Lusine Kamikyan
Homework 1, BME 595
Report
Collaborated with Emma Reid

Code and their explanation is at the end of the report. I used the following link's definition of the convolution to implement the problems: http://cs231n.github.io/convolutional-networks/
I assumed bias = 0 and there is no padding

Part A.
In part A, we do each task for both of the images.
The Task1 return the grayscale image with horizontal lines for K1 filter, Task2 returns the horizontal and vertical lines, for filters K4 and K5, respectively. Task3 returns greyscale images - horizontal lines, vertical lines and blurred (it was hard to see what K3 does, looks a bit blurred). The original images and result images are in the submitted homework folder.

To run PartA in the code, please comment the PartB, and PartC section of the main.py file, and remove the comments from the PartA, and then run.

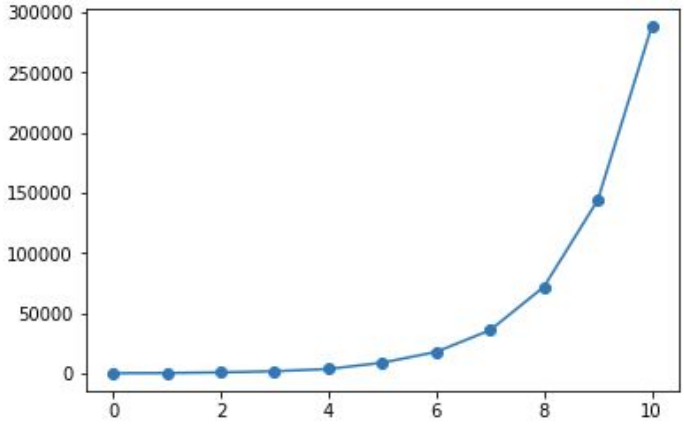
My computer is pretty old and it was taking 1-2 or more hours to run Part B, C and D for larger values of k_size and index i.

Part B.
To run PartB in the code, please comment the PartA, and PartC section of the main.py file, and remove the comments from the PartB, and then run.

The plot and values for i vs time for image of size 1280 by 720:
After i=5, I had to stop, since it was taking forever. But looks like the values are about double of the previous.

i=0	260.14964509
i=1	457.87425303
i=2	979.40748501
i=3	1931.64215589
i=4	3804.61950111
i=5	9000.21378112

i=6	18000
i=7	36000
i=8	72000
i=9	144000
i=10	288000

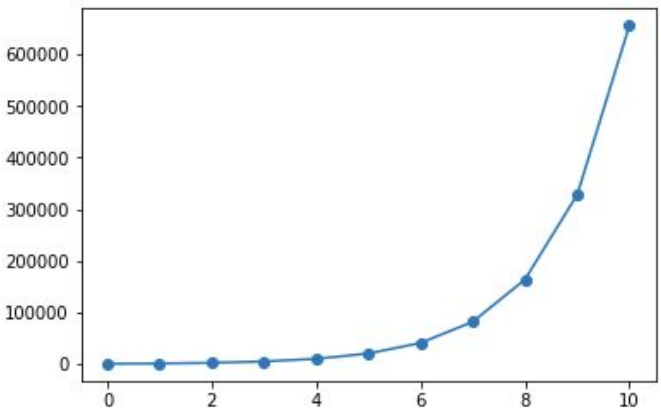


Plot1: i vs time for image of size 1280 by 720

The plot and values for i vs number of operations for image of size 1080 by 1920:
After i=3, I had to stop, since it was taking forever. But looks like the values are about double of the previous.

i=0	579.216
i=1	1186.745749
i=2	2439.28260994
i=3	5125.12890911
i=4	10250
i=5	20500
i=6	41000
i=7	82000

i=8	164000
i=9	328000
i=10	656000



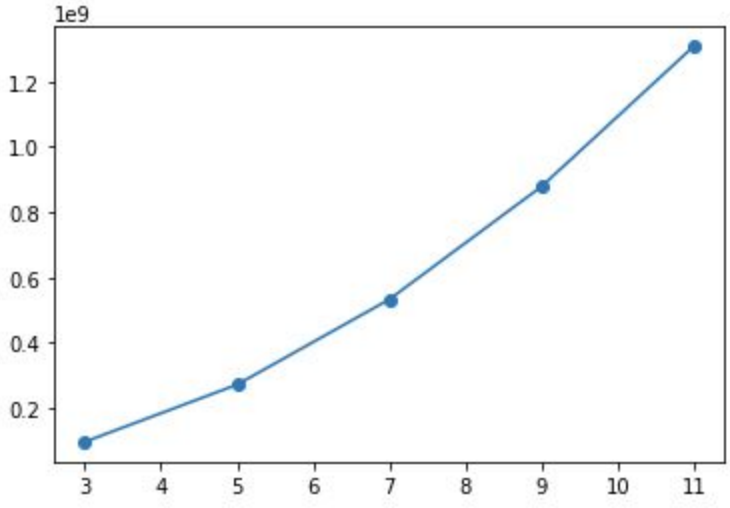
Plot2: i vs time for image of size 1920 by 1080

Part C:

To run PartC in the code, please comment the PartA, and PartV section of the main.py file, and remove the comments from the PartC, and then run.

<u>The plot (k_size vs number of operations) and values of number of operations for 1280 by 720:</u>

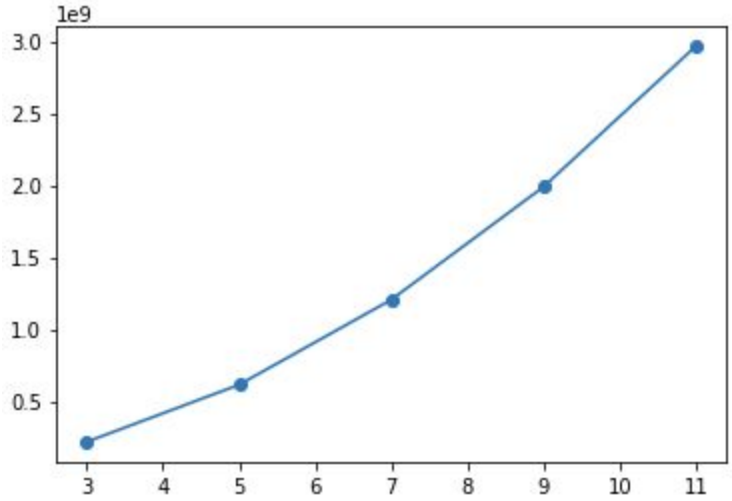
K_size = 3	97,266,024
K_size = 5	272,257,568
K_size = 7	533,046,696
K_size = 9	878,494,080
K_size =11	1,307,465,000



Plot3: k_size vs number of operations for image 1280x720

The plot (k_size vs number of operations) and values of number of operations for 1920 by1080:
It was taking very long, so we use the formula (len of output_image)*(width of output image)*(2*in_channel*k_size^2-1)*o_channel formula to find the number of operations in case of kernel size = 11

K_size = 3	218,166,024
K_size = 5	614,361,568
K_size = 7	1,204,602,700
K_size = 9	1,988,174,080
K_size =11	2,963,365,000

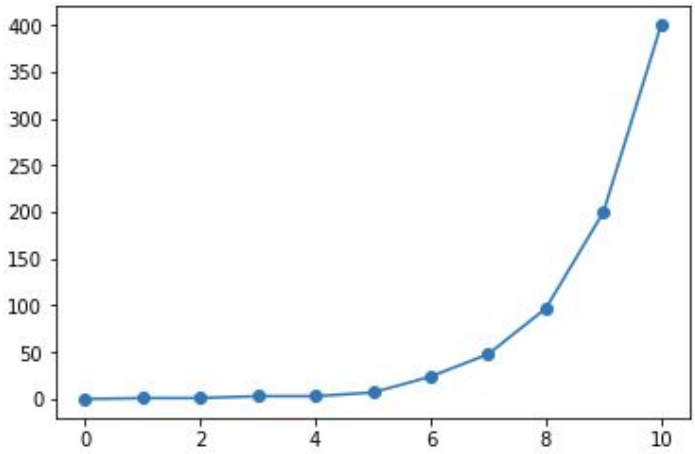


Plot4: k_size vs number of operations for image 1920x1080

Part D:
Used the clock() function to find the time

<u>The plot and values for i vs time for image of size 1280 by 720:</u>
After i = 8, it was taking a long time to run so I had to stop. Looks like when i gets bigger, it about doubles

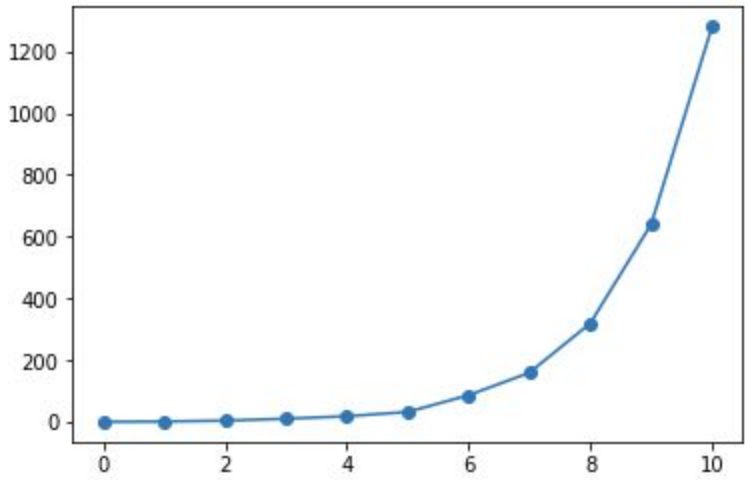
i=0	0
i=1	1
i=2	1
i=3	3
i=4	3
i=5	7
i=6	24
i=7	48
i=8	97
i=9	200
i=10	400



Plot5: i vs time for image of size 1280 by 720

The plot and values for i vs time for image of size 1920 by 1080:
After i = 7, it was taking long time to run

i=0	1
i=1	2
i=2	5
i=3	11
i=4	19
i=5	33
i=6	87
i=7	160
i=8	320
i=9	640
i=10	1280



Plot6: i vs time for image of size 1920 by 1080

Explanation of the code:

Code for all parts is copied at the end of the following report and the corresponding files are put into the folder.

The python code consists of the main.py and the conv.py which contains the Conv2D class. My Conv2D class takes in the *input channel, output channel, the kernel size, the stride and the mode*. Conv2D class contains the *forward()* function where the main part of the convolution occurs, it takes the 3D tensor of the input image as input and outputs the number of the operations and the filtered output image. I save the output_image as greyscale image. I also have a kern() function inside my Class2D, which based on the o_channel, kernel size and mode decides what the kernel(s) are. If there are more than one kernels, it stacks them together using torch.stack. All the tensors, the image and the kernels are made such that they are float tensors. I am assuming there is no padding and bias = 0.
forward() function contains 4 nested for loops to do the convolution operation:
-it loops through columns of the input image by the amount of stride until *#col-k_size+1*. It can't pass this number otherwise it will end up outside of the image, where #col is the number of columns of input image
- Insides the above loop, there is the second loop going through the rows of the input image by amount of *stride* until *#rows - k_size+1 (reason similar to above)*
- The 3rd nested loop loops through the layers of the kernel (layers of kernel = o_channel)
- And the final, 4th loop index goes through the layers of the input image.
For example, we have 1 kernel of size 3x3.It slides over the input image by the amount of *stride*, and the 3x3 parts of the input image get elementwise multiplied by the kernel, we do that in all the layers of the picture,then the results get added, which then gets assigned to the

corresponding position in the output_image tensor. We keep doing this until the kernel passes over all of the input image. If we have more kernels, the same process happens for each. The output image will have same amount of channels as the number of kernels.

The number of operations is calculated the following way: if the kernel size k_size with number of output channels o_channels, then when the kernel is multiplied with the n by n part of the image (just one layer), we have *k_size^2 multiplications and k_size^2-1 additions,* we do this for for each layer of the image (making sure we are not overcounting).

I plot the requested plots in part B and part C.

Part D does the same as part B. My C code consists of the main{} function and the c_conv() function. I loop through values of i = 0 to 10 in the main function and in each loop call the c_conv function which has in_channel, o_channel, k_size and stride as input and returns the time taken to run the convolution (including memory allocation part using malloc).
I define the random value kernel and input image inside c_conv, using malloc. I also define the output_image using malloc.
In the main part of the c_conv, I have 6 nested loops:

- First one going through the layers of the kernel
- Second and third one going through the rows and columns of the image, respectively
- Then I pick which layer of the image I will be using to do the “dot product” between the kernel and the part of the image
- Then the next 2 loops go through the kernel len and width, and I do element-wise multiplication, and add the corresponding results, and save it in the output_image entry

The clock for time runs before the dynamic memory allocation, and stops before freeing the memory.

MyCode:

main.py

```
from conv import Conv2D
import torch
import cv2
import matplotlib.pyplot as plt
import scipy.misc
import time
import numpy as np

img = cv2.imread('pic1.jpg') #change to pic2 to run for teh second picture
```

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#plt.imshow(img)

img = img.astype(float)
tensor_img = torch.from_numpy(img)
tensor_img = tensor_img.type(torch.FloatTensor)


##### PART A ##### uncomment if you want to run, comment partB and partC
conv2d = Conv2D(3,1,3,1,'known')
[num_of_operation, output_image] = conv2d.forward(tensor_img)

for i in range(0,output_image.size(2)):
        result = output_image[:,:,i].numpy()
        scipy.misc.imsave("Pic11111_Task_1_"+str(i+1)+".jpg", result)

##### PART B ##### uncomment if you want to run, comment partA and partC
'''TotalTime = np.zeros(shape = (11,1))
I = np.zeros(shape = (11,1))
for i in range(0,10):
        start_time = time.time()
        conv2d = Conv2D(3,2**i,3,1,'known')
        [num_of_operation, output_image] = conv2d.forward(tensor_img)
        end_time = time.time()
        TotalTime[i] = end_time-start_time
        print(TotalTime[i])
        #print(TotalTime)
        I[i] = i

plt.plot(I,TotalTime, 'o')'''

##### PART C ##### uncomment if you want to run, comment partA and partB
'''I = np.zeros(shape = (5,1))
N_OPER = np.zeros(shape=(5,1))
for i in range(2,5):
        conv2d = Conv2D(3,2,2*i+3,1,'rand')
        [num_of_operation, output_image] = conv2d.forward(tensor_img)
        N_OPER[i]= num_of_operation
        print(N_OPER[i])
        I[i] = 2*i+3

#print(N_OPER)
plt.plot(I, N_OPER, '-o')'''
```

conv.py

```
import torch
import numpy as np
import random as rand

class Conv2D:
    def __init__(self, in_channel, o_channel, kernel_size, stride, mode):
        # in_channel = depth of image, RGB
        # o_channel = number of kernels, the same as output image depth
        # kernel_size = width of kernel
        # stride = by how much we move the kernel vertically and horizontally
        # mode = random vs known kernel
        self.in_channel = in_channel
        self.o_channel = o_channel
        self.k_size = kernel_size
        self.stride = stride
        self.mode = mode


    def forward(self, input_image):

        [row, col,hgt] = input_image.size()

        K = Conv2D.kern(self)
        K = K.type(torch.Tensor)

        output_image = torch.zeros((row-self.k_size)//self.stride+1,(col-self.k_size)//self.stride+1,
self.o_channel)

        count = 0
        for j in range(0,col-self.k_size+1,self.stride): #looping through columns
            for i in range(0,row-self.k_size+1, self.stride): #looping through rows
                for l in range(0,K.size(0)): #layers of the kernel
                    sum_in_depth = 0
                    for k in range(0,hgt): #layers of the picture
                        dot_pr = torch.sum(input_image[i:i+self.k_size,j:j+self.k_size,k]*K[l,:,:])
                        sum_in_depth+=dot_pr
                        # counting the operations
                        count += 2*self.k_size**2
```

```
                count = count-1

                # saving the result in the corresponging entry of output image
                output_image[i//self.stride,j//self.stride,l] = sum_in_depth


        return count, output_image
        #return [int, 3D FloatTensor]

    ## function kern() return the kernel(s) needed to be used
    def kern(self):
        if self.mode == 'known':
            if self.o_channel == 1:
                k1 = np.array([[-1., -1., -1.],[0., 0., 0.],[1., 1., 1.]])
                k1_tens = torch.from_numpy(k1)
                tensor_list = [k1_tens]
                kernel = torch.stack(tensor_list)
            elif self.o_channel == 2:
                k4 = np.array([[-1., -1., -1., -1., -1.],[-1., -1., -1., -1., -1.],[0., 0., 0., 0., 0.],[ 1., 1., 1., 1.,
1.],[1., 1., 1., 1., 1.]])
                k5 = np.array([[-1., -1., 0., 1., 1.],[-1., -1., 0., 1., 1.],[-1., -1., 0., 1., 1.],[ -1., -1., 0., 1.,
1.],[-1., -1., 0., 1., 1.]])
                # convert to tensor
                k4_tens = torch.from_numpy(k4)
                k5_tens = torch.from_numpy(k5)
                # stack the tensors to get width of o_channel tensor
                tensor_list = [k4_tens, k5_tens]
                kernel = torch.stack(tensor_list)
            elif self.o_channel == 3:
                k1 = np.array([[-1., -1., -1.],[0., 0., 0.],[1., 1., 1.]])
                k2 = np.array([[-1.,  0.,  1.], [-1., 0., 1.], [-1., 0., 1.]])
                k3 = np.array([[ 1.,  1.,  1.],[1., 1., 1.], [1., 1., 1.]])
                k1_tens = torch.from_numpy(k1)
                k2_tens = torch.from_numpy(k2)
                k3_tens = torch.from_numpy(k3)
                tensor_list = [k1_tens, k2_tens, k3_tens]
                kernel = torch.stack(tensor_list)
        else:
            kernel = torch.zeros(self.o_channel,self.k_size,self.k_size, dtype = torch.float)
            for j in range(0,self.o_channel):
                kernel[j,:,:] = torch.rand(self.k_size,self.k_size, dtype = torch.float)

        return kernel
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int c_conv(int in_channel, int o_channel, int k_size, int stride);

int main(void) {

	int o_channel;
	int i;
	int total_time;
	int in_channel = 3;
	int k_size = 3;
	int stride = 1;



	for (i =0; i<10; i++) {
	o_channel = (int) pow(2,i);
	total_time = c_conv(in_channel,o_channel, k_size, stride);
	}

	return 0;
}
int c_conv(int in_channel, int o_channel, int k_size, int stride){
	int i,j,k,l,m,n;
	double pr;
	int count;

	time_t start_t, end_t, total_t;
	start_t = clock();

	/* create the random kernel*/
	double ***kernel;
	kernel = (double ***)malloc((k_size)*sizeof(double **));
	for (i =0;i<k_size; i++){
	kernel[i] = (double **)malloc((k_size)*sizeof(double *));
```

```
for (j=0;j<k_size;j++){
kernel[i][j] = (double *)malloc(sizeof(double *) * o_channel);
for (k=0;k<o_channel; k++){
        kernel[i][j][k] = rand();
}
}
}


/* image array*/
int row = 720;//1080;
int col = 1280;//1920;
int hgt = 3;

double ***image;
image = (double ***)malloc(row*sizeof(double **));

for (i = 0 ;  i < row; i++) {
image[i] = (double **)malloc(col*sizeof(double *));

for (j = 0; j < col; j++) {
image[i][j] = (double *)malloc(hgt*sizeof(double));

for (k = 0; k < hgt; k++){
image[i][j][k] = rand() % 255 + 0;
}
}
}



double ***output_image = (double ***)malloc(row*sizeof(double **));

for (i = 0 ;  i < row-k_size+1; i++) {
output_image[i] = (double **)malloc(col*sizeof(double *));

for (j = 0; j < col-k_size+1; j++) {
output_image[i][j] = (double *)malloc(o_channel*sizeof(double));
}
}


/* put the convolution in here */
```

```
count = 0;
for (l=0; l<o_channel;l++){                //through the layers of the kernel
for (i=0;i<row-k_size+1;i=i+stride){    // through the rows of the picture
for (j=0;j<col-k_size+1;j=j+stride){ // through the columns of the picture
        double temp_sum =0;
        for (k=0;k<hgt;k++){              // through the layers of the picture
        for (m=0;m<k_size;m++){       //through rows of the kernel
        for (n=0;n<k_size;n++){  // through columns of the kernel
                pr = image[i+m][j+n][k]*kernel[m][n][l];
                temp_sum += pr;
                count += 2;
        }
        }
        }
        output_image[i/stride][j/stride][l] = temp_sum;
}
}
}
end_t = clock();
total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;

for (i=0;i<row;i++){
for(j=0;j<col;j++){
free(image[i][j]);
}
free(image[i]);
}
free(image);


for (i=0;i<k_size;i++){
for(j=0;j<k_size;j++){
free(kernel[i][j]);
}
free(kernel[i]);
}
free(kernel);

for (i=0;i<row-k_size+1;i++){
for (j=0; j<col-k_size+1;j++){
free(output_image[i][j]);
}
```

```
        free(output_image[i]);
        }
        free(output_image);


        return total_t;
}
```