门牌号数字识别项目

摘要

在日常生活中，如何利用机器帮助我们提高图像识别处理的效率，可以避免将大量人力物力浪费在简单重复的工作当中。在机器学习发光发热之前，机器能够做大量复杂的运算，但在一些“人类”擅长的图像识别、文字翻译、交互聊天等领域，机器在很长一段时间内都无法达到普通人类的水平。而随着深度学习、强化学习算法的逐步完善；计算能力包括 CPU、GPU 和并行云计算平台的进步；以及移动互联时代传感器收集数据能力的增长；在未来，机器将在这些人类擅长的领域中逐步达到甚至超越人类的平均水平。

房间门牌号数字识别项目则是研究学习深入学习算法，应用模型识别图片中的数字序列，从而将需要投入大量人力完成的数字识别工作交给机器，提高工作效率。

一、问题定义

该项目核心的问题是训练一个模型，使它能够识别出现实图片（如街景照片等)里的数字序列。而为了完成这个项目，我们将其分为四个部分：

1、使用 mnist 数据[1]集合成类似门牌号照片的数据，建立简单的模型，使它识别合成数据中的数字序列。

2、将已建立完成的模型应用在 SVHN[2]数据集中，识别现实世界中的房屋门号。

3、在现实世界的照片数据中测试模型的效果。

4、探索一种建立图片数字定位器，提升模型的准确率。

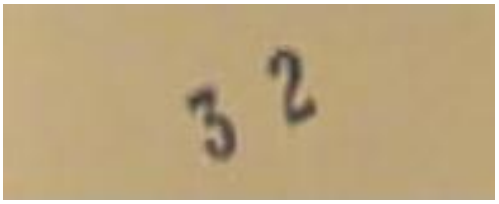
将图片数字序列是否完全正确的准确率[3]作为第 1、2、3 部分的评价指标，项目最终的目标是实现每个分类器的准确率达到 90%以上，并且能够识别真实图片中的数字序列。

将图片数字定位器预测的方框序列与图片提供的方框序列进行比较，以均方差（MSE）作为定位器的评估指标。

二、分析

（一）数据研究：主要从 SVHN 数据集入手，了解 SVHN 数据分布情况。

1、图片读取，在代码中我设置了图片读取展示部分，可以自主设置读取图片数量，并将其打印在 notebook 文件当中。

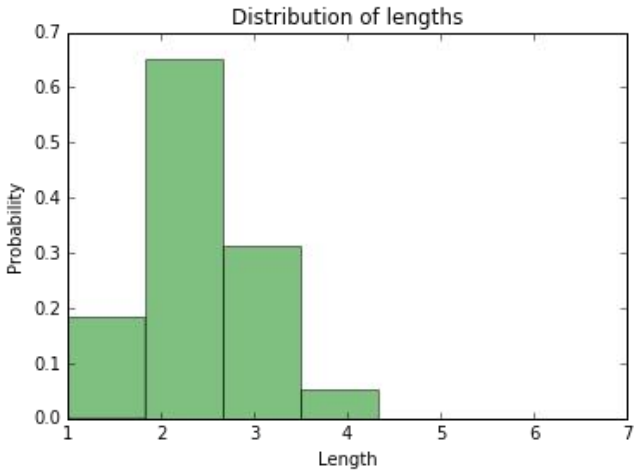


[1] Mnist 数据：一个包含 0-9 的不同手写体格式的 32*32 的标准图片数据，易于处理和识别。
[2] Street View House Numbers (SVHN): （基于谷歌街景的）一个大规模的房屋门号数据库。
[3] 图片序列准确率 = 预测完全正确的图片个数/测试数据总数

2、图片标签数据读取，在这里我们了解到标签文件将每一个图片中的每一个数字作为一条记录保存，为了提高处理效率，我们将每一张图片作为一条记录，将图片中的数字信息整合在一起，主要包括图片数字序列的值（value）、数组（difits）、长度（length）、数字部分的宽度（width）和高度（height）以及数字区域的范围（box）。随后我们进行了简单的统计，发现数据当中数字序列的最小值是 0，最大值是 135458，最小长度是 1，最大长度是 6。

而在这里我们知道,数字序列的值为 0 其实代表的是无，也就是说这很有可能是异常值，因此我们尝试将数字序列值为 0 的照片从标签信息中进行了剔除。

（二）探索性可视化：我尝试对每张图片数字序列的长度进行了统计：



从这里我们可以看出，数字序列的长度分布非常不均匀，因此最终的模型很可能会比较擅长识别 1-3 个长度序列的数字，而不太擅长识别 4 个长度以上的数字序列；因此在

设计 mnist 数据集时，我尝试从长度入手，使得数据集中不同的数字序列长度分布大致相同。

（三）算法与方法：

在数据合成部分，主要运用的方法就是循环算法，在此不做过多赘述。

在建立模型部分，我学习了 tensoflow 的教程、udacity 深度学习课程以及 CS231n 课程视频等，使用卷积神经网络来建立模型，而在最后，我尝试使用 VGG16 算法来训练模型。

在模型中主要包含了输入层、卷积层、池化层、平滑连接层、密集连接层、以及随机删除节点层、输出层、优化器等。主要的方法有：

1、权重初始化：

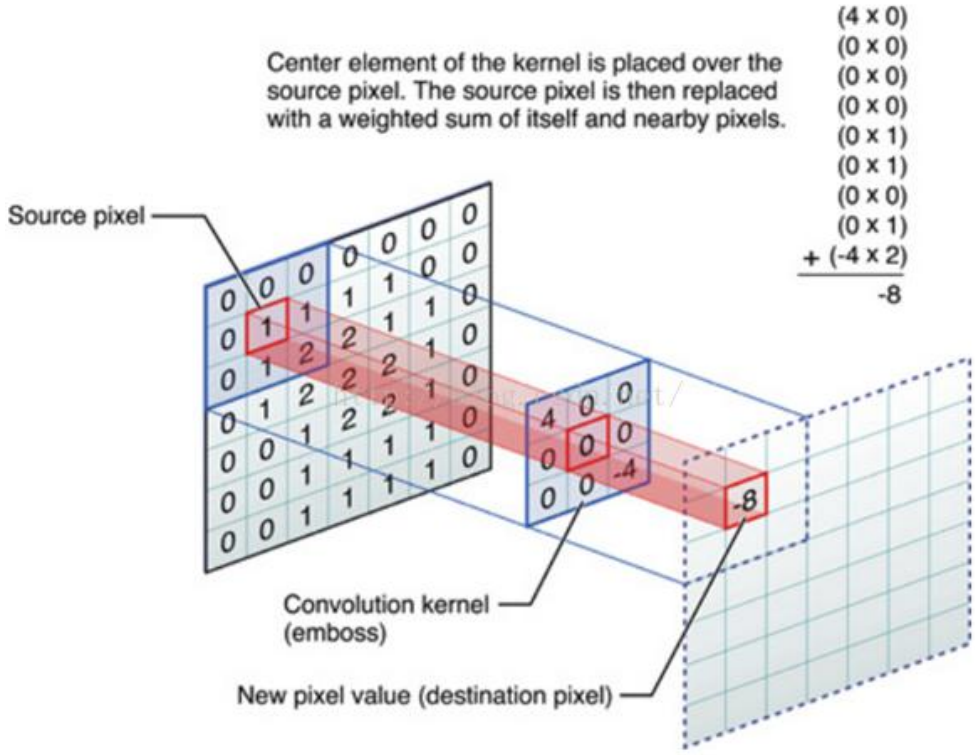
为了创建这个模型，我们需要创建大量的权重和偏置项。这个模型中的权重在初始化时应该加入少量的噪声来打破对称性以及避免 0 梯度。由于我们使用的是 ReLU 神经元，因此比较好的做法是用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为 0 的问题（dead neurons）。

该部分在 Tensorflow 中是比较重要的一个环节，但在 Keras 中，这部分则由算法自动实现，只需设置模型的输入和输出，权重初始化均由 Keras 自行完成。

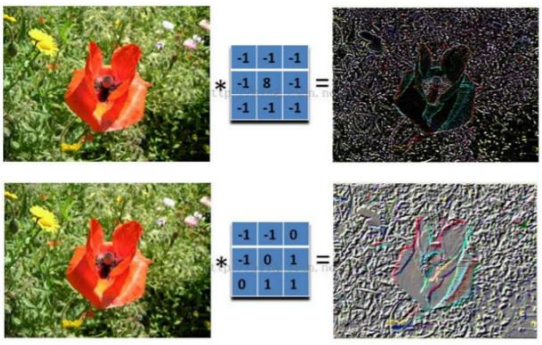
2、卷积:

TensorFlow 和 Keras 在卷积和池化上有很强的灵活性。

而卷积则是设计一个固定大小的滑窗和步长，将图片转换成层数更深，特征更加清晰的多层图片。



而一般在使用卷积层处理后，我们通过不同的卷积核过滤图片可以得到诸如颜色、轮廓等信息。



通过 Tensorflow 和 Keras 的卷积核方法，我们可以使用非常少的参数处理边界、设置步长。

3、池化:

池化方法使用用简单传统的 2x2 大小的模板做 max

pooling。即抽取 2*2 滑窗单元格中的最大值作为输出。

4、平滑连接层：这个概念在 Tensorflow 中不是特别清晰，因为使用 reshape 功能即可轻易达到相应的目的。实现的功能是将一个立体的数据压平，成为一个一维的向量。

5、密集连接层:

压平成为一维向量后，为了降低向量的长度，可以使用密集连接层，将其映射到目标变量，我们把平滑输出的张量 reshape 成一些向量，乘上权重矩阵，加上偏置，然后对其使用 ReLU。

6、随机删除节点:

为了减少过拟合，我们在输出层之前加入 dropout。这样我们可以在训练过程中启用 dropout，随机删除一些变量，强化关键因子权重；在测试过程中关闭 dropout，提高模型预测准确率。

7、输出层:

由于模型需要预测的是数字序列，因此在 SVHN 数据集中，我设置了 6 个输出层，映射到图片数字序列中的每一个数字，其中每个输出层则是一个 11 维的分类变量，包含数字 0-9 以及“空白”。在 mnist 数据集中，我则设置了 5 个输出层。最终将这些输出层组合在一起，形成一整张图片的预测结果。

8、优化器：

之前的部分主要是正向的预测，也就是包含多个隐藏层的神经网络，将一张图片映射到输出结果。而优化器则是利用反向传播算法，将输出结果的残差转变为权重的改变量，优化模型的输出结果。而在损失函数的设计方面，我则是根据映射目标选择了 categorical_crossentropy（多类的对数损失函数）和 MSE（残差平方和）。

三、实现方法

（一）数据预处理：

1、我用 github 上的处理方法[4]，将 digitStruct.mat 文件转换为.csv 格式的文件，易于使用 pandas 进行相应的数据处理和读取。

2、之后我将.csv 的标签数据进行了整合，以每张图片作为一条记录，保存了图片的数字序列值、序列数组、长度、图片宽、高以及数字区域的 box。

3、异常数字序列值处理，我剔除了数字序列值为 0 的记录。

4、边框处理，我只读取了图片中数字区域的部分，但在这里我发现其中有一些数字序列的 box 存在-1 这样的值，后来我检查了这些图片，发现这些图片中只包含了一部分数字，所以在这里我对这些图片的处理方法是：将完整的图片

[4] https://github.com/sarahrn/Py-Gsvhn-DigitStruct-Reader

读入，直接 resize 成目标数据的大小。

5、标签处理：读入标签时，我将数字序列值映射到了[6，11]的二分类变量中，其中 6 代表数字的长度，11 分别表示数字 0-9 和“空白”。

（二）实施：

我使用的是 Tensorflow，在这里我设置了比较简单的网络结构，分别是 Cov 层、Relu 层、Maxpooling 层、Cov 层、Relu 层、Maxpooling 层、Flatten 连接层、FulllyConnect 全连接层、Dropout 层、output 输出层。在这里我调试了学习速率、dropout 值、增加了正则项，但模型在单个标签上的准确率最高只达到了 78%。

```
# define the variables
layer1_weights = tf.Variable(tf.truncated_normal(
    [patch_size, patch_size, num_channels, depth1], stddev = 0.1))
layer1_biases = tf.Variable(tf.zeros([depth1]))
layer2_weights = tf.Variable(tf.truncated_normal(
    [patch_size, patch_size, depth1, depth2], stddev = 0.1))
layer2_biases = tf.Variable(tf.constant(1.0, shape = [depth2]))
layer3_weights = tf.Variable(tf.truncated_normal(
    [image_size / 4 * image_size / 4 * depth2, num_hidden], stddev = 0.1))
layer3_biases = tf.Variable(tf.constant(1.0, shape = [num_hidden]))
keep_prob = tf.placeholder("float")
layer4_weights = tf.Variable(tf.truncated_normal(
    [num_hidden, num_labels], stddev = 0.1))
layer4_biases = tf.Variable(tf.constant(1.0, shape = [num_labels]))

#model
def model_train(data):
    #cov1
    cov = tf.nn.conv2d(data, layer1_weights, [1,1,1,1], padding='SAME')
    hidden = tf.nn.relu(cov + layer1_biases)
    hpool = tf.nn.max_pool(hidden, ksize= [1,2,2,1], strides = [1,2,2,1], padding='SAME')
    #cov2
    cov = tf.nn.conv2d(hpool, layer2_weights, [1,1,1,1], padding='SAME')
    hidden = tf.nn.relu(cov + layer2_biases)
    hpool = tf.nn.max_pool(hidden, ksize= [1,2,2,1], strides = [1,2,2,1], padding='SAME')
    #hidden3
    shape = hpool.get_shape().as_list()
    h_pool_flat = tf.reshape(hpool, [shape[0], shape[1] * shape[2] * shape[3]])
    h_fc = tf.nn.relu(tf.matmul(h_pool_flat, layer3_weights) + layer3_biases)
    h_fc_drop = tf.nn.dropout(h_fc, keep_prob)
    return tf.matmul(h_fc_drop, layer4_weights) + layer4_biases
```

（三）改进：

当我尝试了多种参数组合之后，我发现，模型本身的准确率还是很低，于是我开始尝试增加模型的层数和卷积核

数。但是由于电脑配置较低，我开始在 aws 上使用 Keras 提高模型训练效率、代码编写效率，增加模型复杂度。aws 服务器提供的更高性能的 GPU 计算服务显著地提高了模型训练速度，同时 Keras 可以使用非常少的代码达到与 Tensorflow 相同的效果。经过调试，我增加了现有模型的复杂程度，最终的模型架构如下：

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	(None, 128, 128, 3)	0	
convolution2d_21 (Convolution2D)	(None, 64, 64, 8)	224	input_6[0][0]
convolution2d_22 (Convolution2D)	(None, 32, 32, 16)	1168	convolution2d_21[0][0]
maxpooling2d_11 (MaxPooling2D)	(None, 16, 16, 16)	0	convolution2d_22[0][0]
convolution2d_23 (Convolution2D)	(None, 8, 8, 32)	4640	maxpooling2d_11[0][0]
convolution2d_24 (Convolution2D)	(None, 4, 4, 64)	18496	convolution2d_23[0][0]
maxpooling2d_12 (MaxPooling2D)	(None, 4, 4, 64)	0	convolution2d_24[0][0]
flatten_6 (Flatten)	(None, 1024)	0	maxpooling2d_12[0][0]
dense_25 (Dense)	(None, 2048)	2099200	flatten_6[0][0]
dropout_13 (Dropout)	(None, 2048)	0	dense_25[0][0]
dense_26 (Dense)	(None, 128)	262272	dropout_13[0][0]
dense_27 (Dense)	(None, 128)	262272	dropout_13[0][0]
dense_28 (Dense)	(None, 128)	262272	dropout_13[0][0]
dense_29 (Dense)	(None, 128)	262272	dropout_13[0][0]
dense_30 (Dense)	(None, 128)	262272	dropout_13[0][0]
dense_31 (Dense)	(None, 128)	262272	dropout_13[0][0]
output0 (Dense)	(None, 11)	1419	dense_26[0][0]
output1 (Dense)	(None, 11)	1419	dense_27[0][0]
output2 (Dense)	(None, 11)	1419	dense_28[0][0]
output3 (Dense)	(None, 11)	1419	dense_29[0][0]
output4 (Dense)	(None, 11)	1419	dense_30[0][0]
output5 (Dense)	(None, 11)	1419	dense_31[0][0]

Total params: 3705874

最终在在训练时，因为增加了核的数量，模型在 mnist 数据集上表现很好，但在 SVHN 上效果虽然有了一定的提升，但仍然不理想，在预测现实世界的照片时，模型完全错误。因此我开始尝试使用 VGG16 的模型架构。

最终，经过调试，我减少了 VGG16 模型中核的数量，使得模型的准确率高了很多，同时收敛的速度比 VGG16 也快了

很多。但是这时模型却出现了过拟合的状态，于是我在全连接层上加入了 dropout 和正则项来避免过拟合的状态。

具体的模型架构如下：

Layer (type)	Output Shape	Param #	Connected to
input_5 (InputLayer)	(None, 64, 64, 1)	0	
block1_conv1 (Convolution2D)	(None, 64, 64, 8)	80	input_5[0][0]
block1_conv2 (Convolution2D)	(None, 64, 64, 8)	584	block1_conv1[0][0]
block1_pool (MaxPooling2D)	(None, 32, 32, 8)	0	block1_conv2[0][0]
block2_conv1 (Convolution2D)	(None, 32, 32, 16)	1168	block1_pool[0][0]
block2_conv2 (Convolution2D)	(None, 32, 32, 16)	2320	block2_conv1[0][0]
block2_pool (MaxPooling2D)	(None, 16, 16, 16)	0	block2_conv2[0][0]
block3_conv1 (Convolution2D)	(None, 16, 16, 32)	4640	block2_pool[0][0]
block3_conv2 (Convolution2D)	(None, 16, 16, 32)	9248	block3_conv1[0][0]
block3_conv3 (Convolution2D)	(None, 16, 16, 32)	9248	block3_conv2[0][0]
block3_pool (MaxPooling2D)	(None, 8, 8, 32)	0	block3_conv3[0][0]
block4_conv1 (Convolution2D)	(None, 8, 8, 64)	18496	block3_pool[0][0]
block4_conv2 (Convolution2D)	(None, 8, 8, 64)	36928	block4_conv1[0][0]
block4_conv3 (Convolution2D)	(None, 8, 8, 64)	36928	block4_conv2[0][0]
block4_pool (MaxPooling2D)	(None, 4, 4, 64)	0	block4_conv3[0][0]
block5_conv1 (Convolution2D)	(None, 4, 4, 64)	36928	block4_pool[0][0]
block5_conv2 (Convolution2D)	(None, 4, 4, 64)	36928	block5_conv1[0][0]
block5_conv3 (Convolution2D)	(None, 4, 4, 64)	36928	block5_conv2[0][0]
block5_pool (MaxPooling2D)	(None, 2, 2, 64)	0	block5_conv3[0][0]
flatten (Flatten)	(None, 256)	0	block5_pool[0][0]
drop1 (Dropout)	(None, 256)	0	flatten[0][0]
fc1 (Dense)	(None, 2048)	526336	drop1[0][0]
drop2 (Dropout)	(None, 2048)	0	fc1[0][0]
fc2 (Dense)	(None, 1024)	2098176	drop2[0][0]
drop3 (Dropout)	(None, 1024)	0	fc2[0][0]
output0 (Dense)	(None, 11)	11275	drop3[0][0]
output1 (Dense)	(None, 11)	11275	drop3[0][0]
output2 (Dense)	(None, 11)	11275	drop3[0][0]
output3 (Dense)	(None, 11)	11275	drop3[0][0]

四、结果

在不经裁剪的图片上使用最初建立的简单模型时，分类器达到的准确率只有 40%左右，最终完整预测的准确率也只

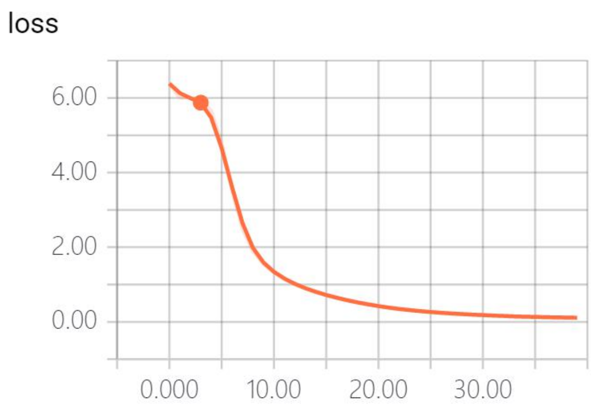
有 24%；而使用调整后的 VGG16 模型时，每个标签的准确率在验证集上到了 90%以上，满足了问题定义要求的结果：

```
Epoch 40/40
61s - loss: 0.1102 - output0_loss: 0.0414 - output1_loss: 0.0388 - output2_loss: 0.0239 - output3_loss: 0.0061 - output4_loss: 1.1864e-05 -
 output5_loss: 1.0095e-05 - output0_acc: 0.9873 - output1_acc: 0.9883 - output2_acc: 0.9925 - output3_acc: 0.9979 - output4_acc: 1.0000 - ou
tput5_acc: 1.0000 - val_loss: 1.1961 - val_output0_loss: 0.4076 - val_output1_loss: 0.4304 - val_output2_loss: 0.2862 - val_output3_loss: 0.
0694 - val_output4_loss: 0.0025 - val_output5_loss: 2.3041e-06 - val_output0_acc: 0.9169 - val_output1_acc: 0.9100 - val_output2_acc: 0.9409
- val_output3_acc: 0.9838 - val_output4_acc: 0.9997 - val_output5_acc: 1.0000
```

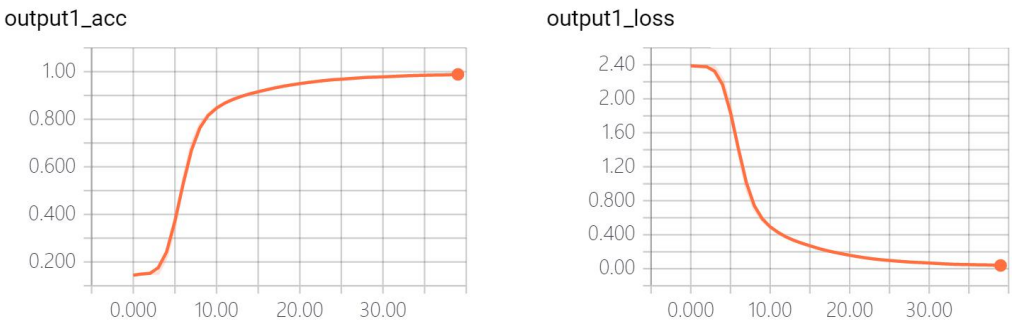
每个标签在测试数据上达到了同样的准确率：

```
: model.evaluate(svhn_test_dataset, svhn_test_labels, batch_size=128, verbose=1)

13054/13054 [==============================] - 11s

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

: [1.1854113487037927,
 0.42977210276613048,
 0.52588554140124988,
 0.19781394102054878,
 0.030239000523628077,
 0.0016981303877912858,
 2.6292863184436709e-06,
 0.91581124590570828,
 0.90401409560694923,
 0.95886318428293205,
 0.99463765924733394,
 0.99984679140649938,
 1.0000000011506336]
```

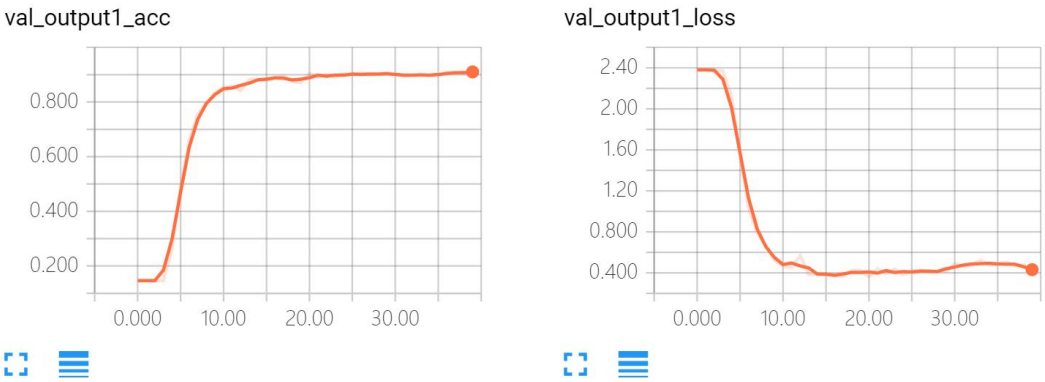
根据训练情况，我们可以看到损失函数的变化情况如下：



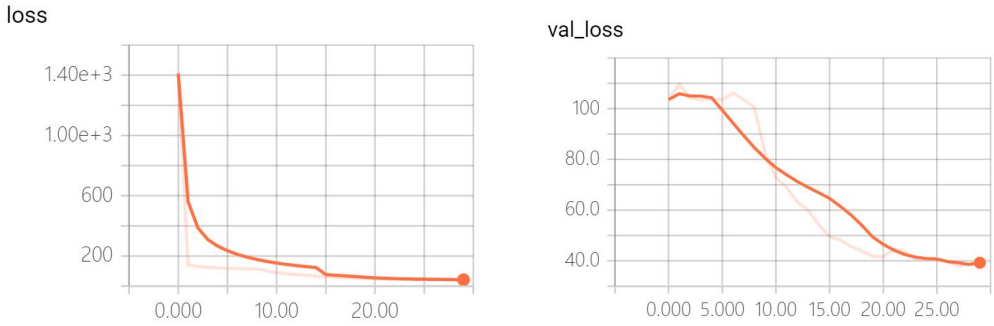
Training_dataset 中第一个分类变量的 acc 和 loss 变化如下：



Validation_dataset 中第一个分类变量的 acc 和 loss 变化如下：



在图片定位器上，模型的 MSE 为：



而在输出预测结果后，我将其按照每张图片进行了比对，在 Mnist 模拟数据集上，模型的准确率达到了 94.69%；在 SVHN 真实数据上，模型的准确率达到了 82.53%。使用同样的模型架构，在测试图片定位器上 MSE 为：88.57

在 SVHN 数据集上，我们可以看到，模型已经出现了过

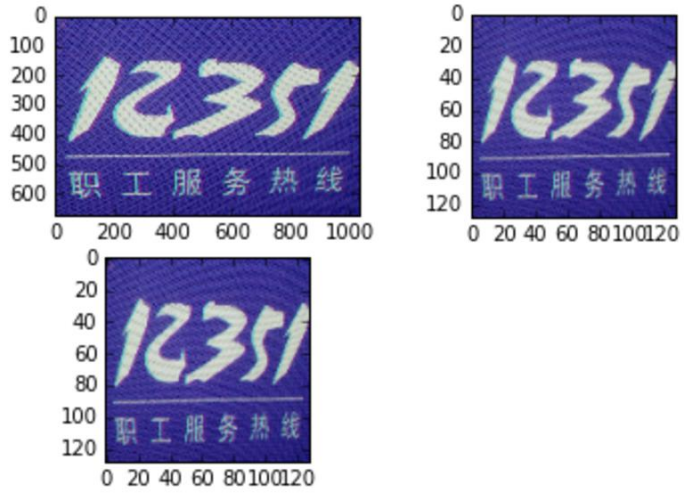
拟合的状态，但整体的 loss 已经在一个非常低的水平了。因此我想到，如果需要继续提升模型的泛化能力，则可以尝试使用 SVHN 中的 EXTRA 数据集，增加卷积层的核数量等方法。

五、结论

（一）真实照片数据预测：

项目中我在外面拍摄了 5 张照片，将其处理后输入模型，模型准确识别了前四张照片，而最后一张出现了错误：

```
('The prediction is ', [4, 3, 5, 4, 10, 10])
```



我认为错误的原因是 SVHN 数据当中，长度为 5 的数字序列数据较少，模型无法有效的预测。我认为引入数据量更大的 extra 数据集或者将增加长度为 4-6 的数字序列图片的训练次数可以提高模型的准确率。

（二）边框定位器：

由于在步骤二时，我已经对图片进行了切割，只提取了包含数字序列部分的图片，因此定位器在 SVHN 测试数据上

表现相对较好，但在真实数据上却表现不好。如果能做出单个数字的定位器，那么边框定位器的准确率或许还会有一定的提升。

（三）改进的思路，我认为如果能够先做一个比较好的单个字符定位器，将图片中的数字或者字符全部提取出来，之后再输入到数字识别模型当中，这样可以减少数字序列、长度不同带来的困扰，然后根据不同的位置，将数字或者字符以此输出，这样模型可以实现的就不只是房门号的识别。定位器和图像识别应用的结合，甚至可以是人类的第三只眼睛，这对视力不好甚至视力障碍的人都会有很大的帮助。

（四）最后的一个小疑问：

当我按照审阅结果再次训练模型时，我发现尝试了很多次之后，模型都无法像之前尝试的那样收敛；但是在这之前，我曾重复试验了 3-4 次，每个分类器都达到了 90%以上的准确率；我在想是不是因为随机设置卷积核的原因，使得模型在某些情况下收敛的速度非常快，但在某些情况下收敛的比较慢；而这个情况在目前是否能够有效地解决呢？

六、引用：

1、卷积神经网络 http://blog.csdn.net/v_july_v/article/details/51812459

2、tensorflow 中文网站：http://tensorfly.cn/