Ruixuan Shen
Hoang Mai Diem Pham
EC ENGR113DA

MINI-PROJECT 2: Classify handwritten digits with Convolutional Neural Network (CNN)

1. Introduction

In the lab, we implemented the program to recognize handwritten digits using the Convolutional Neural Network model on the H7 board.

We first used python, TensorFlow, and Google Colab to train the CNN model in the cloud. The MNIST database for handwritten digits was used as the training set. The training images all have 1 channel and 28 pixels by 28 pixels for their size.

After the training, we implemented the functional modules for CNN and the CNN image classifiers on the H7 board. The CNN structure we designed is shown below:

the output width, output height, and the number of channels are shown in the screenshot below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 28, 28, 16)        160

max_pooling2d_2 (MaxPooling  (None, 14, 14, 16)        0
2D)

conv2d_3 (Conv2D)            (None, 14, 14, 32)        4640

max_pooling2d_3 (MaxPooling  (None, 7, 7, 32)          0
2D)

flatten_1 (Flatten)          (None, 1568)              0

dense_1 (Dense)              (None, 10)                15690

=================================================================
Total params: 20,490
Trainable params: 20,490
Non-trainable params: 0
_________________________________________________________________
```

For the convolution layer, we first padded the input matrix with zeros on four of the outer edges, then convolve the padded input with the filter and stride the filter across the padded matrix. After the striding, we applied ReLU to every element to eliminate the negative results.

For the pooling layer, we applied max pooling for each 2*2 block of the input, thus returning a matrix with ½ side length on each side.

For the dense layer, after applying the neuron weight and the bias to the input, we applied the softmax function $P(y=j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$ for normalization and produce the discrete probability distribution vector for each digit.

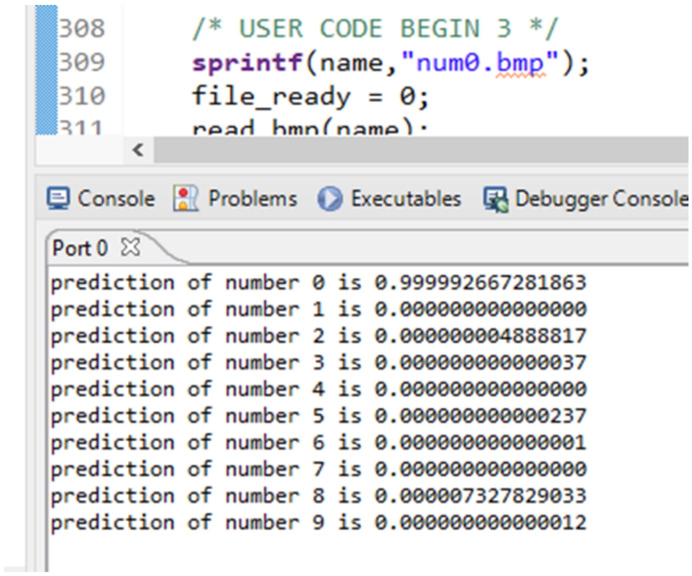
The accuracy for the training in the designed structure was evaluated by python and recorded as below:

```
Accuracy of this model is:
313/313 [==============================] - 3s 10ms/step - loss: 0.1385 - accuracy: 0.9842
[0.13846145570278168, 0.9842000007629395]
```

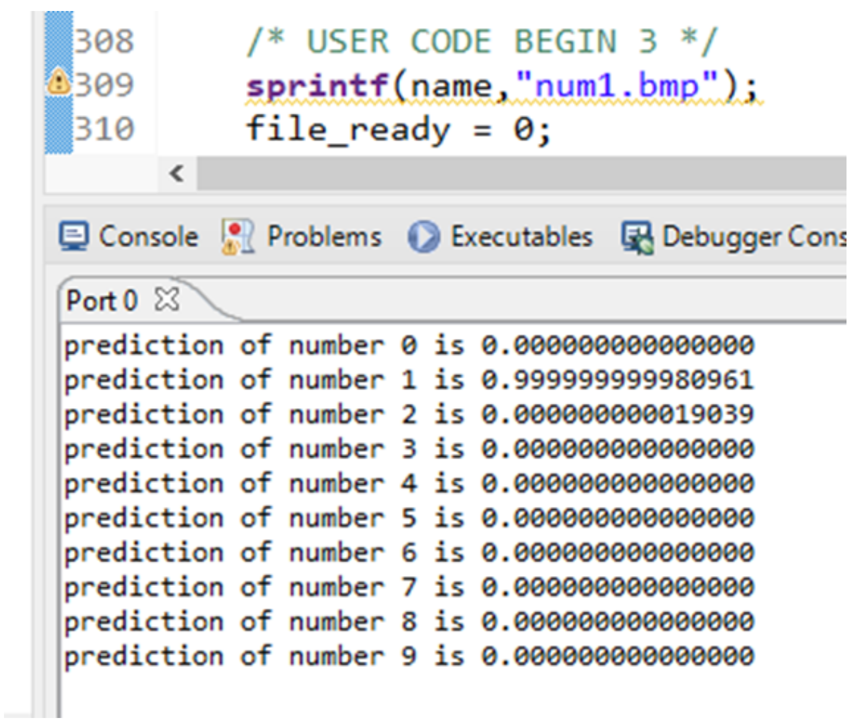
2. Result

The program failed to recognize the digit 8 from the provided test images while recognizing the rest test images successfully. The details are shown below:

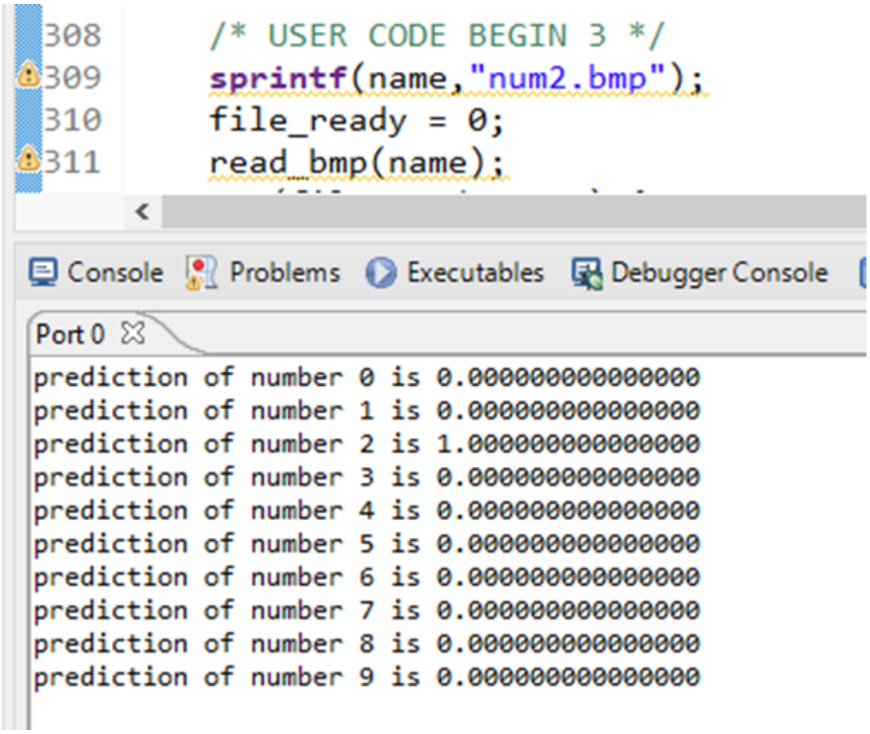
2.1. Check number 0:



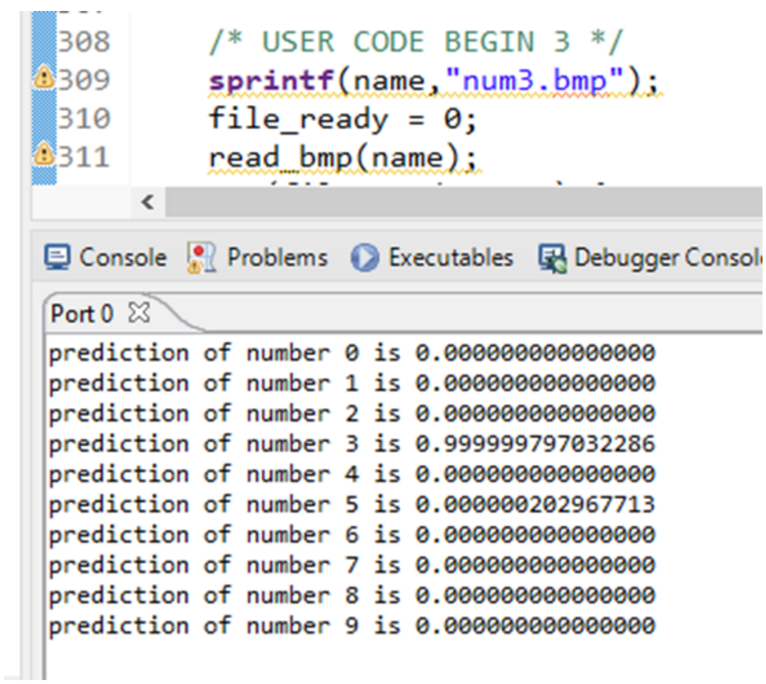
2.2. Check number 1:



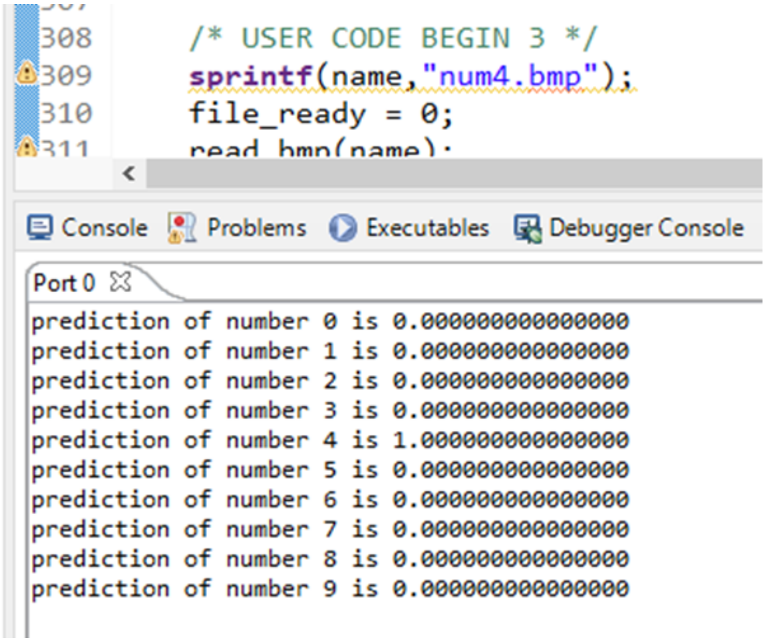
2.3. Check number 2:



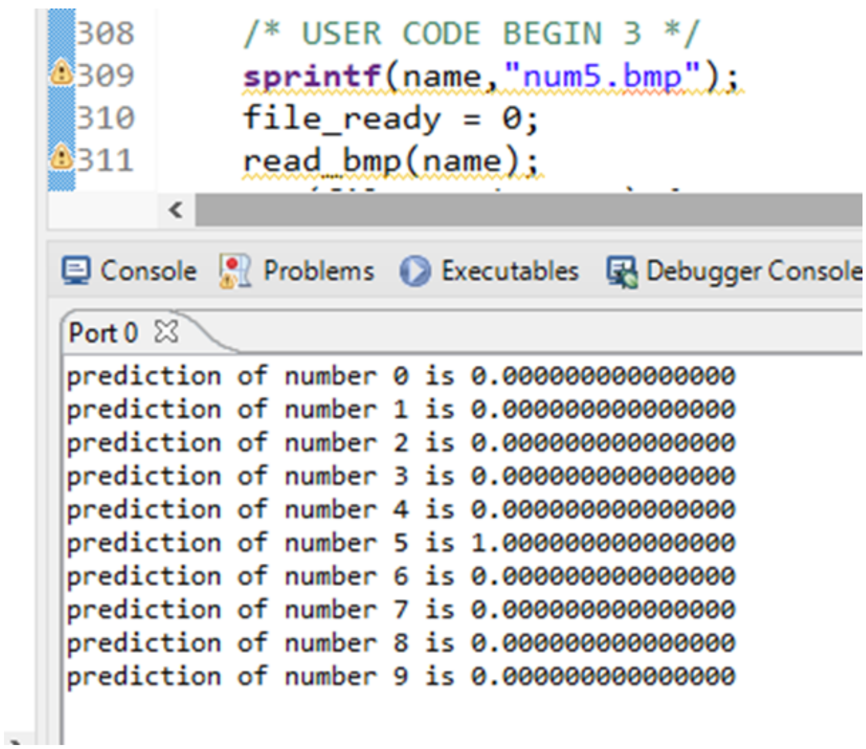
2.4. Check number 3:



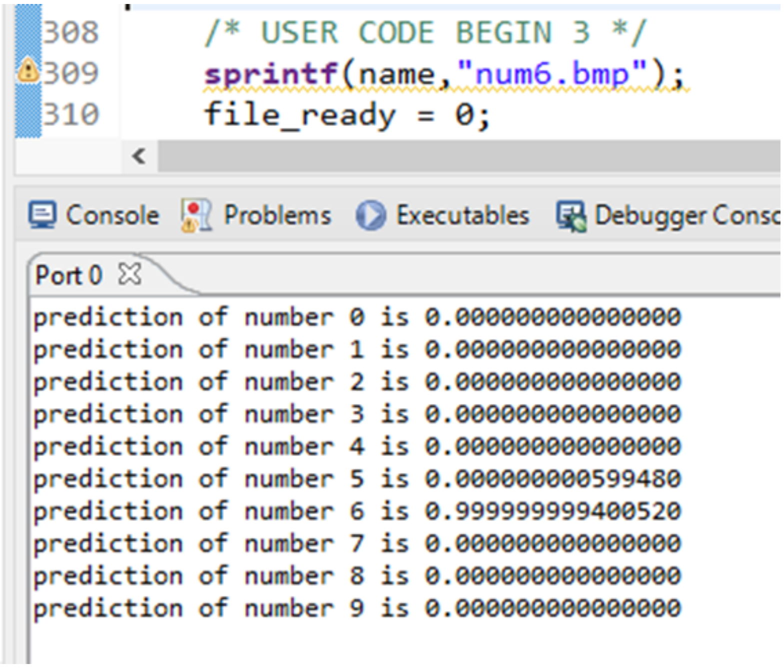
2.5. Check number 4:



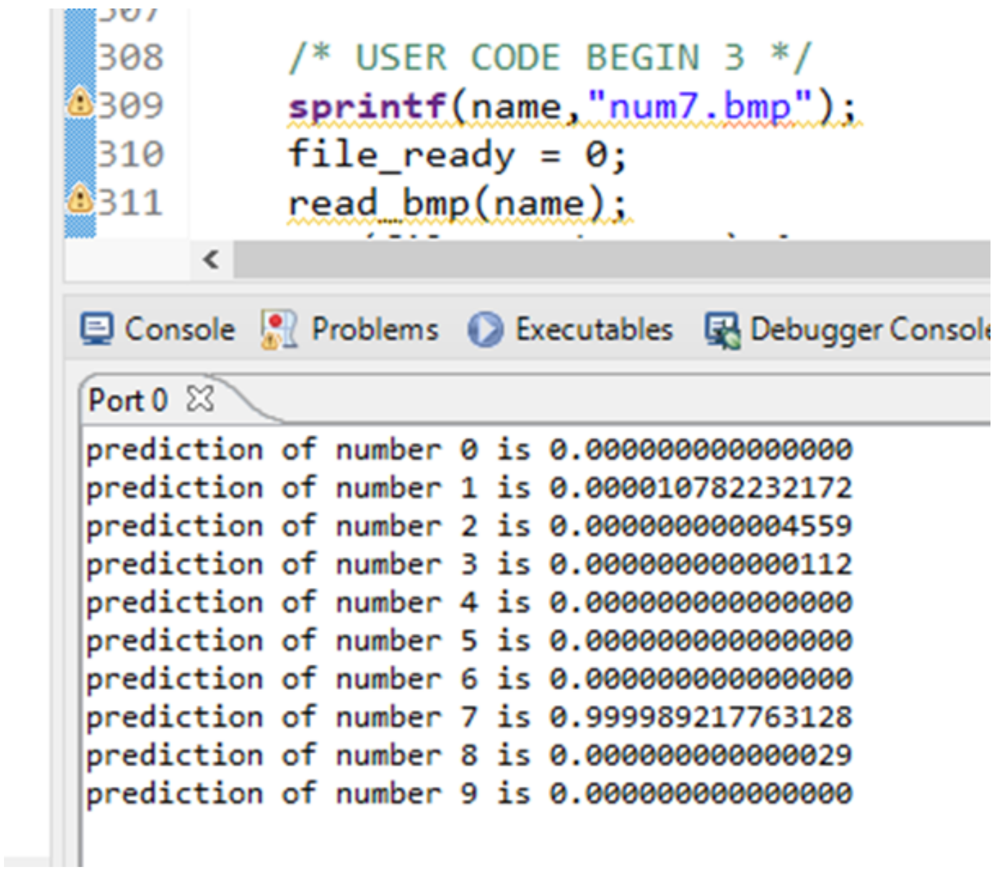
2.6. Check number 5:



2.7. Check number 6:



2.8. Check number 7:



2.9. Check number 8:

```
/* USER CODE BEGIN 3 */
sprintf(name,"num8.bmp");
file_ready = 0;
read_bmp(name);
```

Console | Problems | Executables | Debugger Conso

Port 0

```
prediction of number 0 is 0.000000000000000
prediction of number 1 is 0.000000000000000
prediction of number 2 is 0.000000000000000
prediction of number 3 is 0.000000275727878
prediction of number 4 is 0.000000000000000
prediction of number 5 is 0.000381166275216
prediction of number 6 is 0.999616711888520
prediction of number 7 is 0.000000000000000
prediction of number 8 is 0.000001846108386
prediction of number 9 is 0.000000000000000
```

2.10. Check number 9:

```
/* USER CODE BEGIN 3 */
sprintf(name,"num9.bmp");
file_ready = 0;
read_bmp(name);
```

Console | Problems | Executables | Debugger Cor

Port 0

```
prediction of number 0 is 0.000000000000000
prediction of number 1 is 0.000000000245994
prediction of number 2 is 0.000000001384368
prediction of number 3 is 0.009116702831457
prediction of number 4 is 0.000000000000021
prediction of number 5 is 0.000000000031443
prediction of number 6 is 0.000000000000000
prediction of number 7 is 0.171443969332484
prediction of number 8 is 0.000000001549626
prediction of number 9 is 0.819439324624608
```

3. Discussion

We planned to have two convolution layers and two pooling layers in the first place. When we first tested our model, however, we used very small numbers for each convolution layer channel. (6 for each.) This led to many failures in the digit recognition. Only half of the test images were recognized. As we increased the number of channels, the program worked better.

Still, the program is not able to recognize 8. We tried to modify the “epochs” in the training steps as the project instruction suggested. Unfortunately, there was a limited improvement and the digit 8 was still not recognized. Further testing and modifications are needed for fixing this issue.

4. Code

4.1. main():

```c
    /* USER CODE BEGIN 3 */

    sprintf(name,"num9.bmp");
    file_ready = 0;
    read_bmp(name);
    if (file_ready == 1) {
        out_img = ProcessBmp(rtext);
        break;
    }
  }

  //read in parameters
  sprintf(name,"b1.txt");
  float * b1 = read_txt(name, 16);

  sprintf(name,"w1.txt");
  float * w1 = read_txt(name, 144);

  sprintf(name,"b2.txt");
  float * b2 = read_txt(name, 32);

  sprintf(name,"w2.txt");
  float * w2 = read_txt(name, 4608);

  sprintf(name,"bc.txt");
  float * bc = read_txt(name, 10);

  sprintf(name,"fc.txt");
  float * fc = read_txt(name, 15680);
```

```c
  // perform NN operation
  float *result_conv_1 = (float *)malloc(28*28*16*sizeof(float));
  conv(out_img, w1, b1, result_conv_1, 28, 28, 1, 3, 16);

  float *result_pool_1 = (float *)malloc(14*14*16*sizeof(float));
  pool(result_conv_1, result_pool_1, 28, 28, 16, 2);
  free(result_conv_1);
  free(w1);
  free(b1);

  float *result_conv_2 = (float *)malloc(14*14*32*sizeof(float));
  conv(result_pool_1, w2, b2, result_conv_2, 14, 14, 16, 3, 32);
  free(result_pool_1);

  float *result_pool_2 = (float *)malloc(7*7*32*sizeof(float));
  pool(result_conv_2, result_pool_2, 14, 14, 32, 2);
  free(result_conv_2);
  free(w2);
  free(b2);

  double *result_dense = (double *)malloc(10*sizeof(double));
  dense(result_pool_2, fc, bc, result_dense, 7*7*32, 10);
  free(result_pool_2);

  for (int i=0;i<10;i++){
      printf("prediction of number %d is %.15f\n", i, result_dense[i]);
  }
  free(result_dense);

  while(1);
  /* USER CODE END 3 */
}
```

4.2. Convolution function:

```c
void padding(float input[],int input_height, int input_width){
      int input_size = input_height * input_width;
      float input_1[input_size];
      for (int i = 0; i < input_size; i++) {
          input_1[i] = input[i];
      }

      int padwid = input_width + 2;
      int padheight = input_height +2;
      int padsize = padwid * padheight;

      for (int i = 0; i < padwid; i++) {
          input[i] = 0;
      }

      for (int i= padwid * (input_height + 1); i < padsize; i++) {
          input[i] = 0;
      }

      for (int i = padwid; i < padwid * (input_height + 1); i++){
           if (i% padwid == 0 || i % padwid == padwid-1) {
            input[i] = 0;
           }
           else {
            input[i] = input_1[i-(padwid+1)-((int)(i/padwid)-1)*2];
           }
      }
}
```

```c
void conv (float input[], float kernel[], float bias[], float result[], int input_height, int input_width, int input_channel, int kernel_size, int kernel_channel )
{
    padding(input, input_height, input_width);
    int result_size = input_height * input_width * kernel_channel;
    float sum;
    for(int n=0;n<kernel_channel;n++){
        for (int i=0;i<input_height;i++){
            for (int j=0;j<input_width;j++){
                sum=0;
                for(int k=0;k<kernel_channel;k++){
                    sum+=(input[(i+0)*input_channel*kernel_channel+j*kernel_channel+k]*kernel[0*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+0)*16*kernel_channel+(j+1)*kernel_channel+k]*kernel[1*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+0)*input_channel*kernel_channel+(j+2)*kernel_channel+k]*kernel[2*kernel_channel*kernel_channel+k*kernel_channel+n]);
                    sum+=(input[(i+1)*input_channel*kernel_channel+j*kernel_channel+k]*kernel[3*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+1)*16*kernel_channel+(j+1)*kernel_channel+k]*kernel[4*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+1)*input_channel*kernel_channel+(j+2)*kernel_channel+k]*kernel[5*kernel_channel*kernel_channel+k*kernel_channel+n]);
                    sum+=(input[(i+2)*input_channel*kernel_channel+j*kernel_channel+k]*kernel[6*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+2)*16*kernel_channel+(j+1)*kernel_channel+k]*kernel[7*kernel_channel*kernel_channel+k*kernel_channel+n])
                          +(input[(i+2)*input_channel*kernel_channel+(j+2)*kernel_channel+k]*kernel[8*kernel_channel*kernel_channel+k*kernel_channel+n]);
                }
                sum+=bias[n];
                result[i*input_height*kernel_channel+j*kernel_channel+n]=sum;
                }
            }
        }

    //ReLU: Applied to every number after convolution
    for (int i = 0; i < result_size; i++) {
        if (result[i] < 0) { result[i] = 0; }
    }
}
```

4.3. Pooling function:

```
void pool(float input[], float output[], int height, int width, int channel, int pool_size)
{
    float a, b;
    for (int k = 0; k < (height/pool_size)*(width/pool_size)*channel ; k=k+(height/pool_size)){
        for (int i = 0; i < (height/pool_size); i++) {
            a = fmax(input[pool_size*(i+k)+pool_size*k], input[pool_size*(i+k)+pool_size*k+1]);
            b = fmax(input[pool_size*(i+k)+pool_size*k +width], input[pool_size*(i+k)+pool_size*k + width +1]);
            output[i+k] = fmax (a,b);
        }
    }
}
```

4.4. Dense function:

```
void dense(float input[], float kernel[], float bias[], double output[], int input_size, int output_size)
{
    float y[output_size];
        for (int i = 0; i < output_size; i++){
            double sum = 0;
            for (int j=0; j<input_size; j++){
                sum = sum + kernel[i+output_size*j]*input[j];
            }
            y[i] = sum + bias[i];
        }

        double total = 0;
        for (int i = 0; i < output_size; i++){
            total = total + exp(y[i]);
        }

        for (int i = 0; i < output_size; i++){
            output[i] = exp(y[i]) / total;
        }
}
```

Python Code

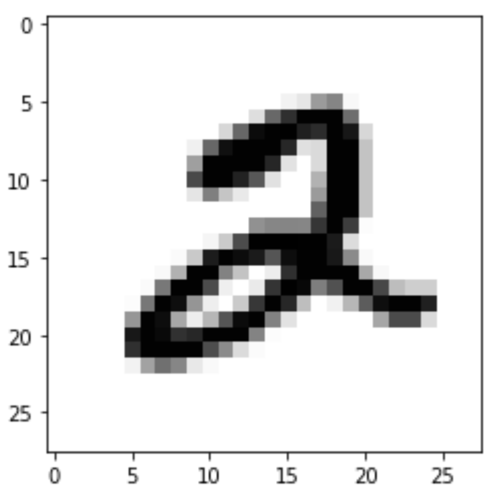
```
#import libraries for nerualnet, math and visualization
from __future__ import absolute_import, division, print_function, unicode_literals

try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import datasets, layers, models
print(tf.__version__)
```

```
2.7.0
```

```
#set seed just for the demonstration
#tf.random.set_seed(1000);
#load in the MNIST dataset for training and testing
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()
```

```
#Plot an image to see what it looks like
plt.figure()
plt.imshow(train_images[5], cmap=plt.cm.binary)
plt.grid(False)
plt.show()
```




```
#TODO: Modify the CNN structure for a slimer network
#Build the neuralnet model
#model = models.Sequential()
#model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='sam
#model.add(layers.MaxPooling2D((2, 2)))
#model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
#model.add(layers.MaxPooling2D((2, 2)))
```

```
#model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
#model.add(layers.MaxPooling2D((2, 2)))
#model.add(layers.Flatten())
#model.add(layers.Dense(10, activation='softmax'))


#diem
model = models.Sequential()
model.add(layers.Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
#model.add(layers.Conv2D(6, (3, 3), activation='relu', padding='same'))
#model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))
#Review the overall model structure
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 28, 28, 16)        160

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 16)       0
 2D)

 conv2d_3 (Conv2D)           (None, 14, 14, 32)        4640

 max_pooling2d_3 (MaxPooling  (None, 7, 7, 32)         0
 2D)

 flatten_1 (Flatten)         (None, 1568)              0

 dense_1 (Dense)             (None, 10)                15690

=================================================================
Total params: 20,490
Trainable params: 20,490
Non-trainable params: 0
_________________________________________________________________
```

```
#Review the overall model structure
#model.summary()


#Reshape the image so it can train in batch (and fit the model's input shape)
train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))


#Training the model
#Hint: change optimizer to 'sgd', and increase epochs if result is bad.
```

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=20)
```

```
Epoch 1/20
1875/1875 [==============================] - 42s 22ms/step - loss: 0.4031 - accuracy: 0
Epoch 2/20
1875/1875 [==============================] - 43s 23ms/step - loss: 0.0764 - accuracy: 0
Epoch 3/20
1875/1875 [==============================] - 44s 24ms/step - loss: 0.0581 - accuracy: 0
Epoch 4/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0494 - accuracy: 0
Epoch 5/20
1875/1875 [==============================] - 43s 23ms/step - loss: 0.0414 - accuracy: 0
Epoch 6/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0405 - accuracy: 0
Epoch 7/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0383 - accuracy: 0
Epoch 8/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0301 - accuracy: 0
Epoch 9/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0295 - accuracy: 0
Epoch 10/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0295 - accuracy: 0
Epoch 11/20
1875/1875 [==============================] - 44s 24ms/step - loss: 0.0278 - accuracy: 0
Epoch 12/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0258 - accuracy: 0
Epoch 13/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0274 - accuracy: 0
Epoch 14/20
1875/1875 [==============================] - 44s 24ms/step - loss: 0.0234 - accuracy: 0
Epoch 15/20
1875/1875 [==============================] - 44s 24ms/step - loss: 0.0214 - accuracy: 0
Epoch 16/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0228 - accuracy: 0
Epoch 17/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0228 - accuracy: 0
Epoch 18/20
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0198 - accuracy: 0
Epoch 19/20
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0248 - accuracy: 0
Epoch 20/20
1875/1875 [==============================] - 43s 23ms/step - loss: 0.0219 - accuracy: 0
<keras.callbacks.History at 0x7effd1b8c4d0>
```

```
#Evaluate the performance with testing dataset
print("Accuracy of this model is:")
model.evaluate(test_images, test_labels)
```

```
Accuracy of this model is:
```

```
313/313 [==============================] - 3s 10ms/step - loss: 0.1385 - accuracy: 0.98
[0.13846145570278168, 0.9842000007629395]
```

```
#View the total number of parameters, so it doesn't overflow the LCDK's memory
print("Total amount of parameter of model is:", model.count_params())
```

```
Total amount of parameter of model is: 20490
```

```
#Example for extract parameter form the first conv layer
#TODO: you need to actually modify model.layers[XXXX], this XXX to fit your actually layer nu
t1, t2 = model.layers[0].get_weights()
np.savetxt('w1.txt', t1.flatten(), delimiter=',',fmt='%.16f')
np.savetxt('b1.txt', t2.flatten(), delimiter=',',fmt='%.16f')
```

```
#Example for extract parameter form the second conv layer
#TODO: you need to actually modify model.layers[XXXX], this XXX to fit your actually layer nu
#And do it multiple times to save all the layer with parameters
t1, t2 = model.layers[2].get_weights()
np.savetxt('w2.txt', t1.flatten(), delimiter=',',fmt='%.16f')
np.savetxt('b2.txt', t2.flatten(), delimiter=',',fmt='%.16f')
```

```
# t1, t2 = model.layers[4].get_weights()
# np.savetxt('w3.txt', t1.flatten(), delimiter=',',fmt='%.16f')
# np.savetxt('b3.txt', t2.flatten(), delimiter=',',fmt='%.16f')
```

```
t1, t2 = model.layers[5].get_weights()
np.savetxt('fc.txt', t1.flatten(), delimiter=',',fmt='%.16f')
np.savetxt('bc.txt', t2.flatten(), delimiter=',',fmt='%.16f')
```

```
import cv2
```

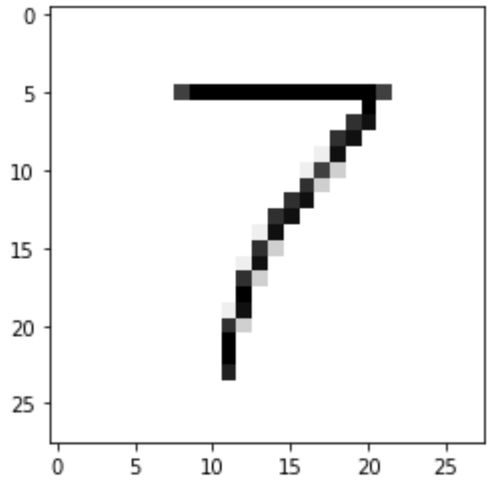
```
im = cv2.imread("num7.bmp")
im = cv2.flip(im,0)
im = im[:,:,0]
for i in range (28):
  for j in range (28):
    im[i,j]=255 -im[i,j]
```

```
im.shape
```

```
(28, 28)
```

```
plt.imshow(im, cmap=plt.cm.binary)
```

```
<matplotlib.image.AxesImage at 0x7effd18bcbd0>
```



```
im = im.reshape(1,28,28,1)
```

```
model.predict(im)
```

```
array([[6.9705730e-35, 1.5201940e-16, 1.3742151e-22, 2.0156587e-24,
        1.5461872e-13, 1.7622024e-19, 2.1085472e-21, 1.0000000e+00,
        2.8086321e-19, 6.5138695e-19]], dtype=float32)
```