2018202112 第二次 报告

丁逸凡 2018202112

上期我们跑通了代码, 这段时间我们的注意力放在读代码, 学习模型及模型背后的原理上

代码大致结构

在图像中进行特征提取, 用CNN是比较commom的做法

这是构建CNN网络的部分, 可以选择两种网络结构

```
    def build_cnn(self):
        """ Build the CNN. """
        print("Building the CNN...")
        if self.config.cnn == 'vgg16':
            self.build_vgg16()
        else:
            self.build_resnet50()
        print("CNN built.")
```

如果不显示指定vgg16网络结构, Resnet50为例

```
    def build_resnet50(self):
        """ Build the ResNet50. """
        config = self.config

        images = tf.placeholder(
            dtype = tf.float32,
            shape = [config.batch_size] + self.image_shape)

        conv1_feats = self.nn.conv2d(images,
                                     filters = 64,
                                     kernel_size = (7, 7),
                                     strides = (2, 2),
                                     activation = None,
                                     name = 'conv1')
        conv1_feats = self.nn.batch_norm(conv1_feats, 'bn_conv1')
        conv1_feats = tf.nn.relu(conv1_feats)
        pool1_feats = self.nn.max_pool2d(conv1_feats,
                                         pool_size = (3, 3),
                                         strides = (2, 2),
                                         name = 'pool1')

        res2a_feats = self.resnet_block(pool1_feats, 'res2a', 'bn2a', 64,
1)
        res2b_feats = self.resnet_block2(res2a_feats, 'res2b', 'bn2b', 64)
        res2c_feats = self.resnet_block2(res2b_feats, 'res2c', 'bn2c', 64)

        res3a_feats = self.resnet_block(res2c_feats, 'res3a', 'bn3a', 128)
        res3b_feats = self.resnet_block2(res3a_feats, 'res3b', 'bn3b', 128)
        res3c_feats = self.resnet_block2(res3b_feats, 'res3c', 'bn3c', 128)
        res3d_feats = self.resnet_block2(res3c_feats, 'res3d', 'bn3d', 128)
```

```
30
31          res4a_feats = self.resnet_block(res3d_feats, 'res4a', 'bn4a', 256)
32          res4b_feats = self.resnet_block2(res4a_feats, 'res4b', 'bn4b', 256)
33          res4c_feats = self.resnet_block2(res4b_feats, 'res4c', 'bn4c', 256)
34          res4d_feats = self.resnet_block2(res4c_feats, 'res4d', 'bn4d', 256)
35          res4e_feats = self.resnet_block2(res4d_feats, 'res4e', 'bn4e', 256)
36          res4f_feats = self.resnet_block2(res4e_feats, 'res4f', 'bn4f', 256)
37
38          res5a_feats = self.resnet_block(res4f_feats, 'res5a', 'bn5a', 512)
39          res5b_feats = self.resnet_block2(res5a_feats, 'res5b', 'bn5b', 512)
40          res5c_feats = self.resnet_block2(res5b_feats, 'res5c', 'bn5c', 512)
41
42          reshaped_res5c_feats = tf.reshape(res5c_feats,
43                                            [config.batch_size, 49, 2048])
44
45          self.conv_feats = reshaped_res5c_feats
46          self.num_ctx = 49
47          self.dim_ctx = 2048
48          self.images = images
```

这样的网络结构是按照论文中的方式实现的, 可以对照着看

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

其中`resnet_block()`的实现是

```
1     def resnet_block(self, inputs, name1, name2, c, s=2):
2         """ A basic block of ResNet. """
3         branch1_feats = self.nn.conv2d(inputs,
4                                        filters = 4*c,
5                                        kernel_size = (1, 1),
6                                        strides = (s, s),
7                                        activation = None,
```

```
                                    use_bias = False,
                                    name = name1+'_branch1')
        branch1_feats = self.nn.batch_norm(branch1_feats, name2+'_branch1')

        branch2a_feats = self.nn.conv2d(inputs,
                                    filters = c,
                                    kernel_size = (1, 1),
                                    strides = (s, s),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2a')
        branch2a_feats = self.nn.batch_norm(branch2a_feats,
name2+'_branch2a')
        branch2a_feats = tf.nn.relu(branch2a_feats)

        branch2b_feats = self.nn.conv2d(branch2a_feats,
                                    filters = c,
                                    kernel_size = (3, 3),
                                    strides = (1, 1),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2b')
        branch2b_feats = self.nn.batch_norm(branch2b_feats,
name2+'_branch2b')
        branch2b_feats = tf.nn.relu(branch2b_feats)

        branch2c_feats = self.nn.conv2d(branch2b_feats,
                                    filters = 4*c,
                                    kernel_size = (1, 1),
                                    strides = (1, 1),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2c')
        branch2c_feats = self.nn.batch_norm(branch2c_feats,
name2+'_branch2c')

        outputs = branch1_feats + branch2c_feats
        outputs = tf.nn.relu(outputs)
        return outputs
```

`resnet_block2()` 的实现是

```
 def resnet_block2(self, inputs, name1, name2, c):
        """ Another basic block of ResNet. """
        branch2a_feats = self.nn.conv2d(inputs,
                                    filters = c,
                                    kernel_size = (1, 1),
                                    strides = (1, 1),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2a')
        branch2a_feats = self.nn.batch_norm(branch2a_feats,
name2+'_branch2a')
        branch2a_feats = tf.nn.relu(branch2a_feats)

        branch2b_feats = self.nn.conv2d(branch2a_feats,
                                    filters = c,
```

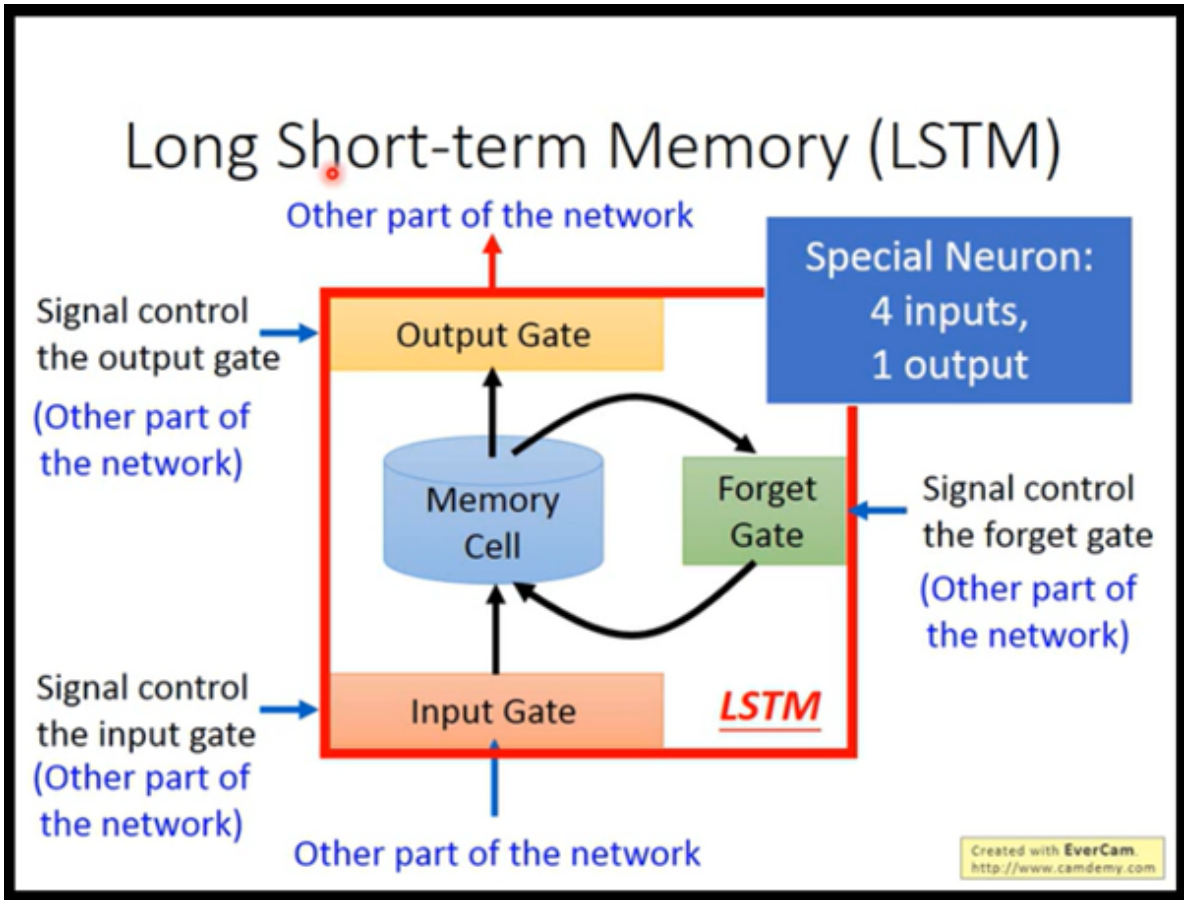
```
                                    kernel_size = (3, 3),
                                    strides = (1, 1),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2b')
        branch2b_feats = self.nn.batch_norm(branch2b_feats,
name2+'_branch2b')
        branch2b_feats = tf.nn.relu(branch2b_feats)

        branch2c_feats = self.nn.conv2d(branch2b_feats,
                                    filters = 4*c,
                                    kernel_size = (1, 1),
                                    strides = (1, 1),
                                    activation = None,
                                    use_bias = False,
                                    name = name1+'_branch2c')
        branch2c_feats = self.nn.batch_norm(branch2c_feats,
name2+'_branch2c')

        outputs = inputs + branch2c_feats
        outputs = tf.nn.relu(outputs)
        return outputs
```

可以看到outputs和inputs要么间隔一次卷积+normalization要么直接相连, 能缓解了梯度消失的问题

文本生成用LSTM是比较常见的做法, 我是这么理解的, 因为单词在文本中的语义要结合上下文出现的单词来看, 直觉上就很容易想到带有"记忆力"的模型会有更好的效果; 但是过于长期的"记忆"也是有害的, 很难想象一个单词或者句子的语义要和相距甚远的另一个句子或者单词联系起来. 所有Long Short-term Memory是最好的



这张图就能很好地解释LSTM网络中一个"cell"对应的相关结构和机制. 注意, 这里的input, 控制输入, 控制遗忘(其实说是控制记忆更好些), 控制输出的各个部分, 都需要有对应的矩阵, 这就导致了要学习的参数非常多, 网络很复杂, 计算量非常庞大. 这点我们在实践中有很深的认识, 跑这个部分用了3天时间

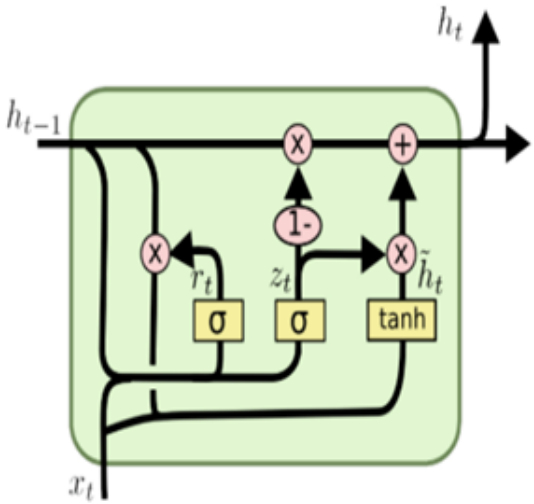
当然结果还不错:

a group of people sitting around a table.

a double decker bus driving down a street.

a man riding a wave on top of a surfboard.

下一步的改进目标就是改LSTM为GRU, GRU中cell的结构如下图所示



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

https://blog.csdn.net/weixin_424210

从直观上我们可以看到, 与LSTM相比, GRU的门数量减少了, 相应地需要学习的参数的数目也变小了, 这样应该能减少运行的时间.