子颢 (/users/223413) 2018-01-22 11:38:41 (最初创作于: 2018-01-21 13:50:14) 发表于： 智能服务事业部 (/teams/259) >> 算法实践 (/teams/259?cid=1195)

知识体系： 修改知识体系
文章标签： CNN (/search?q=CNN&type=INSIDE_ARTICLE_TAG) Attention (/search?q=Attention&type=INSIDE_ARTICLE_TAG) Hierarchical (/search?q=Hierarchical&type=INSIDE_ARTICLE_TAG) 修改标签 标签历史 (/articles/98876/tags/history)
附加属性： 内部资料请勿外传 **作者原创**

基于上下文的文本分类

背景

在以往的文本分类案例中，大多数都是先将文本做Flatten，然后用CNN、RNN或者传统的机器学习算法进行文本分类，虽然也能达到较好的效果，但是这样并不能很好的利用到词与词之间、句与句之间的ngram上下文信息。

训练数据集说明

文本的平均长度为100个中文字符，每个文本平均包含10个句子，每句大约10个分词。
一共大约50万条训练数据，30个类别标签。
训练数据统一进行了预处理，包括样本均衡、数字归一化、停用词过滤等。

建模过程

文本Flatten

本次试验以文本Flatten作为baseline，并分别对比了softmax Regression、DNN和CNN的分类结果。

softmax Regression

softmax Regression实际上是logistic Regression的变种，模型输入特征分别用了bag of word和TF-TDF，但是试验结果相差并不大。
运行了20个epoch差不多已经收敛，batch_size设为256，实验结果准确率只有80%。

```
# 三个待输入的数据
        self.input_x = tf.placeholder(tf.float32, [None, self.config.vocab_size], name='input_x')
        self.input_y = tf.placeholder(tf.float32, [None, self.config.num_classes], name='input_y')
        W = tf.Variable(tf.truncated_normal([self.config.vocab_size, self.config.num_classes], stddev=0.1))
        b = tf.Variable(tf.constant(0.1, shape=[self.config.num_classes]))

        with tf.name_scope("score"):
            y_conv = tf.matmul(self.input_x, W) + b
            self.y_pred_cls = tf.argmax(y_conv, 1)  # 预测类别

        with tf.name_scope("optimize"):
            self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=self.input_y, logits=y_conv
            # 优化器
            self.optim = tf.train.AdamOptimizer(learning_rate=self.config.learning_rate).minimize(self.loss)

        with tf.name_scope("accuracy"):
            # 准确率
            correct_pred = tf.equal(tf.argmax(self.input_y, 1), self.y_pred_cls)
            self.acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

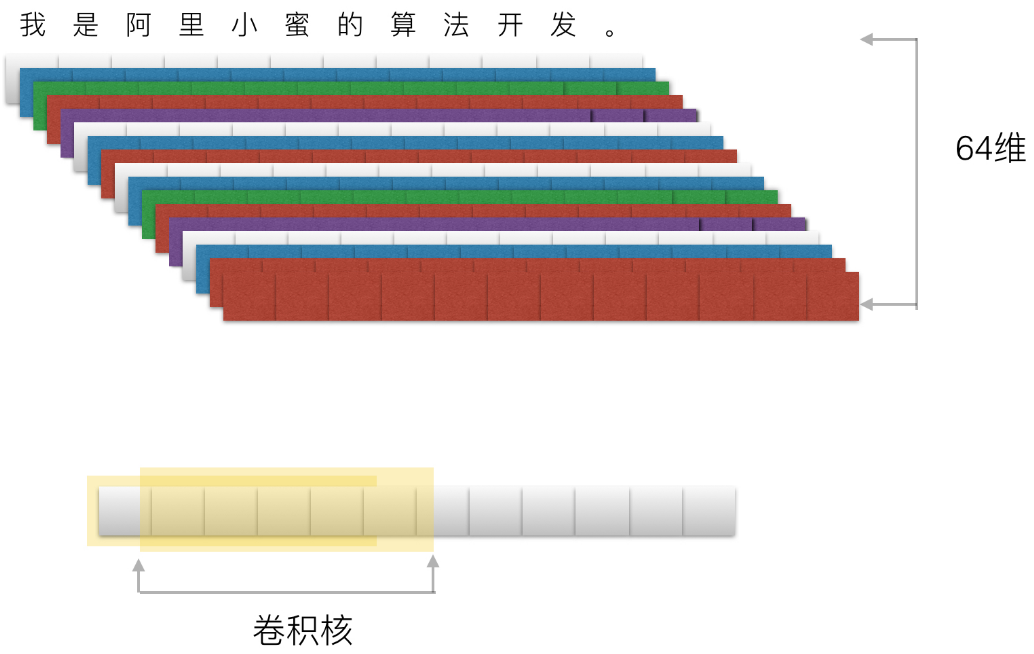
DNN

DNN模型直接用bag of word作为输入特征。
一个隐藏层，隐藏层神经元个数为100，激活函数Relu，学习率1e-3，并且运行200个iteration，准确率为86.7%。

```
## TRAIN
logreg = neural_network.MLPClassifier() # 其实使用sk-learn工具包只需要这一句代码
logreg.fit(train_feature, train_target)
## PREDICT
test_predict = logreg.predict(test_feature)
## ACCURACY
true_false = (test_predict == test_target)
accuracy = np.count_nonzero(true_false)/float(len(test_target))
print("\naccuracy is %f" % accuracy)
```

一维CNN

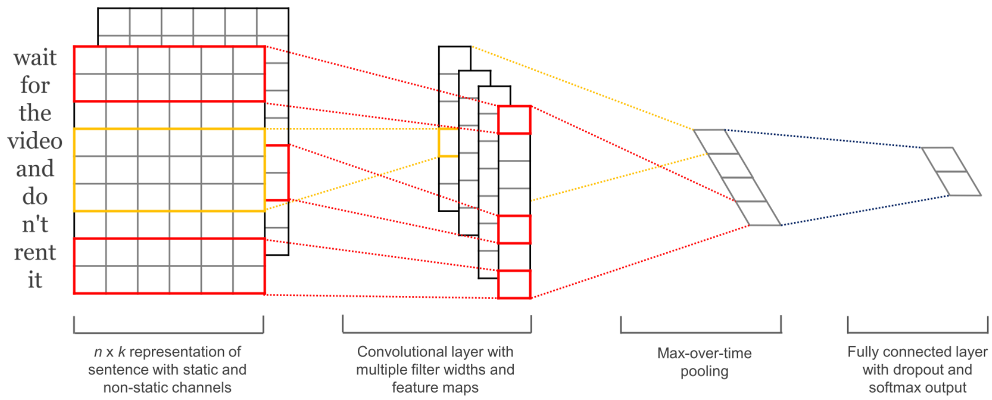
CNN模型采用了两种不同的卷积策略，一种是横向卷积，一种是纵向卷积，并且都以word embedding作为模型输入特征，维度为64（实验中也使用了预训练的词向量进行迁移学习，效果并不明显）。
纵向卷积：之所以称之为纵向卷积，是因为这种卷积方式实际上是将word embedding的维度作为输入通道的数目，卷积核大小为5的一维卷积。



(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/3ccdf0fbb6bb33945c79718ee765bac6.png)

```
    # 三个待输入的数据
        self.input_x = tf.placeholder(tf.int32, [None, self.config.seq_length], name='input_x')
        self.input_y = tf.placeholder(tf.float32, [None, self.config.num_classes], name='input_y')
        self.keep_prob = tf.placeholder(tf.float32, name='keep_prob')
        self.cnn()

    def cnn(self):
        """CNN模型"""
        # 词向量映射
        embedding = tf.get_variable('embedding', [self.config.vocab_size, self.config.embedding_dim])
        embedding_inputs = tf.nn.embedding_lookup(embedding, self.input_x)

        with tf.name_scope("cnn"):
            # CNN layer:输入为100*64, 64为input_channels
            # conv = tf.layers.conv1d(embedding_inputs, self.config.num_filters, self.config.kernel_size, name='c
            filter_w = tf.Variable(tf.truncated_normal([self.config.kernel_size, self.config.embedding_dim, self.
            conv = tf.nn.conv1d(embedding_inputs, filter_w, 1, padding='SAME')
            print(conv)
            # global max pooling layer
            gmp = tf.reduce_max(conv, reduction_indices=[1], name='gmp')
            print(gmp)

        with tf.name_scope("score"):
            # 全连接层，后面接dropout以及relu激活
            fc = tf.layers.dense(gmp, self.config.hidden_dim, name='fc1')
            fc = tf.contrib.layers.dropout(fc, self.keep_prob)
            fc = tf.nn.relu(fc)

            # 分类器
            self.logits = tf.layers.dense(fc, self.config.num_classes, name='fc2')
            self.y_pred_cls = tf.argmax(self.logits, 1)  # 预测类别

        with tf.name_scope("optimize"):
            # 损失函数，交叉熵
            cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=self.logits, labels=self.input_y)
            self.loss = tf.reduce_mean(cross_entropy)
            # 优化器
            self.optim = tf.train.AdamOptimizer(learning_rate=self.config.learning_rate).minimize(self.loss)

        with tf.name_scope("accuracy"):
            # 准确率
            correct_pred = tf.equal(tf.argmax(self.input_y, 1), self.y_pred_cls)
            self.acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

横向卷积：其实最常用的是这种卷积方式，卷积核大小为2 * 64的二维卷积，如下图所示。



(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/8786a5eb87f268c58195b2d8bf2fc36b.png)http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-06-at-8.03.47-AM.png (http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-06-at-8.03.47-AM.png)

```
filter_w_1 = tf.Variable(tf.truncated_normal([2, 64, 1, output_channel], stddev=0.1))
filter_b_1 = tf.Variable(tf.constant(0.1, shape=[output_channel]))
conv_1 = tf.nn.relu(tf.nn.conv2d(input, filter_w_1, strides=[1, 1, 1, 1], padding='VALID') + filter_b_1)
```

实验结果：横向卷积和纵向卷积效果相差无几，准确率约为86%。

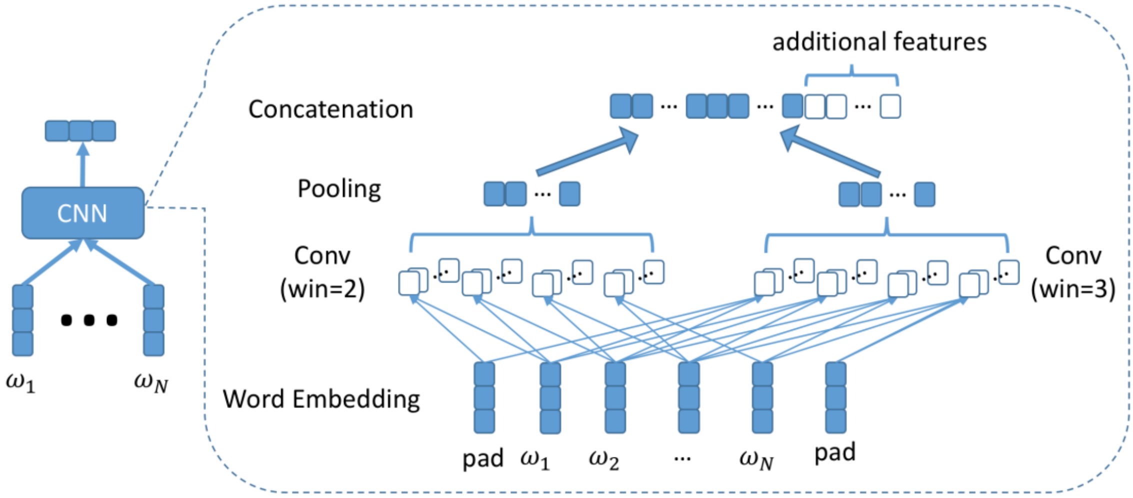
ngram CNN

虽然文本Flatten已经能达到比较不错的效果，但是它并不能很好的利用到词与词之间、句与句之间的ngram上下文信息，而这种上下文信息其实对分类结果至关重要。
ngram CNN模型的思路也很简单，先将长度为100的文本reshape成10 * 10的矩阵，分别表示10个句子，每个句子10个分词，不足则都padding 0，然后分别用ngram进行纵向二维卷积。
实验证明这种方式达到了state of art的结果，准确率为89%，并且在GPU上的训练效率非常高。

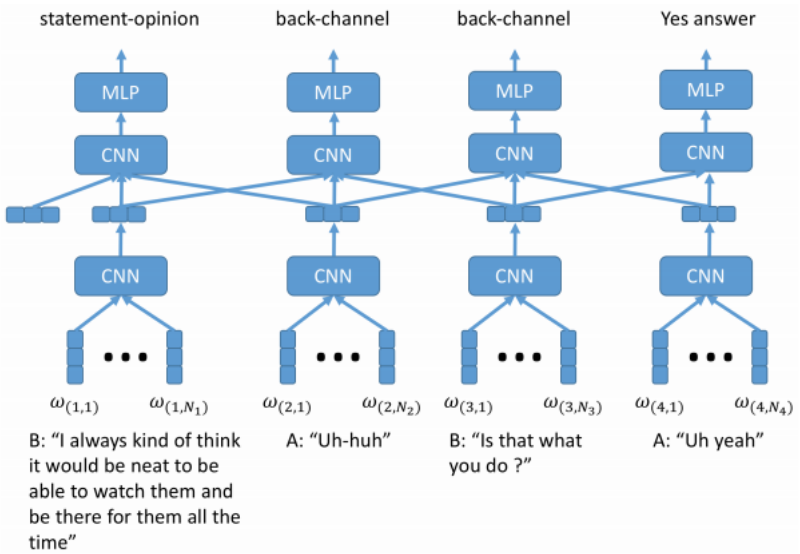
```
def __init__(self, config):
        self.config = config

        # 三个待输入的数据
        self.input_x = tf.placeholder(tf.int32, [None, self.config.text_length, self.config.sentence_length], nam
        self.input_y = tf.placeholder(tf.float32, [None, self.config.num_classes], name='input_y')
        self.keep_prob = tf.placeholder(tf.float32, name='keep_prob')

        self.cnn()

    def cnn(self):
        """CNN模型"""
        # 词向量映射
        embedding = tf.get_variable('embedding', [self.config.vocab_size, self.config.embedding_dim])
        embedding_inputs = tf.nn.embedding_lookup(embedding, self.input_x)
        print(embedding_inputs)

        def conv(gram):
            filter_w_1 = tf.Variable(tf.truncated_normal([gram, gram, 64, 128], stddev=0.1))
            filter_b_1 = tf.Variable(tf.constant(0.1, shape=[128]))
            conv_1 = tf.nn.conv2d(embedding_inputs, filter_w_1, strides=[1, 1, 1, 1], padding='SAME') + filter_b_
            h_conv_1 = tf.nn.relu(conv_1)
            h_pool_1 = tf.nn.max_pool(h_conv_1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
            h_pool1_flat1 = tf.reduce_max(h_pool_1, axis=1)
            h_pool1_flat2 = tf.reduce_max(h_pool1_flat1, axis=1)
            return tf.reshape(h_pool1_flat2, [-1, 128])

        with tf.name_scope("1-gram"):
            flaten_1 = conv(1)

        with tf.name_scope("2-gram"):
            flaten_2 = conv(2)

        with tf.name_scope("3-gram"):
            flaten_3 = conv(3)

        with tf.name_scope("4-gram"):
            flaten_4 = conv(4)

        with tf.name_scope("score"):
            # 全连接层，后面接dropout以及relu激活
            W_fc1 = tf.Variable(tf.truncated_normal([4 * 128, 256], stddev=0.1))
            b_fc1 = tf.Variable(tf.constant(0.1, shape=[256]))

            h_pool1 = tf.concat([flaten_1, flaten_2, flaten_3, flaten_4], 1) # 列上做concat

            # h_pool_flat = tf.reshape(h_pool1, [-1, 1 * 128])
            h_fc1 = tf.nn.relu(tf.matmul(h_pool1, W_fc1) + b_fc1)

            # keep_prob = tf.placeholder(tf.float32)
            h_fc1_drop = tf.nn.dropout(h_fc1, self.keep_prob)

            # 分类器
            ............
```

Hierarchical CNN

在ngram CNN的基础上，我又尝试了新的模型HCNN，参考了一篇不太引人注目的论文http://aclweb.org/anthology/D17-1231，但是同样达到了不错的效果，准确率88.55%。 (http://aclweb.org/anthology/D17-1231%EF%BC%8C%E4%BD%86%E6%98%AF%E5%90%8C%E6%A0%B7%E8%BE%BE%E5%88%B0%E4%BA%86%E4%B8%8D%E9%94%99%E7%9A%84%E6%95%88%E6%9E%9C%EF%BC%8C%E5%87%86%E7%A1%AE%E7%8E%8788.55%%E3%80%82)
模型原理也同样简单，首先在句子层上进行一次CNN，得到每个句子的representation，然后在此基础上再进行一次CNN，最后经过全连接层进行文本分类。



(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/5a153eb01e9c5d72eef9a4e9a7c5d2bd.png)



(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/1cf250bab6b00c32ea4e3ce808af6a1d.png)

```
# 词向量映射
        embedding = tf.get_variable('embedding', [self.config.vocab_size, self.config.embedding_dim])
        embedding_inputs = tf.nn.embedding_lookup(embedding, self.input_x)

        def conv(gram, input, dim, input_channel, output_channel):
            filter_w_1 = tf.Variable(tf.truncated_normal([gram, dim, input_channel, output_channel], stddev=0.1))
            filter_b_1 = tf.Variable(tf.constant(0.1, shape=[output_channel]))
            conv_1 = tf.nn.relu(tf.nn.conv2d(input, filter_w_1, strides=[1, 1, 1, 1], padding='VALID') + filter_b
            reduce_1 = tf.reduce_max(conv_1, axis=1)
            return tf.reshape(reduce_1, [-1, output_channel])

        with tf.name_scope("hcnn"):
            # 1、单词级卷积
            # reshape为[batch_size * sent_in_doc, word_in_sent, embedding_size]
            embedding_inputs_word = tf.reshape(embedding_inputs, [-1, self.config.sentence_length, self.config.em
            # 输入shape：[batch * 20, 20, self.config.embedding_dim, 1]
            embedding_inputs_ex = tf.expand_dims(embedding_inputs_word, -1)
            conv_word_1 = tf.expand_dims(conv(1, embedding_inputs_ex, self.config.embedding_dim, 1, 64), -1)
            conv_word_2 = tf.expand_dims(conv(2, embedding_inputs_ex, self.config.embedding_dim, 1, 64), -1)
            conv_word_3 = tf.expand_dims(conv(3, embedding_inputs_ex, self.config.embedding_dim, 1, 64), -1)
            conv_word_4 = tf.expand_dims(conv(4, embedding_inputs_ex, self.config.embedding_dim, 1, 64), -1)
            concat_word = tf.concat([conv_word_1, conv_word_2, conv_word_3, conv_word_4], 2) # shape为[None, 128,
            print(concat_word)

            # 2、句子级卷积
            # 将输入还原为[batch, 20, 128, 4]
            sent_input = tf.reshape(concat_word, [-1, self.config.text_length, 64, 4])
            print(sent_input)
            conv_sent_1 = conv(1, sent_input, 64, 4, 64)
            conv_sent_2 = conv(2, sent_input, 64, 4, 64)
            conv_sent_3 = conv(3, sent_input, 64, 4, 64)
            conv_sent_4 = conv(4, sent_input, 64, 4, 64)
            concat_sent = tf.concat([conv_sent_1, conv_sent_2, conv_sent_3, conv_sent_4], 1)  # shape为[None, 128
            print(concat_sent)

        with tf.name_scope("score"):
            # 全连接层，后面接dropout以及relu激活
            fc = tf.layers.dense(concat_sent, self.config.hidden_dim, name='fc1')
            fc = tf.contrib.layers.dropout(fc, self.keep_prob)
            fc = tf.nn.relu(fc)

            # 分类器
            ............
```

Hierarchical Attention CNN

后来我又在HCNN模型的基础上加入了Attention机制，同样参考了另外一篇论文 https://www.cs.cmu.edu/~diyiy/docs/naacl16.pdf，最终效果并不十分理想，准确率只有86.66%。(https://www.cs.cmu.edu/%7Ediyiy/docs/naacl16.pdf%EF%BC%8C%E6%9C%80%E7%BB%88%E6%95%88%E6%9E%9C%E5%B9%B6%E4%B8%8D%E5%8D%81%E5%88%86%E7%90%86%E6%83%B3%EF%BC%8C%E5%87%86%E7%A1%AE%E7%8E%87%E5%8F%AA%E6%9C%8986.66%%E3%80%82)

模型的原理有点复杂，但并不难理解。首先在句子层进行了一次bi-LSTM，得到每个句子的representation，然后进行句子层的self-Attention，再在文本层针对所有句子representation再进行一次bi-LSTM和self-Attention，最后加上一层全连接层进行文本分类。

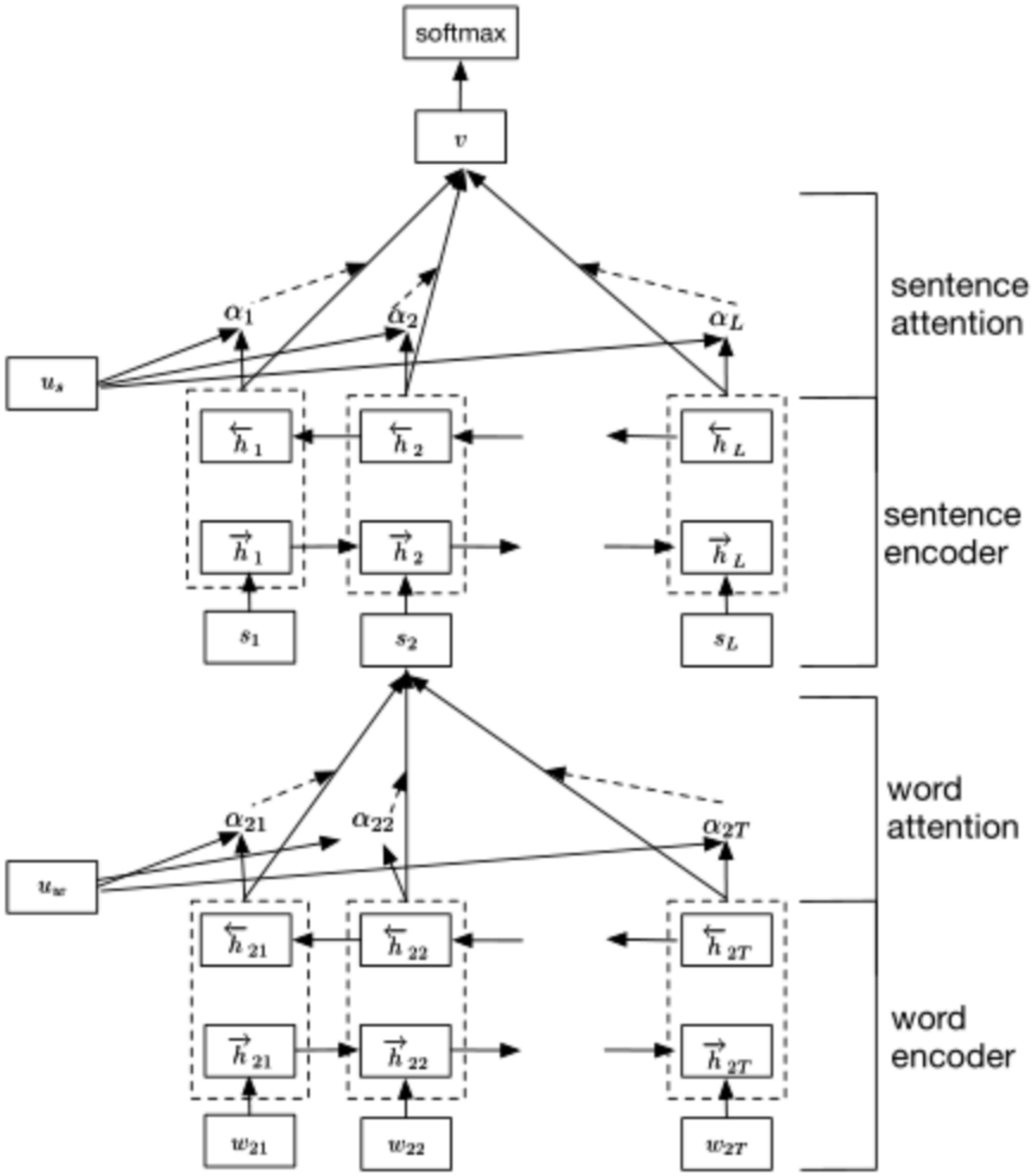


Figure 2: Hierarchical Attention Network.

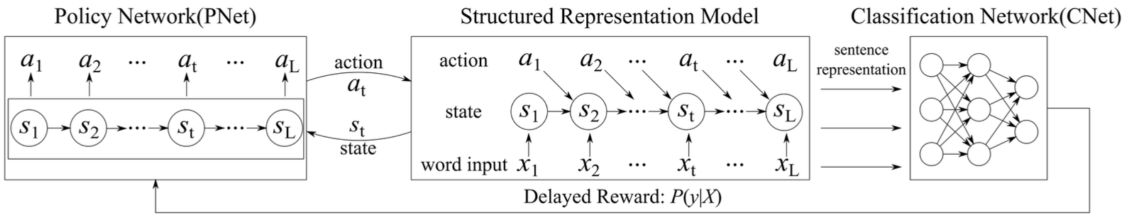
(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/31594a21a65cc8246ab8969585dfbf80.png)
我在论文的基础上进行了一点小改动，即把bi-LSTM那部分全部换成了CNN，因为考虑到RNN网络的效率以及上线成本问题。

```
def hacnn(self):
        """Hierarchical Attention CNN模型"""
        # 词向量映射
        embedding = tf.get_variable('embedding', [self.config.vocab_size, self.config.embedding_dim])
        embedding_inputs = tf.nn.embedding_lookup(embedding, self.input_x)

        def conv(gram, input, dim, input_channel, output_channel):
            filter_w_1 = tf.Variable(tf.truncated_normal([gram, dim, input_channel, output_channel], stddev=0.1))
            filter_b_1 = tf.Variable(tf.constant(0.1, shape=[output_channel]))
            conv_1 = tf.nn.relu(tf.nn.conv2d(input, filter_w_1, strides=[1, 1, 1, 1], padding='VALID') + filter_b
            reduce_1 = tf.reduce_max(conv_1, axis=1)
            return tf.reshape(reduce_1, [-1, output_channel])

        with tf.name_scope("hacnn"):
            # 1、单词级卷积+Attention
            # reshape为[batch_size * sent_in_doc, word_in_sent, embedding_size]
            embedding_inputs_word = tf.reshape(embedding_inputs, [-1, self.config.sentence_length, self.config.em
            # 输入shape：[batch * 20, 20, self.config.embedding_dim, 1]
            embedding_inputs_ex = tf.expand_dims(embedding_inputs_word, -1)
            conv_word_1 = conv(1, embedding_inputs_ex, self.config.embedding_dim, 1, 64)
            conv_word_2 = conv(2, embedding_inputs_ex, self.config.embedding_dim, 1, 64)
            conv_word_3 = conv(3, embedding_inputs_ex, self.config.embedding_dim, 1, 64)
            conv_word_4 = conv(4, embedding_inputs_ex, self.config.embedding_dim, 1, 64)
            concat_word = tf.concat([conv_word_1, conv_word_2, conv_word_3, conv_word_4], 1) # shape为[None, 64 *
            print(concat_word)
            # self Attention Layer
            fc_word = tf.nn.tanh(tf.layers.dense(concat_word, 64 * 4))
            print(fc_word)
            u_context = tf.Variable(tf.truncated_normal([64 * 4], stddev=0.1))
            alpha_word = tf.nn.softmax(tf.multiply(fc_word, u_context))
            # 拓展alpha为(?, 64 * 4)
            # alpha_word = self.extend_alpha(alpha_word)
            print(alpha_word)
            atten_w_output = tf.multiply(concat_word, alpha_word)

            # 2、句子级卷积+Attention
            # 将输入还原为[batch, 20, 4 * 64]
            sent_vec = tf.reshape(atten_w_output, [-1, self.config.text_length, 256])
            # 得到句子的向量表示：带alpha权重的Word Average，shape为[batch, 20, 4 * 64]
            print(sent_vec)
            sent_vec_ex = tf.expand_dims(sent_vec, -1)
            conv_sent_1 = conv(1, sent_vec_ex, 256, 1, 64)
            conv_sent_2 = conv(2, sent_vec_ex, 256, 1, 64)
            conv_sent_3 = conv(3, sent_vec_ex, 256, 1, 64)
            conv_sent_4 = conv(4, sent_vec_ex, 256, 1, 64)
            concat_sent = tf.concat([conv_sent_1, conv_sent_2, conv_sent_3, conv_sent_4], 1) # shape为[None, 64 *
            print(concat_sent)
            # self Attention layer
            fc_sent = tf.nn.tanh(tf.layers.dense(concat_sent, 64 * 4))
            u_sent_context = tf.Variable(tf.truncated_normal([64 * 4], stddev=0.1))
            alpha_sent = tf.nn.softmax(tf.multiply(fc_sent, u_sent_context))
            # alpha_sent = self.extend_alpha(alpha_sent, 0.8)
            atten_s_output = tf.multiply(concat_sent, alpha_sent)  # [batch, 4 * 64]

        with tf.name_scope("score"):
            # 全连接层，后面接dropout以及relu激活
            # 得到文章的向量表示：带alpha权重的Sentence Average，shape为[batch, 64]
            # text_vec = tf.reduce_sum(atten_s_output, axis=1)
            fc = tf.layers.dense(atten_s_output, self.config.hidden_dim, name='fc1')
            fc = tf.contrib.layers.dropout(fc, self.keep_prob)
            fc = tf.nn.relu(fc)

            # 分类器
            ............
```

Reinforcement Learning model

再后来我又进行了深度强化学习的探索（仅仅只是探索，欢迎有兴趣的同学一同探讨交流），因为强化学习在文本分类领域还没有任何的尝试，2018年收录了两片相关的论文，我参考其中的一篇Learning Structured Representation for Text Classification via Reinforcement Learning。
模型原理也很简单，主要是结合了Policy Gradient和CNN文本分类。论文认为对于文章中的每一个分词，action空间有两种动作：retain和delete，每个action执行后把最终文本向量送入CNN文本分类网络获得reward，reward即为cross-entropy。

每处理完一个文本，便开始学习。



(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/50e879f7906ea2245942982c4730f958.png)

```
for epoch in range(3000):
        print('Epoch:', epoch + 1)
        batch_train = batch_iter(x_train, y_train, 256)
        for x_batch, y_batch in batch_train:
            for text_idx in range(len(x_batch)):
                text = x_batch[text_idx]
                retained = []
                for i in range(len(text)):
                    if text[i] == 0:
                        continue
                    # observation = RL.word_embedding([text[i]])
                    action = RL.choose_action([[text[i]]])
                    # print(action)
                    if action:  # 0:delete; 1:retain
                        retained.append(text[i])
                    # 返回reword
                    x_pad = pad_sequences([retained], 60)
                    y_pad = to_categorical([y_batch[text_idx]], RL.num_classes)
                    reward = RL.get_reward(x_pad, y_pad)
                    # 存储记忆
                    RL.store_transition(text[i], action, reward)
                # 每处理完一个text，便开始学习
                RL.learn()
```

训练数据规整及上线

一次偶然的机会，我对训练数据进行重新调整，去除了大部分噪声数据，同样的ngram CNN模型，准确率提高到94.6%，😕，永远记住：数据质量决定了模型能力的上限。
TensorFlow Serving：模型训练完成只能代表任务完成了一半，还要想办法将模型弄上线，这部分其实也有一定的难度和工作量。

传统机器学习算法对比

最后我还对一些传统机器学习算法用同样的数据进行了对比，结果如下表所示，注：算法没有优劣之分，只有适用场景的区别。

算法	准确率	说明
LR	0.7956	逻辑回归
NB	0.730129	朴素贝叶斯
KNN	0.776645	
SVM	0.822484	
DecisionTree	0.853710	
RandomForest	0.860677	
AdaBoost	0.357516	对异常样本敏感，异常样本在迭代中可能会获得较高的权重，影响最终的强学习器的预测准确性。
FastText	0.744	
GBDT	0.697(10 estimators)	estimators越大，效果越好
MaxEnt	0.56	最大熵：1. 用NLTK自带的MaxentClassifier，迭代巨慢，且效果差。2. 自己手动实现了一个，效果更差。

(http://ata2-img.cn-hangzhou.img-pub.aliyun-inc.com/9b2e3b8b59ed482f114b38785a46af6f.png)

写在最后

每一次不一样，都源自于一个勇敢的开始，并且只要全力以赴就无所谓失败。

评论文章 (2) 9 (/articles/98876/voteup) 0 4 收藏 (/articles/98876/mark/)

他们赞过该文章

海青 (/users/5333) 辰峰 (/users/60090) 凤律 (/users/107031) 子奚 (/users/114808) 泽煜 (/users/154527) 重修 (/users/285357) 彧翊 (/users/311459) 士武 (/users/337157) 宛言 (/users/346914)

相似文章

- Depp Learning 介绍2（转） (/articles/5763)
- 国际人工智能联合会议 IJCAI 201... (/articles/36683)
- ACL2015参会报告3 (/articles/39072)
- 为什么深度网络训练难度大？ (/articles/10996)
- 卷积神经网络(CNN)在文本分类中的应用 (/articles/36988)
- 论文札记之 - Generative Adversarial Nets (/articles/71005)

上一篇：基于Apache Beam的DTC实时数据平台Neptune (…

1F 彧翊 (/users/311459) 2018-01-22 08:54:19

非常不错的算法实践

0 (/comments/160726/voteup) 1

子颢 (/users/223413) 2018-01-22 09:26:41

谢谢，多交流~

0 (/comments/160726/subcomments/50835/voteup)

写下你的评论…

2F 宛言 (/users/346914) 2018-01-23 10:01:15

做了很多很多很多实验 手动点赞

0 (/comments/160858/voteup) 0

写下你的评论…

评论

ATA社区常见问题汇总 (/articles/67826)