows, k_cols = len(kernel_mat), len(kernel_mat[0])

    # Finding out the dilated kernel matrix
    kernel_mat_dil = k_mat_dil(dil,k_rows,k_cols,kernel_mat)

    i_rows, i_cols =len(image_mat), len(image_mat[0])
    k_rows, k_cols = len(kernel_mat_dil), len(kernel_mat_dil[0])

    output_rows = i_rows-k_rows+1
    output_cols = i_cols-k_cols+1

    convoluted_mat=[]
    for i in range(output_rows):
        temp = []
        for j in range(output_cols):
            res = 0
            for k in range(k_rows):
                for l in range(k_cols):
                    res += image_mat[k+i][l+j]*kernel_mat_dil[k][l]
            temp.append(res)
        convoluted_mat.append(temp)
    return convoluted_mat


# Function computing the dilated kernel matrix
def k_mat_dil(dil,k_rows,k_cols,kernel_mat):
    if(dil>1):
        dilated_mat_rows = k_rows+(dil-1)*(k_rows-1)
        dilated_mat_cols = k_cols+(dil-1)*(k_cols-1)
        for i in range(0,dilated_mat_rows,dil):
            for j in range(1,dilated_mat_cols-1,dil-1):
                for k in range(dil-1):
                    kernel_mat[i].insert(j,0)
            if (i+1)<dilated_mat_rows:
                for k in range(dil-1):
                    kernel_mat.insert((i+1),([0]*dilated_mat_cols))
    return kernel_mat


# Using PyTorch

def using_pytorch(image_mat, kernel_mat, dil):
    image_2D = torch.tensor(image_mat) .unsqueeze(0).unsqueeze(0)
    kernel_mat = torch.tensor(kernel_mat) .unsqueeze(0).unsqueeze(0)
    return (F.conv2d(image_2D, kernel_mat, dilation = dil))

# Function call
image_mat = [[1,2,3,4], [5,6,7,8], [9,0,1,2], [3,4,5,6]]
```

```
kernel = [[1,2],[3,4]]
dilation = 2

print("The convoluted matrix using pytorch is: ")
print(using_pytorch(image_mat, kernel, dilation))
print()
print("The convoluted matrix with dialation is: ", convolution(image_mat, kern
el, dilation))
```

```
The convoluted matrix using pytorch is:
tensor([[[[38, 18],
          [48, 58]]]])

The convoluted matrix with dialation is:  [[38, 18], [48, 58]]
```

In [ ]: