Keras-CNN卷积神经网络训练识别手写数字模型

目录

网上的开源项目良莠不齐，连个能跑的都没有，很多都是因为包版本不够运行不起来。因此这里先从项目的配置开始。

一、配置

python

```
python 3.11.0
```

External Libs(主要是以下几个)

```
numpy 1.26.0
keras 2.14.0
tensorflow 2.14.0
opencv-python 4.8.0.76
```

Mnist数据集

```
通过Kares集成的mnist模块在线下载的，不用手动去yann下载了
（如果需要下载，选择http，不然要账号密码）
```

尝试复现（Lec3是对应的代码）

```
https://github.com/Liyanhao1209/Machine_Vision_Practice.git
```

二、模型训练

2.0 轮子

无论网络架构长啥样，总归要有对应几个层的轮子（卷积、池化、全连接），先导包。

```
import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten
from keras.utils import to_categorical
import tensorflow._api.v2.compat.v1 as tf
import keras
from keras.datasets import mnist
tf.disable_v2_behavior()
```

`tf.disable_v2_behavior`是因为tf的api有两个版本，我们用的第一个，所以屏蔽第二个版本的API的行为。

1. Sequential：我个人理解就是CNN框架本身，像个队列容器，后续的各层逐一进入，随后模型训练的时候按照入队顺序进行各层的操作。
2. Conv2D：卷积层
3. MaxPool2D：池化层
4. Flatten：展平层
5. Dense：全连接层
6. 其他：一些工具函数

2.1 数据预处理

先把数据导进来（本质上是在线下载）

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

（后续我会用opencv读取数据，看看这些手写数字都长啥样）

这里放一张样例

大概就长这样，黑底白字，28px * 28px。训练集60000张，测试集10000张，有监督。

一般用来训练和测试的数据都是灰度图。但Mnist的数据集已经是灰度图了。（单通道）
所以在后面我们编写用户程序的时候，需要先把RGB转灰度图，这里就不用转了。

但这里因为API自己的问题，我们要显式声明一下我们的数据确实是单通道的灰度图。

```
img_x, img_y = X_train.shape[1], X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], img_x, img_y, 1)
X_test = X_test.reshape(X_test.shape[0], img_x, img_y, 1)
```

然后归一化。（确保精度）

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

随后要把标量标签矢量化（API要求的）

简单来说，如果标签为i，那么要生成一个长度为k（调用者指定，一般为10，下面的例子为9），第i位为1，其余均为0的向量，每有一个标签生成这么一个向量，最后拼成矩阵。

eg.

```
from keras.utils.np_utils import *
#类别向量定义
b = [0,1,2,3,4,5,6,7,8]
```

```
#调用to_categorical将b按照9个类别来进行转换
b = to_categorical(b, 9)
print(b)

执行结果如下：
[[1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

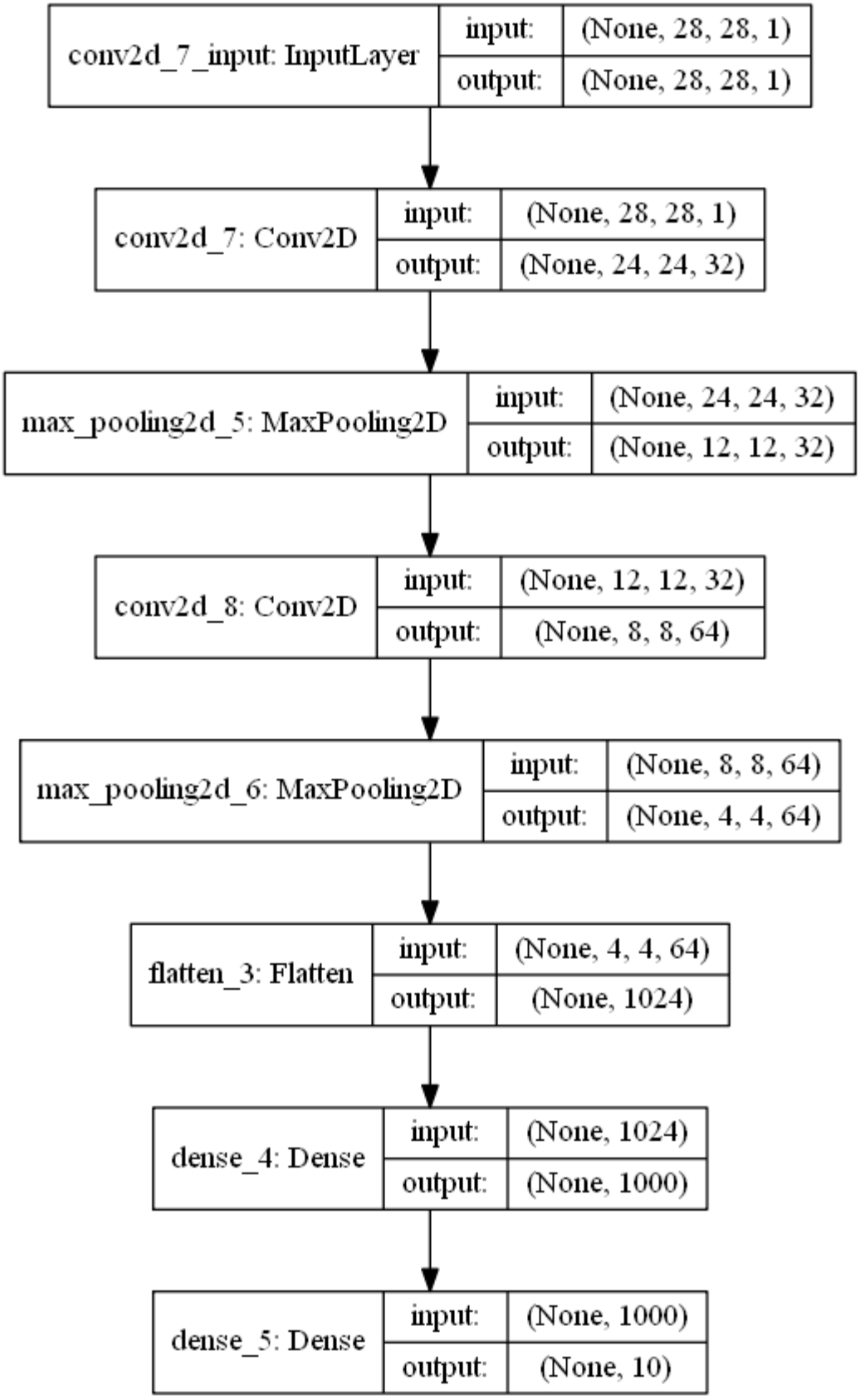
这种方式称为：**one_hot encoding(独热编码/一位有效编码)**

为什么要ohE? https://machinelearningmastery.com

至此，数据预处理算结束了。

2.2 网络架构

架构图如下：


conv2d_7_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_7: Conv2D
input: (None, 28, 28, 1)
output: (None, 24, 24, 32)
max_pooling2d_5: MaxPooling2D
input: (None, 24, 24, 32)
output: (None, 12, 12, 32)
conv2d_8: Conv2D
input: (None, 12, 12, 32)
output: (None, 8, 8, 64)
max_pooling2d_6: MaxPooling2D
input: (None, 8, 8, 64)
output: (None, 4, 4, 64)
flatten_3: Flatten
input: (None, 4, 4, 64)
output: (None, 1024)
dense_4: Dense
input: (None, 1024)
output: (None, 1000)
dense_5: Dense
input: (None, 1000)
output: (None, 10)

1. 从输入层开始，数据集28*28单通道灰度图
2. 第一层卷积：kernel大小5*5
3. 第一层池化：最大池化，pool大小2*2
4. 第二层卷积：kernel大小5*5
5. 第二层池化：最大池化，pool大小2*2
6. 展平层
7. 第一层全连接层：激活函数relu
8. 第二层全连接层：激活函数softmax

按上述入队顺序把每一层推到Sequential里

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5,5), activation='relu', input_shape=(img_x, img_y, 1)))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

2.3 模型训练

先编译模型，选一种损失函数、优化器

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

训练

```
model.fit(X_train, y_train, batch_size=128, epochs=10)
```

评估（10轮迭代训练集精度99.02%，忘截图了，重新跑一次时间很久）

```
score = model.evaluate(X_test, keras.utils.to_categorical(y_test, num_classes=10))
print('acc', score[1])
```

把模型保存下来，可以复用

```
model.save('cnn.h5')
```

这一块，包括之前的神经网络架构，对我来说都是黑盒，只能多调参了。

三、测试集精度

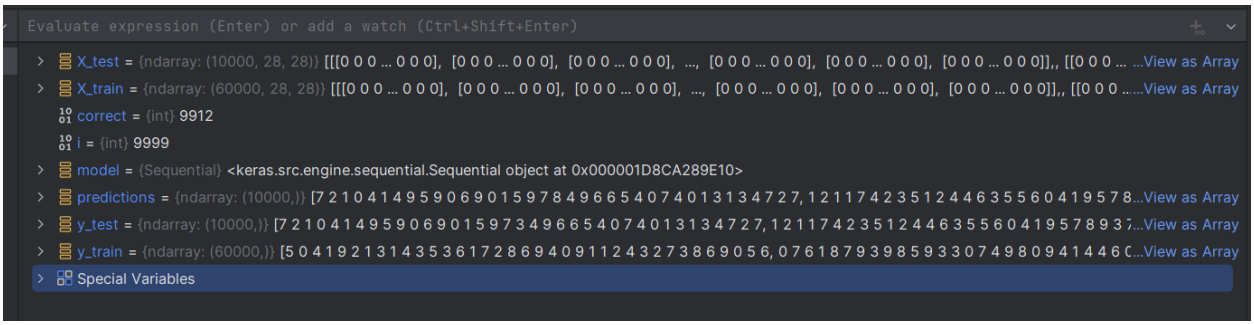
根据测试集数据，预测结果，然后和监督值比较

```
import numpy as np
from keras.datasets import mnist
from keras.models import load_model

(X_train, y_train), (X_test, y_test) = mnist.load_data()
model = load_model('cnn.h5')

predictions = np.argmax(model.predict(X_test),axis=1)
correct = 0
for i in range(len(y_test)):
    if predictions[i] == y_test[i]:
        correct += 1
print("accuracy" + str(correct / len(predictions)))
```

输出结果：

```
accuracy0.9912
```

贴个中间变量



确实是9912/10000(9999是索引，从0开始的，所以有10000个）

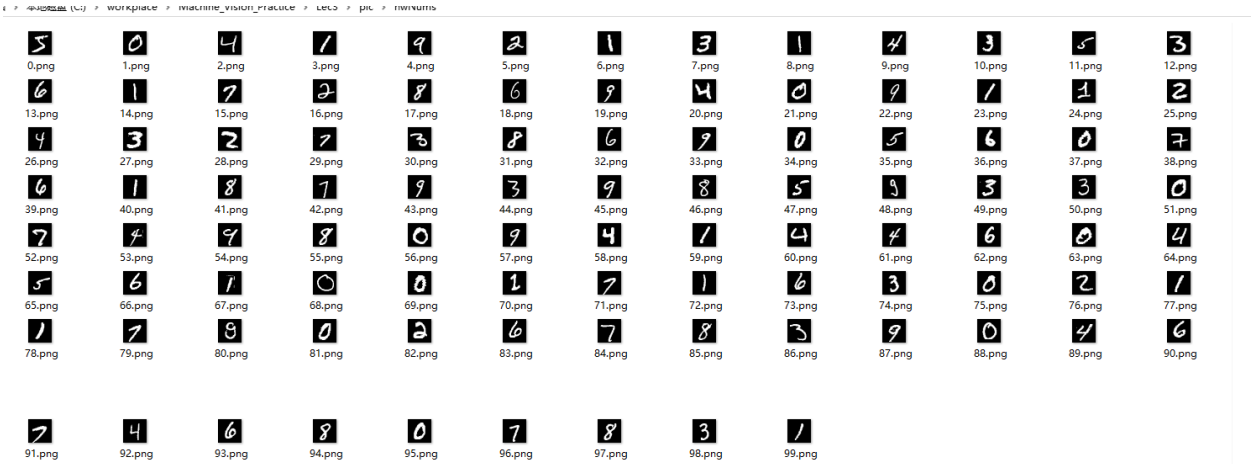
按一般要求来说，0.99的精度算是够了

四、泛化到应用程序

通过opencv把测试集读出来，然后写到一个指定文件夹里。后面就可以用这个指定文件夹里的图片做我们编写的用户程序的输入了（当然也可以自己手写）

```
# getPics.py
import cv2
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

path = "C:\\Users\\Administrator\\Desktop\\hwNums\\"
for i in range(0,100):
    cv2.imwrite(path+str(i)+".png", X_train[i])
```

这里读了100张



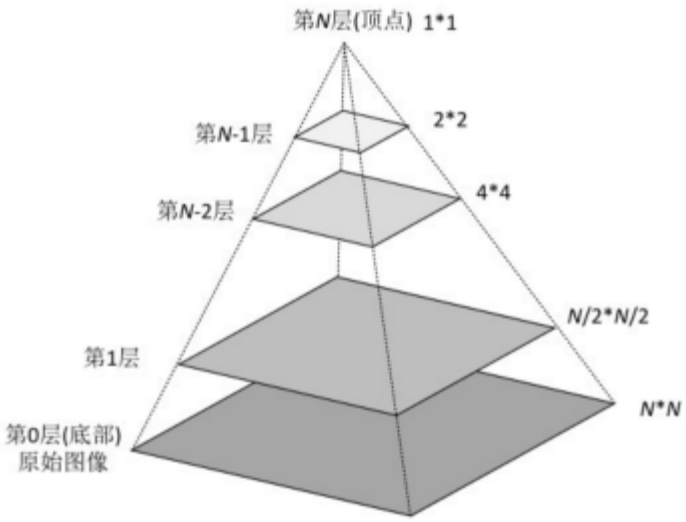
这里要注意一点，我们训练出来的模型的输入是28*28*1的ndarray，所以在复用这个模型时，我们需要让输入的数据也满足28*28*1。我读取到的图片是28*28的，但用户可能输入的不是28*28的图片，所以这里必须要对图片大小做适配。

一般来说我们可能想到cv2里的resize()函数，但resize在放缩时会丢掉大量的特征，这对于模型的预测结果来说非常不利，所以我们要换一种放缩方式。

4.0 图像金字塔

什么是图像金子塔？

图像金字塔是由一幅图像的多个不同分辨率的子图构成的图像集合。是通过一个图像不断的降低采样率产生的，最小的图像可能仅仅有一个像素点。下图是一个图像金子塔的示例。从图中可以看到，图像金字塔是一系列以金字塔形状排列的、自底向上分辨率逐渐降低的图像集合。



通常情况下，图像金字塔的底部是待处理的高分辨率图像（原始图像），而顶部则为其低分辨率的近似图像。向金字塔顶部移动时，图像的尺寸和分辨率都不断地降低。通常情况下，每向上移动一级，图像的宽和高都降低为原来的二分之一。

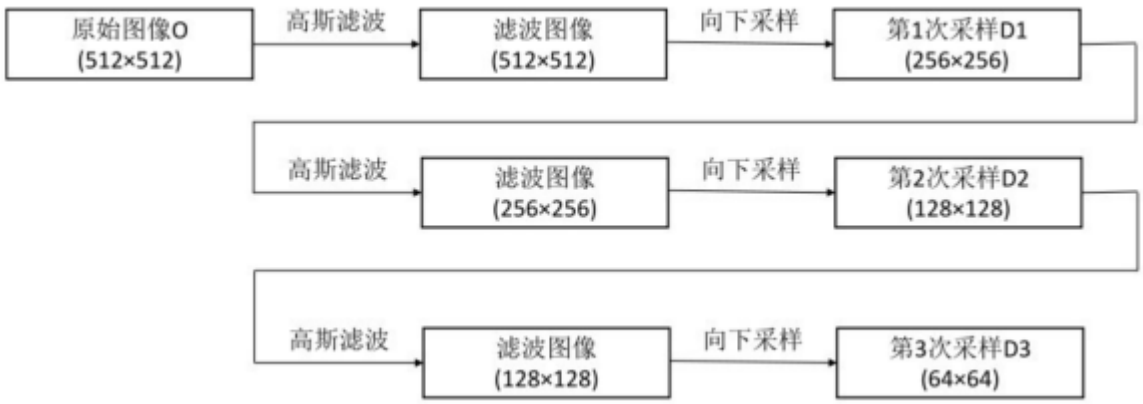
最简单的图像金字塔可以通过不断的删除图像的偶数行和偶数列得到的。例如，有一幅图像，其大小是N*N*，*删除其偶数行和偶数列后得到一幅(N/2)*(N/2)大小的图像。经过上述处理后，图像的大小变为原来的四分之一，不断重复该过程，就可以得到该图像的图像金字塔。

也可以通过先对原始图像滤波，得到原始图像的近似图像，然后将近似图像的偶数行和偶数列删除以获取向下采样的结果。有多种滤波器可以选择。

领域滤波器：采用邻域平均值计算求原始图像的近似图像。该滤波器能够产生平均金字塔。

高斯滤波器：采用高斯滤波器对原始图像进行滤波，得到高斯金字塔。这是OpenCV函数`cv2.pyrDown()`所采用的的方式。

高斯金字塔是通过不断地使用高斯金字塔滤波、采样所产生的，其过程如下：



经过上述处理后，原始图像与各次向下采样所得到的结果图像共同构成了高斯金字塔。例如，可以将原始图像称为第0层，第1次向下采样的结果图像称为第一层、第2次向下采样的结果图像称为第2层，以此类推。

简单来说，强大的图像金字塔（拉普拉斯金字塔）可以让图片在放缩时丢失的信息没那么多。

这里我们选用高斯金字塔，每次缩小1/2，直至下一次缩小小于目标大小（28），但光这样可能还不够，可能最后一次执行完图片大小还不是28，这个时候成倍的放缩就没用了，就要使用resize()（不过此时resize丢失的信息没那么多了）

```
def adjustImg(e, target):
    while min(e.shape[0], e.shape[1]) > target:
        e = cv2.pyrDown(e, )
    e = cv2.resize(e, (target, target), interpolation=cv2.INTER_CUBIC)
    return e
```

（利用高斯金字塔逐渐收敛到目标大小）

4.1 编写用户程序

需求：用户提供一个文件夹的图片（仅支持png格式），利用已经训练好的模型，对每张图片中的手写数字进行识别。

首先要求用户提供文件夹路径（绝对路径），确保里面只有png图片，不存在其他任何格式的文件（包括子文件夹）

```
path = input("输入手写数字图片所在文件夹路径:")
```

随后读取该文件夹下所有png图片，回显给用户。（之所以这样是因为读取顺序不一定是文件夹中文件存放的顺序）

这里由于opencv在窗口中显式图片的API是阻塞的（不继续操作就不能并行的做其他操作），所以我选择开一个子线程，子线程负责显示图片，主线程用来进行计算。这样显示图片和计算预测两件事可以并行操作。

```
files = readFiles(path)
images = getImages(files)
t_show = threading.Thread(target=showImages, args=(files,))
t_show.start()
```

随后和训练时一样，保证进入模型的数据格式是对的（声明单通道，归一化）

```
img_x, img_y = images.shape[1], images.shape[2]
images = images.reshape(images.shape[0], img_x, img_y, 1)
images = images.astype('float32')
images /= 255
```

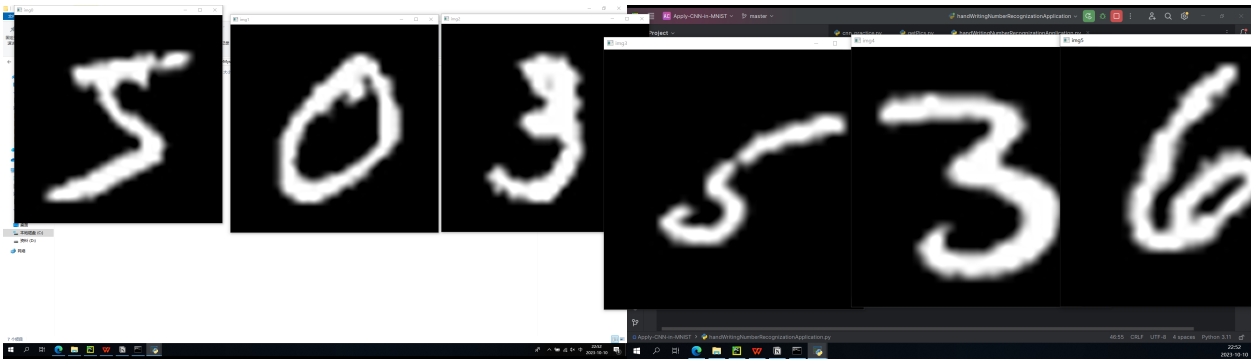
装载模型

```
model = load_model('cnn.h5')
```

预测，打印结果

```
predictions = model.predict(images)
print('predictions')
print(np.argmax(predictions, axis=1))
```

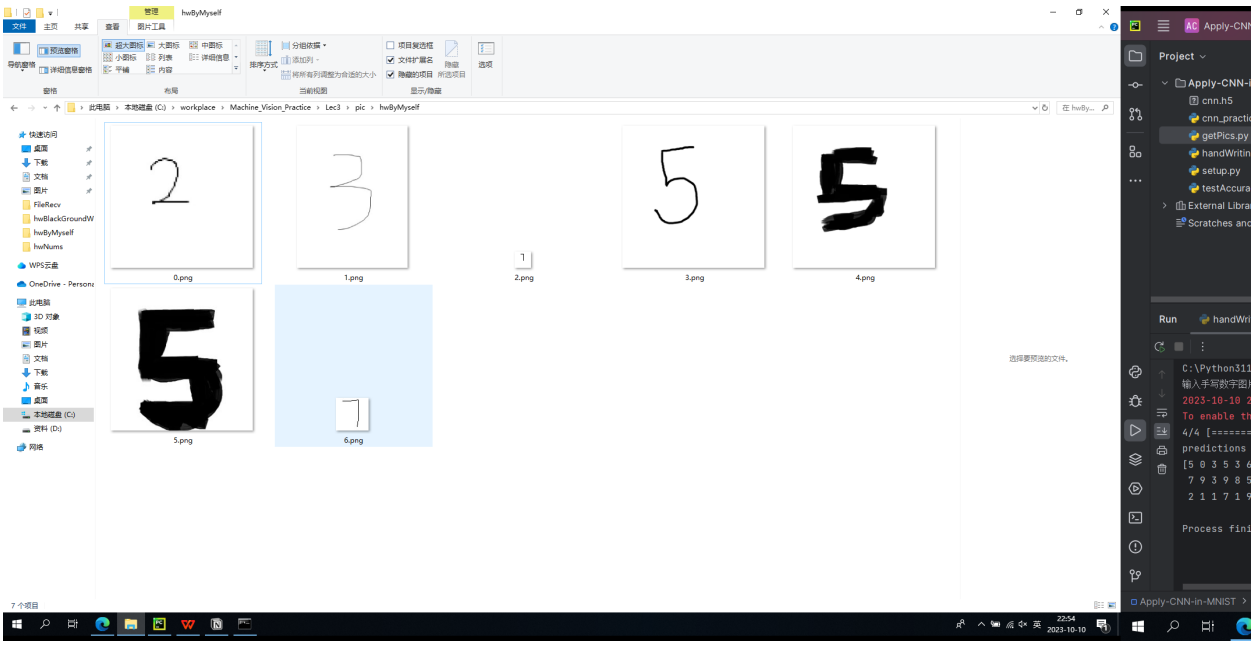
如果把100张测试集中的图片作为输入，则结果如下：



```
C:\Python311\python.exe C:\workplace\Machine_Vision_Practice\Lec3\Apply-CNN-in-MNIST\handWritingNumberRecognizationApplication.py
输入手写数字图片所在文件夹路径:C:\workplace\Machine_Vision_Practice\Lec3\pic\hwNums
2023-10-10 22:50:43.746741: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critica
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
4/4 [==============================] - 0s 13ms/step
predictions
[5 0 3 5 3 6 1 7 2 8 6 9 4 4 0 9 1 1 2 4 3 2 7 1 3 8 6 9 0 5 6 0 7 6 9 1 8
 7 9 3 9 8 5 9 3 2 3 0 7 4 9 8 0 9 4 1 1 4 4 6 0 4 5 6 1 0 0 3 1 7 1 6 3 0
 2 1 1 7 1 9 0 2 6 7 8 3 9 0 4 4 6 7 4 6 8 0 7 8 3 1]
```

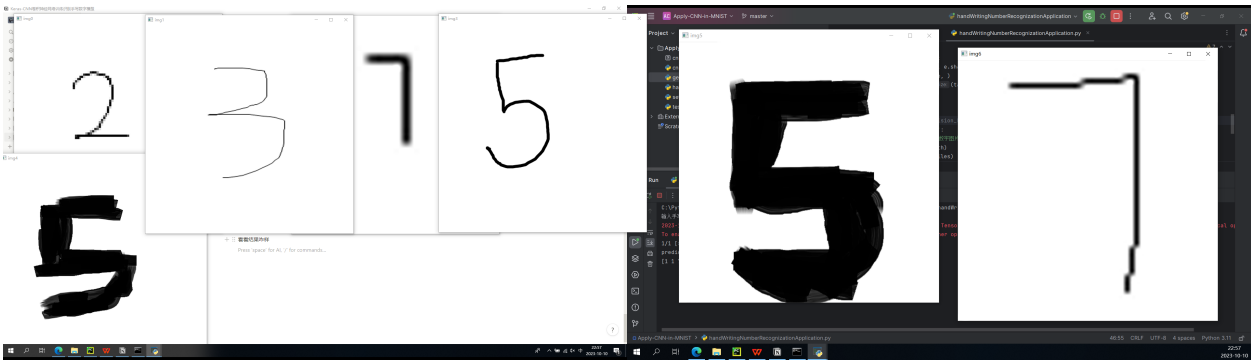
这里100个测试回显，太多了，我截了前6个，可以看到5，0，3，5，3，6，全部正确，基本上是没问题的。

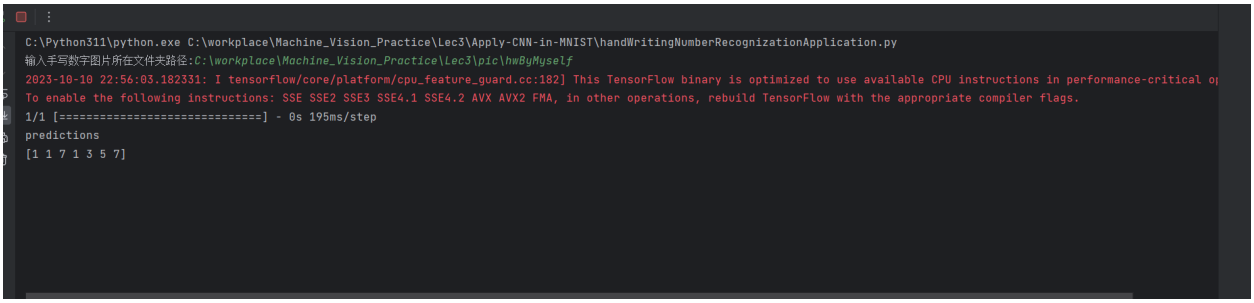
但之前也提到过，用户给的数据可能不一样。比如我自己又用画图板手写了几张图，大小不一。然后传进去看看。



从图标就能看出来，大小不太一样，最小的那个（第三个，7）是28*28的，其他大小各有不同。形状上也不同，有正方形，有长方形。

看看结果咋样



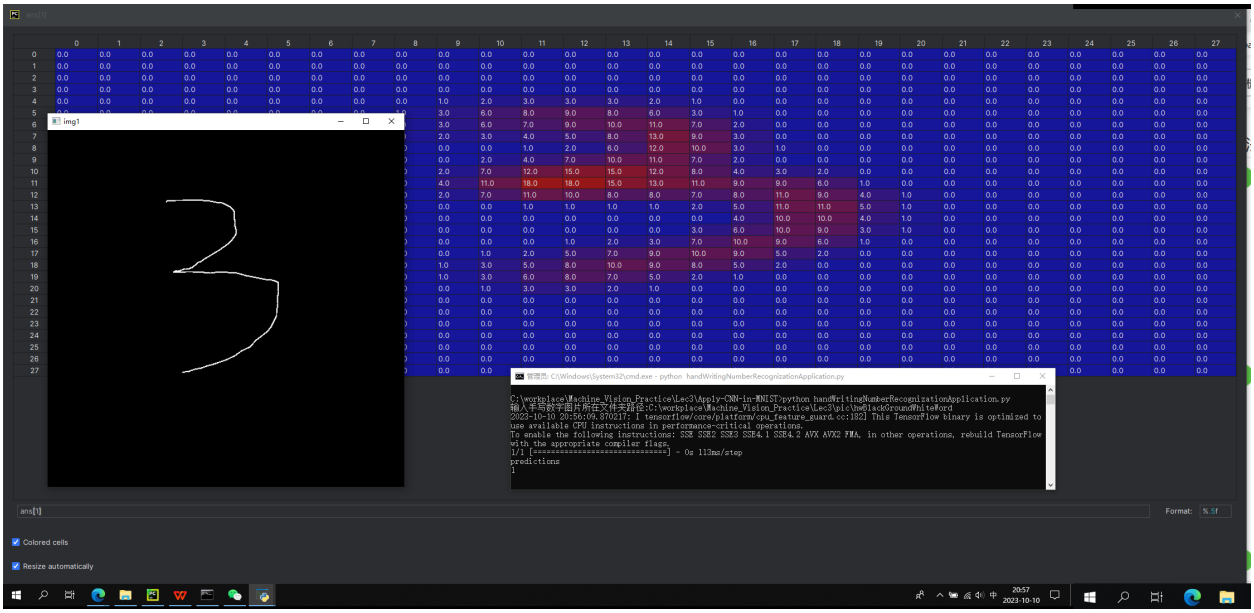


可以看到，正确结果为：2，3，7，5，5，5，7。模型预测结果为：1,1,7,1,3,5,7，命中率3/7。大概才42%。可以说很低了。

这是因为什么呢？我们可以看到，两个7的图片一个28*28，一个56*56，和28*28（也就是模型训练时用的数据）都每场接近。那么他俩在图片放缩时就不会丢掉太多关键信息，所以可以模型能识别出来。

但2这张图，笔画比较细，并且还是560*560的，在放缩时丢掉太多特征，所以就认不出来了。

再看看中间变量(这是另一组的，我用的白笔黑底，和训练集一样，但结果也不好，这说明主要问题不在字体颜色上)



可以看到，中间变量中的点阵图还是很清晰的显示出了3的，但我们的模型被训练集里那些写的非常好看的手写数字”宠坏了“，连这个比较清晰的也认不出来。

这就是过拟合（over-fitting)。我们的模型吸纳了太多训练集中的特征（60000张），以至于出了这个训练集之外的其他稍微”畸形“一点的数据都认不出来。

五、改进点

1. 正如刚才所说的，我们的模型可能因为训练集的原因过拟合了，因此我们需要修改训练集（加入更多风格不同的手写数字），或者是在合适的时机停止训练（不要吸纳过多训练集的特征），或是加入一些反例：监督不一定是正例监督，也可能是反例监督，区别度大更容易训练出泛化误差更小的模型。
2. 尝试将输入数据处理得更好。在放缩数据时，即便我采用了图像金字塔，但丢失的信息仍然很多，我尝试着搜索一些等比缩放的API，但效果都一般（resize效果比现在还差）。这个需要更多时间来发掘。
3. 更换性能更好的模型。说白了就是调参调出一个泛化能力更强的CNN手写识别模型，不仅是参数，网络架构也一样。（卷积反卷/池化反池等)