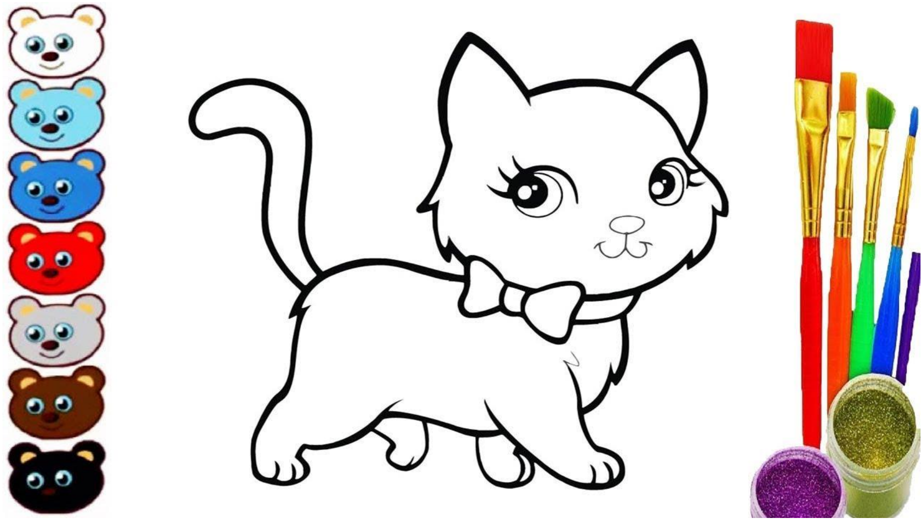
1. The two input images are:

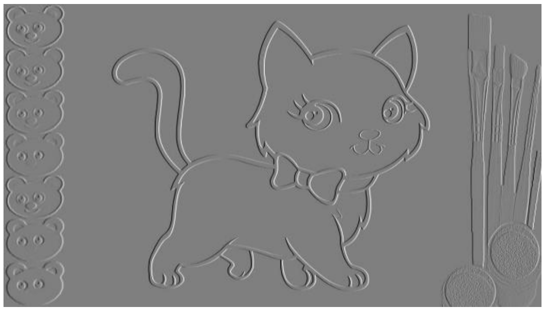
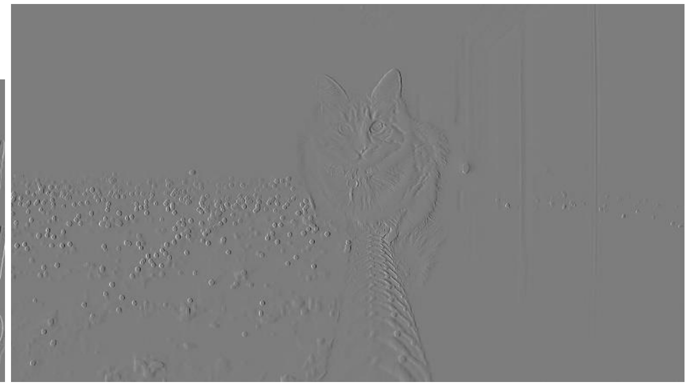
(1) 1280x720.jpg



(2) 1920x1080.jpg

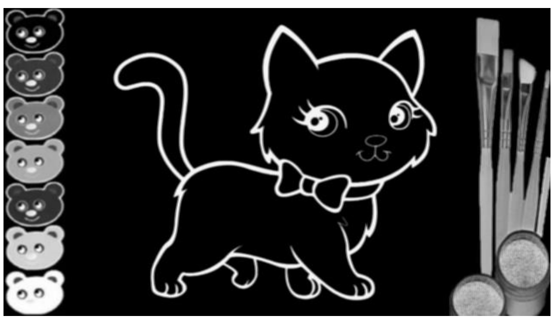
2. Part A.

(1) Results of task 1.

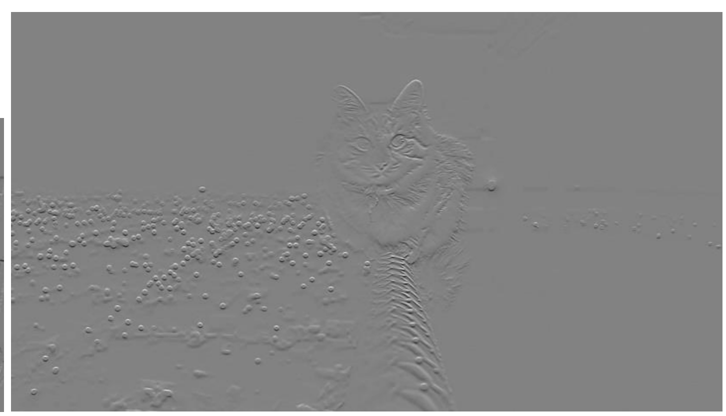



(2) Results of task 2

K4

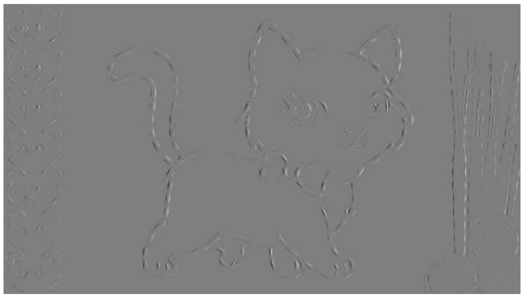
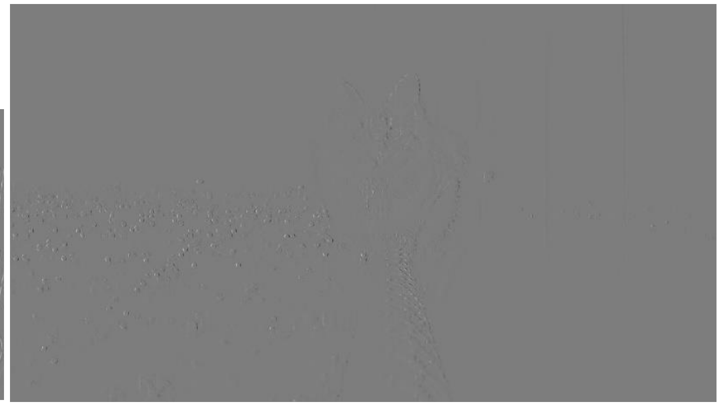



K5

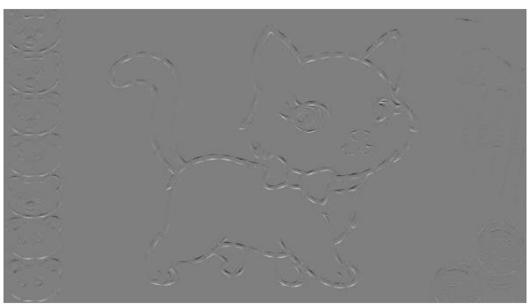
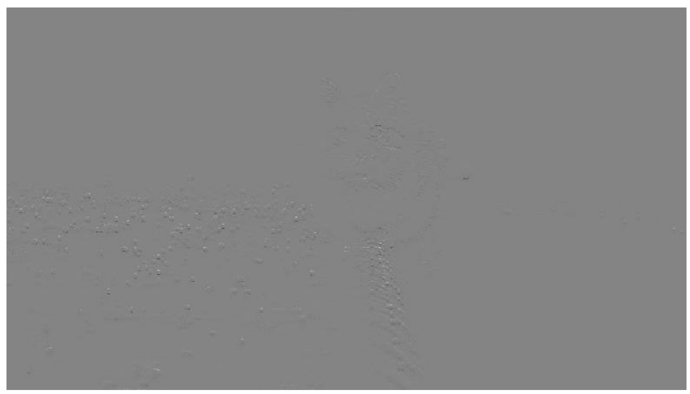



(3) Results of task 3

K1

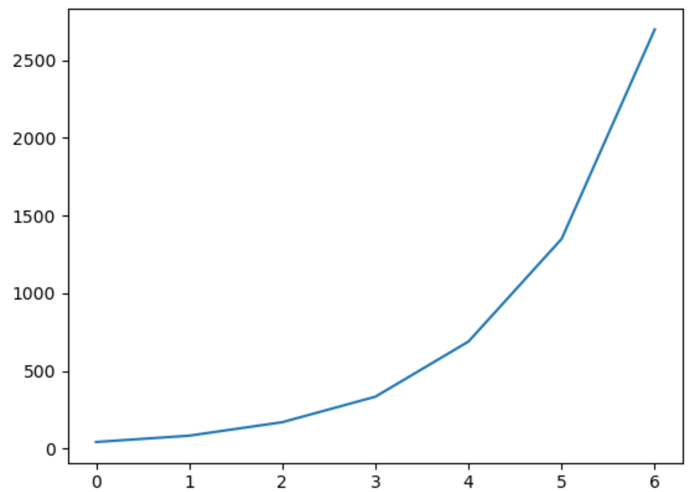
K2

K3


3. Part B

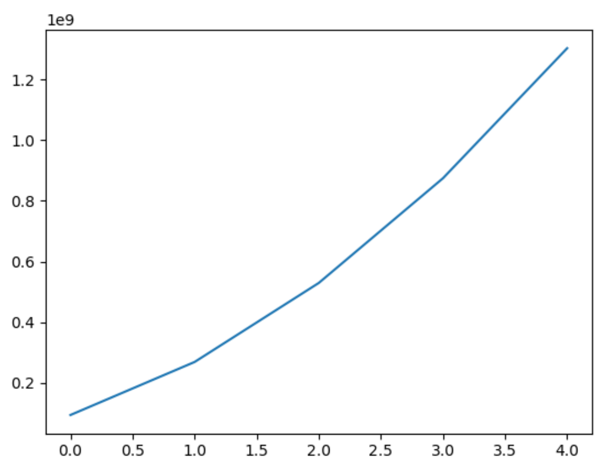
The time taken for performing each *forward()* as a function of *i* is shown as



Here, *x-axis* represents *i, y-axis* represents the time taken (in seconds). Here only shows i <= 6 because *i=7* takes more than 1 hour to process (CPU only). We can see that with the increase of *i*, the time taken has improved a lot.

4. Part C

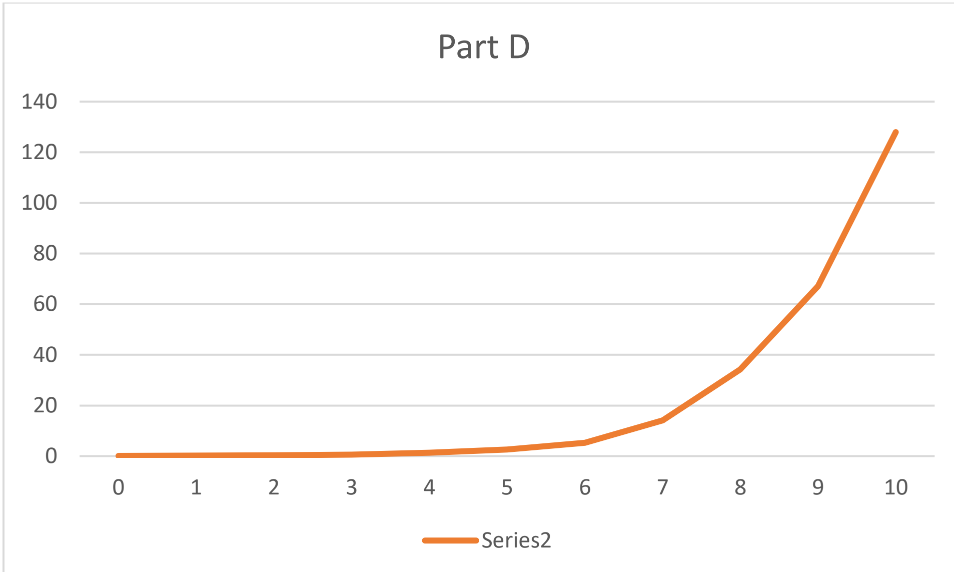
The number of operations used to perform convolution as a function of kernel_size is



Here, *x-axis* should be 3, 5, …, 11, representing *kernel_size, y-axis* represents the number of operations. We can see that with the increase of *i*, the time taken has improved a lot. In the code, each convolution operation has *kernel_size*kernel_size* of multiplications, and *kernel_size*kernel_size-1* of additions.

5. Part D

Using C code, the time taken for performing each *forward()* as a function of *i* is shown as (plotted in excel)



We can see that it has the same trending as Part C. But much faster.

main.py

```
# perform 2d convolution
import torch
from PIL import Image
import torchvision.transforms as transforms
import torchvision
import matplotlib.pyplot as plt
import time



from conv import Conv2D


# convert PIL to tensor
pil2tensor = transforms.ToTensor()
tensor2pil = transforms.ToPILImage()


img_pil = Image.open('1280x720.jpg')
img = pil2tensor(img_pil) # convert JpegImageFile object to tensor


# task1
conv2d = Conv2D(in_channel=3, o_channel=1, kernel_size=3, stride=1, mode='known')
[numOperates, outImg] = conv2d.forward(img)
print(numOperates)
torchvision.utils.save_image(outImg, 'task1_1280x720.jpg', padding = 0, normalize = True)


if False:
    # task2
    conv2d = Conv2D(in_channel=3, o_channel=2, kernel_size=5, stride=1, mode='known')
    [numOperates, outImg] = conv2d.forward(img)
```

```
    print(numOperates)
    torchvision.utils.save_image(outImg[0], 'task2_1280x720_K4.jpg', padding = 0, normalize = True)
    torchvision.utils.save_image(outImg[1], 'task2_1280x720_K5.jpg', padding = 0, normalize = True)

if False:
    # task3
    conv2d = Conv2D(in_channel=3, o_channel=3, kernel_size=3, stride=2, mode='known')
    [numOperates, outImg] = conv2d.forward(img)
    print(numOperates)
    torchvision.utils.save_image(outImg[0], 'task3_1920x1080_K1.jpg', padding = 0, normalize = 
True)
    torchvision.utils.save_image(outImg[1], 'task3_1920x1080_K2.jpg', padding = 0, normalize = 
True)
    torchvision.utils.save_image(outImg[2], 'task3_1920x1080_K3.jpg', padding = 0, normalize = 
True)



# Part B
if False:
    timeB = torch.zeros([11, 1], dtype = torch.float64)
    for i in range(0,11):
        a = time.time()
        conv2d = Conv2D(in_channel=3, o_channel= 2**i, kernel_size=3, stride=1, mode='rand')
        [numOperates, outImg] = conv2d.forward(img)
        timeB[i] = time.time() - a
        print( str(i) + ': ' + str(timeB[i]) )

    plt.plot(timeB)
    plt.show()
```

```
# Part C
if False:
    numC = torch.zeros([5, 1], dtype = torch.int32)
    for i in range(0, 5):
        i
        a = time.time()
        conv2d = Conv2D(in_channel=3, o_channel= 2, kernel_size=(i+1)*2+1, stride=1, mode='rand')
        [numOperates, outImg] = conv2d.forward(img)
        print('time is ' + str(time.time() - a) )
        numC[i] = numOperates
        print(numOperates)

    plt.plot(numC)
    plt.show()
```

conv.py

```
# perform 2d convolution
import torch
from PIL import Image
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

#Conv2D(in_channel, o_channel, kernel_size, stride, mode)


class Conv2D(object):

    def __init__(self, in_channel, o_channel, kernel_size, stride, mode):
        self.in_channel = in_channel
        self.o_channel = o_channel
        self.kernel_size = kernel_size
        self.stride = stride
        self.mode = mode

    def forward(self, input_image):

        def out_image(K, imgR, imgG, imgB, kernel_size, stride):

            [r, c] = imgR.size()
            d = int(kernel_size / 2)

            outR = torch.zeros([r-kernel_size+1, c-kernel_size+1], dtype = torch.float64)
            outG = torch.zeros([r-kernel_size+1, c-kernel_size+1], dtype = torch.float64)
```

```
        outB = torch.zeros([r-kernel_size+1, c-kernel_size+1], dtype = torch.float64)

        numCalulates = 0

        for i in range(d, r-d, stride):
            for j in range(d, c-d, stride):
                blkR = imgR[i-d: i+d+1, j-d: j+d+1]
                outR[i-d, j-d] = (blkR * K).sum()

                blkG = imgG[i-d: i+d+1, j-d: j+d+1]
                outG[i-d, j-d] = (blkG * K).sum()

                blkB = imgB[i-d: i+d+1, j-d: j+d+1]
                outB[i-d, j-d] = (blkB * K).sum()

                numCalulates = numCalulates + (K.size()[0] * K.size()[1]*2 -1)*3

        outImg = (outR + outG + outB) / 3.0

        return numCalulates, outImg

    print(input_image.size())
    imgR = input_image[0]
    imgG = input_image[1]
    imgB = input_image[2]

    [r, c] = imgR.size()

    # output image
```

```
        outImg = torch.zeros(self.o_channel, r-self.kernel_size+1, c-self.kernel_size+1)  # output grayscale image

        # Part B
        if(self.kernel_size == 3 and self.mode == 'rand' and self.stride == 1):
            numOperates = 0

            for i in range(0, self.o_channel):
                Kb = torch.rand(3,3) *2 -1  # in range [-1, 1]
                [numCalulates, outImg1] = out_image(Kb, imgR, imgG, imgB, self.kernel_size, self.stride)
                numOperates = numOperates + numCalulates
                outImg[i] = outImg1
            return numOperates, outImg



        # Part C
        #if False:
        elif(self.o_channel == 2 and self.mode == 'rand' and self.stride == 1):
            Kc1 = torch.rand(self.kernel_size, self.kernel_size) *2 -1
            Kc2 = torch.rand(self.kernel_size, self.kernel_size) *2 -1
            numOperates = 0

            [numCalulates, outImg1] = out_image(Kc1, imgR, imgG, imgB, self.kernel_size, self.stride)
            outImg[0] = outImg1
            numOperates = numOperates + numCalulates

            [numCalulates, outImg1] = out_image(Kc2, imgR, imgG, imgB, self.kernel_size, self.stride)
```

```
            outImg[1] = outImg1
            numOperates = numOperates + numCalulates

            return numOperates, outImg


        # task 1
        elif(self.o_channel == 1 and self.kernel_size == 3 and self.mode == 'known' and
self.stride == 1):
            numOperates = 0
            K1 = torch.tensor([[-1., -1., -1.], [0., 0., 0.], [1., 1., 1.]])
            K1 = K1.transpose(0, 1)

            [numOperates, outImg1] = out_image(K1, imgR, imgG, imgB, self.kernel_size,
self.stride)
            outImg = outImg1
            return numOperates, outImg

        # task 2
        #if False:
        elif(self.o_channel == 2 and self.kernel_size == 5 and self.mode == 'known' and
self.stride == 1):
            K4 = torch.tensor([[-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.]])
            K4 = K4.transpose(0, 1)

            K5 = torch.tensor([[-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.]])
            K5 = K5.transpose(0, 1)
```

```
            numOperates = 0

            [numCalulates, outImg1] = out_image(K4, imgR, imgG, imgB, self.kernel_size,
self.stride)
            outImg[0] = outImg1
            numOperates = numOperates + numCalulates

            [numCalulates, outImg1] = out_image(K5, imgR, imgG, imgB, self.kernel_size,
self.stride)
            outImg[1] = outImg1
            numOperates = numOperates + numCalulates

            return numOperates, outImg

        # task 3
        #if False:
        elif(self.o_channel == 3 and self.kernel_size == 3 and self.mode == 'known' and
self.stride == 2):
            K1 = torch.tensor([[-1., -1., -1.], [0., 0., 0.], [1., 1., 1.]])
            K1 = K1.transpose(0, 1)

            K2 = torch.tensor([[-1., 0., 1.], [-1., 0., 1.], [-1., 0., 1.]])
            K2 = K1.transpose(0, 1)

            K3 = torch.tensor([[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])
            K3 = K1.transpose(0, 1)

            numOperates = 0
```

```
            [numCalulates, outImg1] = out_image(K1, imgR, imgG, imgB, self.kernel_size, self.stride)
            outImg[0] = outImg1
            numOperates = numOperates + numCalulates

            [numCalulates, outImg1] = out_image(K2, imgR, imgG, imgB, self.kernel_size, self.stride)
            outImg[1] = outImg1
            numOperates = numOperates + numCalulates

            [numCalulates, outImg1] = out_image(K3, imgR, imgG, imgB, self.kernel_size, self.stride)
            outImg[2] = outImg1
            numOperates = numOperates + numCalulates

            return numOperates, outImg
```

main.c

// perform 2d convolution. Output channel 2^i (i=0,1,...,10)

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


int c_conv(int in_channel, int o_channel, int kernel_size, int stride)
{
    int height = 1280;
    int width = 720;
    size_t nbytes = height * width * in_channel * sizeof(char);
    int i, j, k, m, n, p;
    double val;
    long int row;
    int num = 0;

    // simulate input image as all black
    char *input_img = (char*)malloc(nbytes);
    memset(input_img, 0, nbytes);

    char *out_img = (char *)malloc((height - 1) * (width - 1) * sizeof(char));
    memset(out_img, 0, (height - 1) * (width - 1) * sizeof(char));

    double *k1 = (double*)malloc(kernel_size*kernel_size*sizeof(double));

    for (m = 0; m < o_channel; m++)
    {
```

```
    // randomly generate kernel
    for (p = 0; p<kernel_size*kernel_size; p++)
        k1[p] = rand() / (2.0) - 1;


    for (i = 1; i < height - 1; i++)
      for (j = 1; j < width - 1; j++)
      {
        val = 0.0;
        row = i * width + j;


        for (n = 0; n < in_channel; n++)
        {
          val += input_img[(row - width - 1) * in_channel + n] * k1[0] + input_img[(row - width) * in_channel + n] * k1[1] + input_img[(row - width + 1) * in_channel + n] * k1[2] +
            input_img[(row - 1) * in_channel + n] * k1[3] + input_img[(row) * in_channel + n] * k1[4] + input_img[(row + 1) * in_channel + n] * k1[5] +
            input_img[(row + width - 1) * in_channel + n] * k1[6] + input_img[(row + width) * in_channel + n] * k1[7] + input_img[(row + width + 1) * in_channel + n] * k1[8];
        }
        out_img[(i - 1)*(width - 1) + j - 1] = val / (double)in_channel;
        num += (kernel_size * kernel_size * 2 - 1) * in_channel;
      }
  }
  return num;
}



void main()
{
       int o_channel[11]; // output channels
```

```
            clock_t t;
            double times[11];
            double val; // value storing convolution results
  int in_channel = 3, kernel_size = 3, stride = 1;
  int numOperates;
  int num;
  int k;


            for (k = 0; k < 11; k++)
            {
                        o_channel[k] = pow(2, k); // # of output channels
    printf("%d:  ", o_channel[k]);
    numOperates = 0;

                        // do 2d convolution
                        t = clock();
    num = c_conv(in_channel, o_channel[k], kernel_size, stride);
    numOperates += num;


                        times[k] = (clock() - t) / (double)CLOCKS_PER_SEC;

                        printf("%f\n", times[k]);
            }

}
```