# 实验四 基于 RNN—LSTM+CTC 的注册码识别实践

## 1 实验内容

使用 FreeType 库生成图片注册码，并将图片转彩色，加上点噪声和线噪声后生成最终的样本集，并通过 LSTM+CTC 的网络结构完成识别。

## 2 工作流程

FreeType 库生成注册码、图片转彩色加点线噪声生成数据集→图片灰度化、二值化和去噪预处理→定义 lstm 网络结构→模型训练(使用 ctc loss) →测试

## 3 关键代码解析

实验的注册码图片和标签由 freetype 库生成的。

```
#根据生成的 text，生成 image,返回标签和图片元素数据
def gen_image(self):
    text,vec = self.random_text()
    img = np.zeros([32,256,3])
    #生成彩色图像
    color_ = (0,0,255) # Write
    pos = (0, 0)
    text_size = 21
    image = self.ft.draw_text(img, pos, text, text_size, color_)
    return image[:,:,2],text,vec
```

转为 RGBA 图像，加上点、线噪声：

```
image, text, vec = obj.gen_image()
#转为 RGBA 图像
img = Image.fromarray(image, mode='RGBA')
color_ = random_color(0, 128)  # , random.randint(220, 250)
create_noise_dots(img, color_)
create_noise_curve(img, color_)
```

图片灰度化和二值化去噪处理：

```
img = img.convert('L')
```

```
image = img.convert('1')
image = np.array(image)
```

网络模型由 lstm+一个全连接层组成。lstm 网络拥有 1 个隐藏层和 64 个神经元的，BasicLSTMCell 中 num_units 这个参数的大小就是 LSTM 输出结果的维度。原网络定义的时候其实是考虑了 52 个大小写字母+0~9 十个数字+空格+ctcblank=64 这样来设计的。经过预处理后的图像是 32x256 像素的，按时序输入的原理将图片拆成 256 个长为 32 的一维向量送入 lstm cell 中。256*32 的图片输入经历 lstm 后输出为 256*64。全连接层有 12 个神经元， Logits 是网络最终输出的结果,形状为 256*batchsize*12，存放的是 batchsize 张图片 256 个像素列向量对应 0~9 十个数字+空格+ctcblank12 个字符的概率值。Targetsctc_loss 需要的稀疏矩阵，存放的是标签值。seq_len 长为 batchsize，存放的是每张图片的时序长度 256。

```
def get_train_model():
    # OUTPUT_SHAPE = (32,256)
    inputs = tf.placeholder(tf.float32, [None, None, OUTPUT_SHAPE[0]])
    #定义 ctc_loss 需要的稀疏矩阵
    targets = tf.sparse_placeholder(tf.int32)
    #1 维向量 序列长度 [batch_size,]
    seq_len = tf.placeholder(tf.int32, [None])
    #定义 LSTM 网络
    cell = tf.contrib.rnn.LSTMCell(num_hidden, state_is_tuple=True)
    outputs, _ = tf.nn.dynamic_rnn(cell, inputs, seq_len, dtype=tf.float32)
    shape = tf.shape(inputs)
    batch_s, max_timesteps = shape[0], shape[1]
    outputs = tf.reshape(outputs, [-1, num_hidden])
    W = tf.Variable(tf.truncated_normal([num_hidden,
                                         num_classes],  # num_classes = 12
                                        stddev=0.1), name="W")
    b = tf.Variable(tf.constant(0., shape=[num_classes]), name="b")
    logits = tf.matmul(outputs, W) + b
    logits = tf.reshape(logits, [batch_s, -1, num_classes])
    logits = tf.transpose(logits, (1, 0, 2))
    return logits, inputs, targets, seq_len, W, b
```

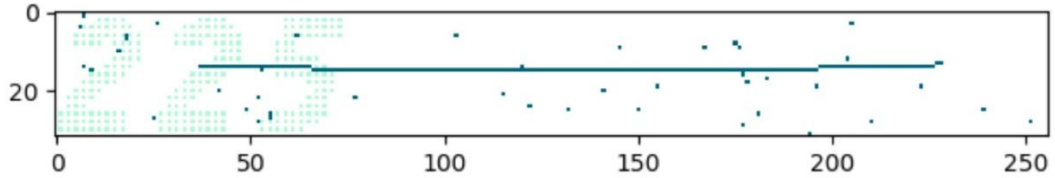
训练模型时使用的学习率是指数衰减学习率，损失函数是 CTCloss，参数的优化是使用的 Adam 梯度下降方法。将模型的输出结果 logits、标签系数矩阵 target 和 seq_len 传入 ctc_loss 方法中，CTC 的损失函数将从模型中预测出来的划分出来的 256 个块对应于每个数字字符的类属概率分布，进行极大似然估计算出标签序列的概率。ctc 的优化过程是算最大似然，CTC 的计算包含一个 softmax output layer， 而且也会多一个 label (blank)。准确率 acc 是用预测值和标签值的最小编辑距离（Edit Distance, ED）的均方差来衡量的。

```
def train():
    global_step = tf.Variable(0, trainable=False)
    #指数衰减学习率
    learning_rate = tf.train.exponential_decay(INITIAL_LEARNING_RATE,
                                               global_step,
                                               DECAY_STEPS,
                                               LEARNING_RATE_DECAY_FACTOR,
                                               staircase=True)
    logits, inputs, targets, seq_len, W, b = get_train_model()
    loss = tf.nn.ctc_loss(labels=targets, inputs=logits,
sequence_length=seq_len)
    cost = tf.reduce_mean(loss)
    #Adam 梯度下降
    optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss,globa
l_step=global_step)
    decoded, log_prob = tf.nn.ctc_beam_search_decoder(logits, seq_len,
merge_repeated=False)
    acc = tf.reduce_mean(tf.edit_distance(tf.cast(decoded[0], tf.int32),
targets))
    init = tf.global_variables_initializer()

    def do_report():
        test_inputs, test_targets, test_seq_len =
get_next_batch(BATCH_SIZE)
        test_feed = {inputs: test_inputs,
                     targets: test_targets,
                     seq_len: test_seq_len}
        dd, log_probs, accuracy = session.run([decoded[0], log_prob,
acc], test_feed)
        return report_accuracy(dd, test_targets)
        # decoded_list = decode_sparse_tensor(dd)

    def do_batch():
        train_inputs, train_targets, train_seq_len =
get_next_batch(BATCH_SIZE)
        feed = {inputs: train_inputs, targets: train_targets, seq_len:
train_seq_len}
        b_loss, b_targets, b_logits, b_seq_len, b_cost, steps, _ =
session.run([loss, targets, logits, seq_len, cost, global_step,
optimizer], feed)

        print(b_cost, steps)
        if steps > 0 and steps % REPORT_STEPS == 0:
```

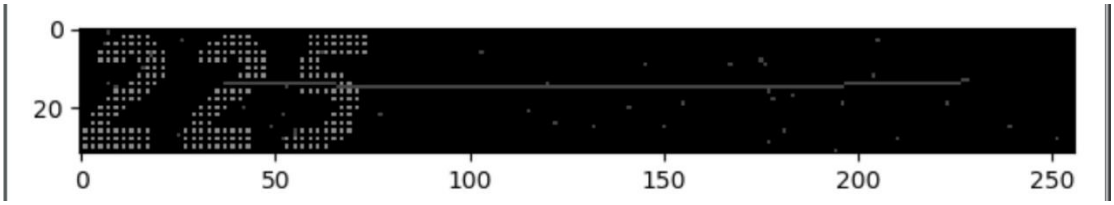
```
            if(do_report()>0.9):
                save_path = saver.save(session, "./ocr.model",
global_step=steps)
                print(save_path)
                raise FError("Train succcess")
        return b_cost, steps

    with tf.Session() as session:
        session.run(init)
        saver = tf.train.Saver(tf.global_variables(), max_to_keep=100)
        for curr_epoch in range(num_epochs):
            print("Epoch.......", curr_epoch)
            train_cost = train_ler = 0
            for batch in range(BATCHES):
                start = time.time()
                c, steps = do_batch()
                train_cost += c * BATCH_SIZE
                seconds = time.time() - start
                print("Step:", steps, ", batch seconds:", seconds)

            train_cost /= TRAIN_SIZE

            train_inputs, train_targets, train_seq_len =
get_next_batch(BATCH_SIZE)
            val_feed = {inputs: train_inputs,
                        targets: train_targets,
                        seq_len: train_seq_len}

            val_cost, val_ler, lr, steps = session.run([cost, acc,
learning_rate, global_step], feed_dict=val_feed)
            log = "Epoch {}/{}, steps = {}, train_cost = {:.3f}, train_ler =
{:.3f}, val_cost = {:.3f}, val_ler = {:.3f}, time = {:.3f}s, learning_rate
= {}"
            print(log.format(curr_epoch + 1, num_epochs, steps, train_cost,
train_ler, val_cost, val_ler, time.time() - start, lr))
```

# 4 实验结果

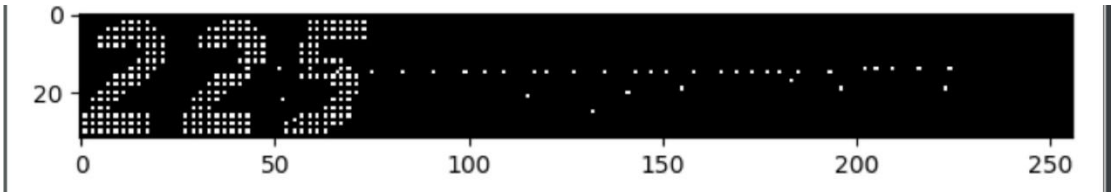
这是经过彩色和加点、线噪声处理后的注册码图片：



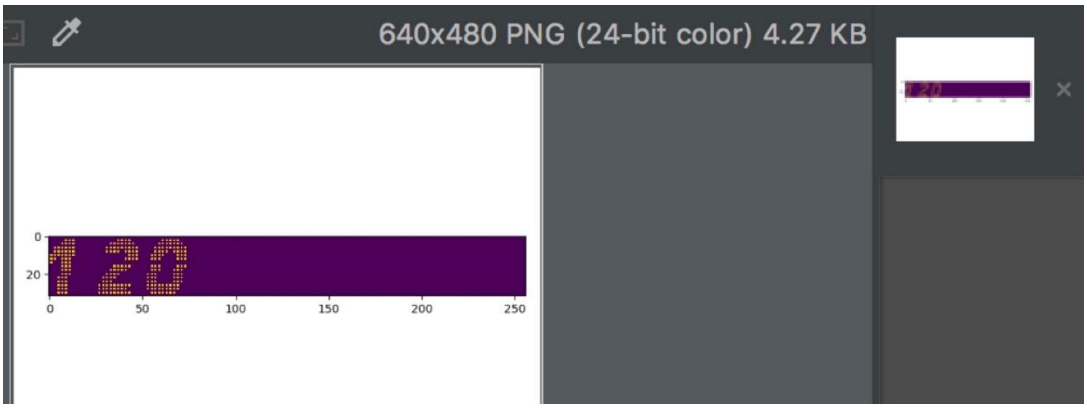
经过灰度化处理：




二值化处理：




模型预测结果如下图所示：




```
[[[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 1. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]]
[['3', '6', '7']]
INFO:tensorflow:Restoring parameters from ./ocr.model-6800
T/F: original(length) <-------> detectcted(length)
True ['1', '2', '0'] ( 3 ) <------->  ['1', '2', '0'] ( 3 )
Test Accuracy: 1.0
```

# 4 总结与理解

本次实验注册码识别问题，像素序列对应不确定长度的字符序列也是一种输入特征和输出标签之间对齐关系不确定的时间序列问题。实验中生成的注册码图片是 32 x 256 像素的，将图片按列切分为 256 个 32 长度的一维向量按照时序送入 lstm 中，这么做的原因是什么呢，切分后的每一列并不能完整地识别出一个数字字符，最终是多列组成一个数字，前面的内容会对后面的内容的判别产生影响。所以我们采用长短时记忆网络来训练，记忆先前的计算结果，协同判别后序

输入的类属概率。最终再经过一个全连接后我们得出的结果形状是batchsize*256*12 的，应对与 12 个标签。另外就是为什么要将系列列表处理为稀疏矩阵，一方面是对于真实标签矩阵，它的每一位对应每一个字符只会有一栏为 1，其他的会为 0，尤其是在还有字母和识别中文等情况下，矩阵就会有许多栏目为 0，十分稀疏，浪费空间。另一部分，这个输出也是为了呼应 ctc_loss 方法的输入要求。

```
def ctc_loss(labels, inputs, sequence_length,
                preprocess_collapse_repeated=False,
                ctc_merge_repeated=True,
                ignore_longer_outputs_than_inputs=False, time_major=True):
loss = tf.nn.ctc_loss(labels=targets, inputs=logits, sequence_length=seq_len)
```

CTC 的损失函数会从模型中预测出来的划分出来的 256 个块对应于每个数字字符的类属概率分布，进行极大似然估计算出标签序列的概率。CTC 过程是算最大似然，其数定义如下所示:

$$L(S) = -ln \prod_{(x,z)\in S} p(z|x) = -\sum_{(x,z)\in S} lnp(z|x)$$

其中 p(z|x)代表给定输入 x，输出序列 zz 的概率，S 为训练集。

CTC 的损失函数中还用到了最小编辑距离（预测值与标签值的最小编辑次数）。TC 的计算包含一个 softmax output layer， CTC 网络的 softmax 输出层输出的类别有字符长度+1 种，因为有一个 ctc blank 分隔符，这个分隔符很好地解决了 label 和 output 不对齐的情况中区分哪些子串是属于同一个文字的图片区域的输出结果的问题。