文本分类算法及应用

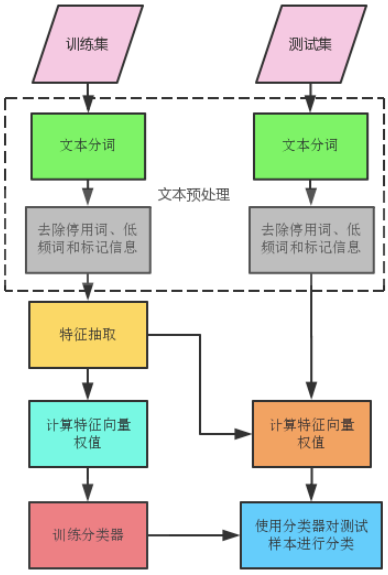
update by WenYa, 05/2017

1、文本分类简介

文本分类是指在给定分类体系下，根据文本内容自动确定文本类别的过程。比如：一些新闻网站含有大量的政治、军事、科技、体育等不同类别的文章，如何基于这些文章内容，自动的按题材进行分类。再比如，在京东淘宝等电子商务网站，用户会对交易商品进行评价。商家需要对用户的评价划分为正面和负面，以进行反馈统计。类似的，邮箱垃圾邮件的过滤，垃圾广告的处理等等，都是需要用到文本分类技术处理的场合。

当下主流的文本分类的方式，是基于机器学习、深度学习的方法。主要以统计理论为基础，进行有监督的学习，通常对已知的训练数据做统计分析，从而获得一些规律，再运用规律对未知的数据做预测。

文本分类的常见流程如下：



上图可以使用 sklearn+jieba 完成，具体过程如下：

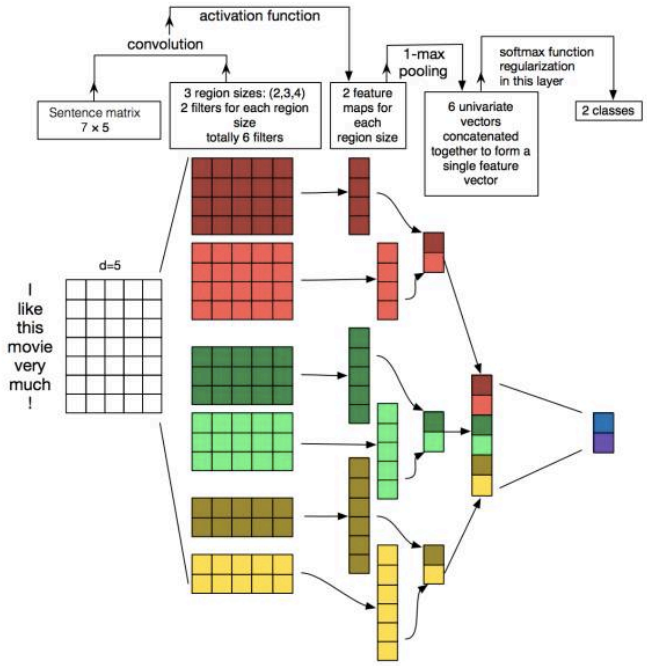
1、 数据准备：要得到一个准确率高、效果好的模型，大量的数据是必须的。只有数据的量达到一定程度，模型的正确率才能达到一个可以接受的水平。数据的量和模型复杂度，以及置信率的关系，可以参见 **vc 维理论**。

2、 数据清洗：海量的数据中，必然有不靠谱的错误的数据，我们可以先对数据进行适当的清洗。

3、 特征提取：使用 jieba 对文本进行分词，将文本表示成向量的形式，对于文本分类问题来说，数据的特征可以是：词频特征、TF-IDF 特征，由于词袋的规模非常的大，我们可以使用 Hashing Trick 技巧来进行降维操作。

4、 训练模型：将提取的特征喂给分类器，比如 bayes 分类器，训练 bayes 分类器，最后使用 bayes 分类器对未知的文本进行分类。

而当下文本分类任务，主流的方法是基于深度学习的方式，benchmark 则为卷积神经网络在文本分类中的应用。基于卷积神经网络的文本分类，最经典的论文就是 Kim Yoon 的 Convolutional Neural Networks for Sentence Classification，该论文的分类模型在诸如情感分类任务中获得了良好的性能，且目前这篇论文称为了文本分类的 baseline。其流程如下图所示：



详细过程：第一层是图中最左边的 7 乘 5 的句子矩阵，每行是词向量，维度=5，这个可以类比为图像中的原始像素点了。然后经过有 filter_size=(2,3,4) 的一维卷积层，每个 filter_size 有两个输出 channel。第三层是一个 1-max pooling 层，这样不同长度句子经过 pooling 层之后都能变成定长的表示了，最后接一层全连接的 softmax 层，输出每个类别的概率。

2、运行环境

python 2.7.13 + tensorflow 1.2 + genism(可选,在词向量使用 word2vec 时使用)

3、数据集获取

英文数据集：

http://www.cs.cornell.edu/people/pabo/movie-review-data/，使用和 Kim Yoon 论文中也使用的数据集：sentence polarity dataset v1.0 (includes 5331 positive and 5331 negative processed sentences / snippets.)

MR 电影评论数据，其中包含 10662 条评论，正面和负面评论各一半。共包含 18765 个单词（vocab_size），最长的评论有 59 个单词，数据集保存在 data 目录下的 rt-polarity.neg 和 rt-poliarity.pos 文件中。

中文数据集（待上传，sogou）

4、数据预处理

rt-poliarity.pos 正面评论的两个例子：

```
the charms of the lead performances allow us to forget most of the film's problems .
a vivid , sometimes surreal , glimpse into the mysteries of human behavior .
```

rt-polarity.neg 负面评论的两个例子：

```
simplistic , silly and tedious .
it's so laddish and juvenile , only teenage boys could possibly find it funny .
```

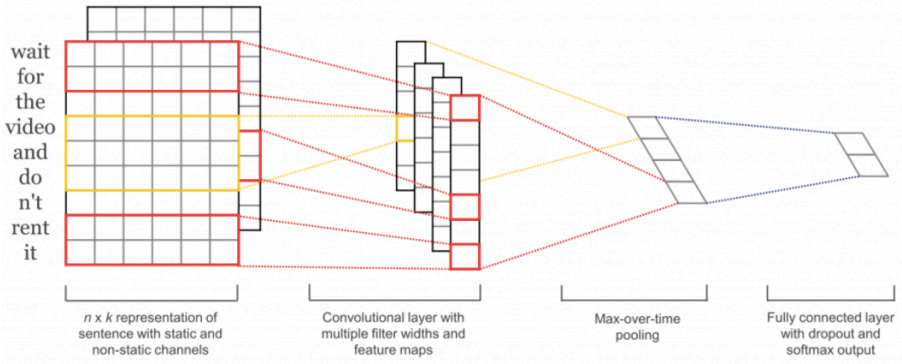
MR 电影评论数据，数据集相对来说还是比较的小，在数据处理阶段，首先我们需要适当的对数据集作清洗，以及对数据集进行训练集、验证测试集划分。这里取 10%的数据作为验证测试集，用于模型验证和参数调节。其次，将数据中非英文字母数字用 re.sub 函数空格替换，对于一些标点符号，加入空格，用于后续分词。然后，如果基于 one-hot 编码方式，我们需要构建词汇索引表，将每个单词映射到 0 ~ 18765 之间（18765 是词汇量大小），那么每个句子就变成了一个整数的向量。对于句子长度不一致的问题，我们可以通过 padding 的方式来解决，即填充标记至最大句子长度。如果基于 word2vec 方式，我们需要加载 word2vec 对数据进行处理，这部分后续我会补充提到。此外，由于文本分类，被作为一个有监督学习来对待，所以，训练数据，需要构建为数据+标签的形式，这里需要对数据集按照 pos(1), neg(0)的方式对句子评论标签化处理。最后，再利用数据训练 CNN 模型时，我们通常对数据批量化进行处理，也就

是设定 batch_size，批训练的引入最大好处是针对非凸损失函数来做的，毕竟非凸的情况下，全样本就算工程上算的动，也会卡在局部优上，而批量化处理相当于对全样本的部分抽样，修正梯度上的采样噪声，使其更有可能搜索最优值。

详细代码见：data_preprocess.py

5、 模型构建

CNN 用于文本分类最基本的 5 层架构如下图所示：包括 embedding layer、convolutional layer、max-pooling layer 、fully-connected layer 、softmax-layer。



第一层网络将词向量嵌入到一个低维的向量中。第二层网络就是利用多个卷积核在前一层网络上进行卷积操作。比如，每次滑动 3 个，4 个或者 5 个单词。第三层网络是一个 max-pool 层，从而得到一个长向量，并且添加上 dropout 正则项。第四层是全连接层，最后一层使用 softmax 函数对进行分类。由于就是对文本进行正负面二分类。

构建深度神经网络，最主要的两部分就是网络结构和网络参数的问题。这里网络结构，我们采用的是基本的 CNN 网络结构，针对这种网络结构，结合输入数据，我们需要设置一些基本参数，并对这些参数进行初始化。假设我们

现在已经构建好了网络，那么应该向这个网络里传入什么参数尼？如下表所示：

sequence_length	输入句子的长度（max_length = 59）
num_classes	输出的类别（正负面二分类）
vocabulary_size	词典大小（这里是 18765）
embedding_size	词向量的维度（这里是 128）
kernel_size	卷积核大小（纵向宽度，这里是 3，4，5）
num_filters	卷积核个数（这里是 128）
l2_reg	正则化强度（这里是 0）
stride	卷积核的步长，默认为 1
batch_size	批处理：一个 batch 中句子的数量
learning_rate	学习率：优化算法更新网络权重幅度大小
dropout_keep_prob	训练时候为 0.5，测试的时候置为 1

TensorFlow 中该如何定义：

1、首先实现 init 函数，传入参数：

```
def __init__(
  self, sequence_length, num_classes, vocab_size,
  embedding_size, filter_sizes, num_filters, l2_reg_lambda=0.0):
```

2、设置输入输出的占位符

```
self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name="input_x")
self.input_y = tf.placeholder(tf.float32, [None, num_classes], name="input_y")
```

这里第二个参数是输入张量的形状。`None` 的意思是，该维度的长度可以是任意值。在我们的模型中，第一个维度是批处理大小，而使用 `None` 来表示这个值，说明网络允许处理任意大小的批次。

3、embedding 层，根据输入 x 中每个单词在 voca 中的索引经过 lookup 得到其词向量。这里词向量使用随机初始化 W，而并未使用已经训练好的 word2vec。

```
with tf.device('/cpu:0'), tf.name_scope("embedding"):
    self.W = tf.Variable(tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0),name="W")
    self.embedded_chars = tf.nn.embedding_lookup(self.W, self.input_x)
    self.embedded_chars_expanded = tf.expand_dims(self.embedded_chars, -1)
```

这里设置代码在 cpu 上运行，且定义了一个 embedding 节点。W 是一个嵌入矩阵，刚开始使用随机均匀分布来初始化，后续通过从数据中不断训练校正。

Lookup 就是一个查找表,查找 input_x 对应在词汇表中的向量表示，返回 input_x 对应索引的三维张量[None, sequence_lengh, embedding_size]，因为卷积操作需要输入的是 4D 数据（大小，宽度，高度，通道数）而 embedded_chars 只有前三维，所以需要对维度进行扩展，添加 1 维，设为 1。变为：[None, sequence_length, embedding_size, 1]

4、convolutional 层，由于我们选择了三个卷积核大小[3,4,5]，所以，每个卷积核与嵌入层数据经过卷积运算后返回的张量是不同维度的。 这个过程可以见第 2 个图所示的那样。所以，我们需要遍历每一个卷积核，在 tensorflow 图中，为每个过滤器的卷积运算建立一个节点，然后将每个卷积核返回的结果合并成一个大的特征向量。卷积运算过程如下：

```
filter_shape = [filter_size, embedding_size, 1, num_filters]
W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name="W")
b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name="b")
conv = tf.nn.conv2d(
    self.embedded_chars_expanded,
    W,
    strides=[1, 1, 1, 1],
    padding="VALID",
    name="conv")
# Apply nonlinearity
h = tf.nn.relu(tf.nn.bias_add(conv, b), name="relu")
```

其中 tf.nn.conv2d 是 TensorFlow 里面实现卷积的函数，是搭建卷积神经网络比较核心的一个方法。

tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, name=None)
除去 name 参数用以指定该操作的 name，与方法有关的一共五个参数：

第一个参数 input：它要求是一个 4 维的 Tensor，具有[batch, in_height, in_width, in_channels]这样的 shape，具体含义是[训练时一个 batch 的数量，高度，宽度，通道数]，要求类型为 float32 和 float64 其中之一

第二个参数 filter：相当于 CNN 中的卷积核，它要求是一个 Tensor，具有[filter_height, filter_width, in_channels, out_channels]这样的 shape，具体含义是[卷积核的高度，卷积核的宽度，通道数，卷积核个数]，要求类型与参数 input 相同，有一个地方需要注意，第三维 in_channels，就是参数 input 的第四维

第三个参数 strides：卷积时在每一维的步长，这是一个一维的向量，长度为 4

第四个参数 padding：string 类型的量，只能是"SAME","VALID"其中之一，这个值决定了不同的卷积方式。`"VALID"` = without padding，`"SAME"` = with zero padding。

- `"VALID"` = without padding:

```
inputs:         1  2  3  4  5  6  7  8  9  10 11 (12 13)
               |_____________________|               dropped
                                     |_____________________|
```

- `"SAME"` = with zero padding:

```
               pad|                                      |pad
inputs:         0 |1  2  3  4  5  6  7  8  9  10 11 12 13|0  0
               |_____________________|
                                     |_____________________|
                                                           |_____________________|
```

第五个参数：use_cudnn_on_gpu:bool 类型，是否使用 cudnn 加速，默认为 true

结果返回一个 Tensor，这个输出，就是我们常说的 feature map。

h 表示采用激活函数 Relu 对经过卷积运算的输出进行非线性处理之后的结果，因为真实世界中大多数需要学习的问题都是非线性的，单纯的卷积操作-矩阵相乘、相加是线性的，所以才需要额外的计算引入非线性。非线性函数比如 tanh 或 sigmoid 也可以替代 relu，但多数情况下 relu 表现更好。

5、max-pooling 层：也可以称之为下采样，池化操作是利用一个矩阵窗口在输入张量上进行扫描，并且将每个矩阵窗口中的值通过取最大值，平均值或者其他方法来减少元素个数。每个池化操作的矩阵窗口大小是由 ksize 来指定的，并且根据步长参数 strides 来决定移动步长。通过该层，能够降低每个 feature map 的维度，保留最重要的信息。最大池化，也就是取每个区域的最大值。其操作过程如下：

```
pooled = tf.nn.max_pool(
    h,
    ksize=[1, sequence_length - filter_size + 1, 1, 1],
    strides=[1, 1, 1, 1],
    padding='VALID',
    name="pool")
```

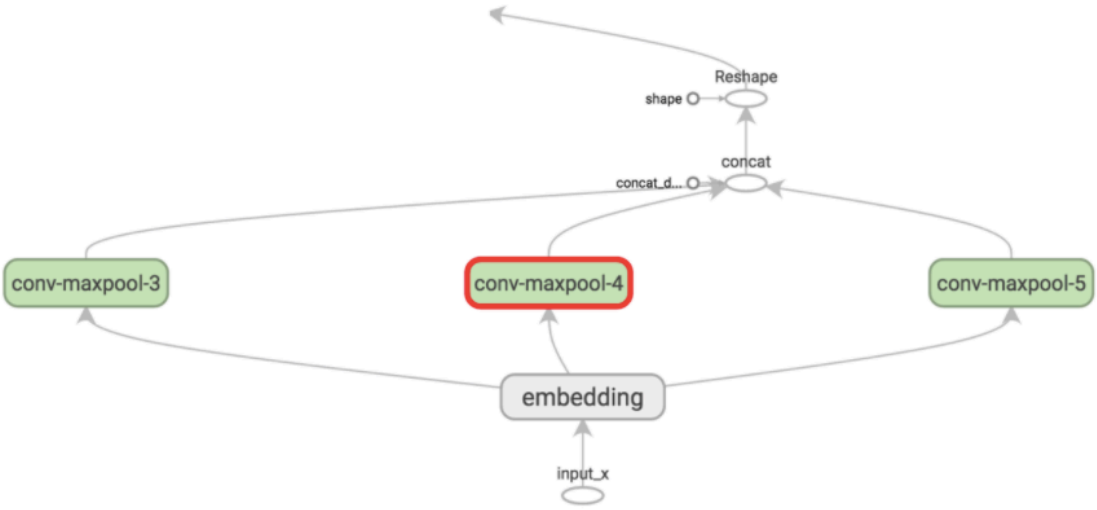
做最大池化操作，使得我们得到的张量维度是 [batch_size, 1, 1, num_filters]。这实质上就是一个特征向量，其中最后一个维度就是对应于我们的特征。一旦我们拥有了来自各个卷积核的输出向量，pooling 最终为一个长度为 3 的列表。每一个元素都是[None,1,1,128]的 Tensor 张量

那么我们就可以把它们合并成一个长的特征向量，如下面的操作：

```
# Combine all the pooled features
num_filters_total = num_filters * len(filter_sizes)
self.h_pool = tf.concat(pooled_outputs, 3)
self.h_pool_flat = tf.reshape(self.h_pool, [-1, num_filters_total])
```

即在第四个维度上进行合并，变成一个[None,1,1,384]Tensor 张量。该向量的维度是 [batch_size, num_filters_total] 。在 tf.reshape 中使用-1，就是告诉 TensorFlow 在可能的情况下，将 h_pool flatten 化，展开成两维 Tensor[None,384]。

上述的嵌入、卷积和池化整个过程可以用下图表示：



- 在输出层之前，可以加入 drop_out，防止过拟合。我们用一 placeholder 来代表一个神经元的输出在 dropout 中保持不变的概率。这样我们可以在训练过程中启用 dropout，在测试过程中关闭 dropout。

```
self.dropout_keep_prob = tf.placeholder(tf.float32, name="dropout_keep_prob")
```

```
self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)
```

6 、输出 softmax-layer 层，全连接层计算输出向量(w*h+b)和预测(scores 向量中的最大值即为预测结果)。操作如下：

```
with tf.name_scope("output"):
    W = tf.get_variable(
        "W",
        shape=[num_filters_total, num_classes],
        initializer=tf.contrib.layers.xavier_initializer())
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name="b")
    l2_loss += tf.nn.l2_loss(W)
    l2_loss += tf.nn.l2_loss(b)
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name="scores")
    self.predictions = tf.argmax(self.scores, 1, name="predictions")
```

7、设置损失函数，损失是网络犯错的度量，我们的目标是减小它。分类问题的标准损失函数是 cross-entropy loss。

```
losses = tf.nn.softmax_cross_entropy_with_logits(logits=self.scores, labels=self.input_y)
self.loss = tf.reduce_mean(losses) + l2_reg_lambda * l2_loss
```

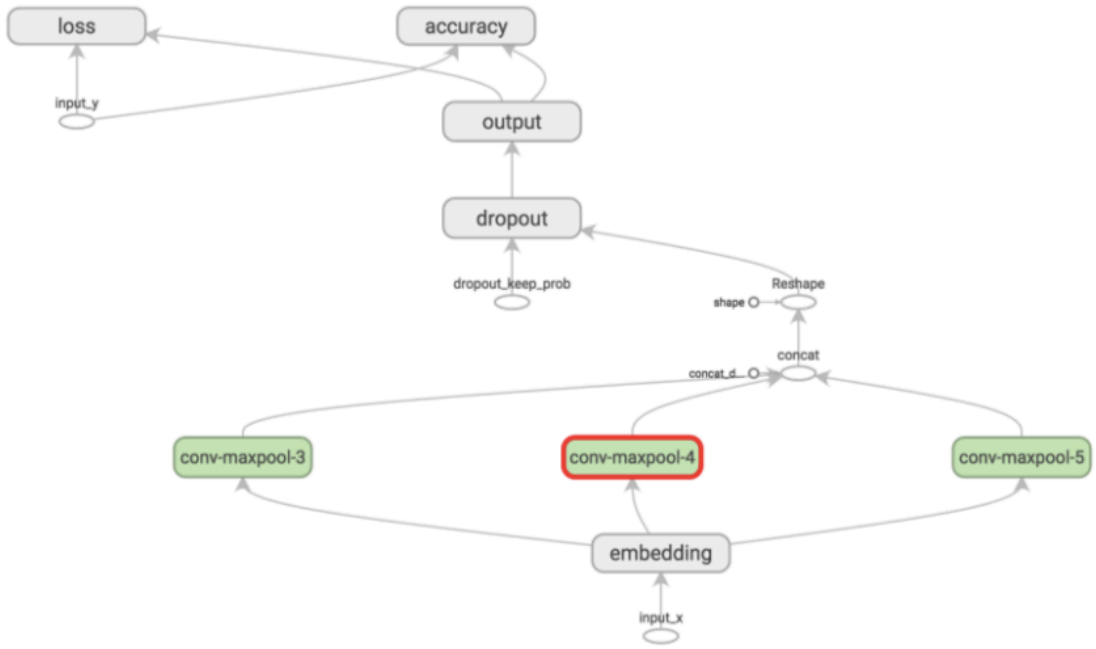
其中 tf.nn.softmax_cross_entropy_with_logits(logits, labels, name=None)

除去 name 参数用以指定该操作的 name，与方法有关的一共两个参数：第一个参数 logits：就是神经网络最后一层的输出，如果有 batch 的话，它的大小就是[batchsize，num_classes]，单样本的话，大小就是 num_classes。第二个参数 labels：实际的标签，大小同上。注意这个函数的返回值并不是一个数，而是一个向量，求 loss，则要做一步 tf.reduce_mean 操作，对向量求均值。

此外，我们可以计算准确度，预测和真实标签相同即为正确。

```
correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"), name="accuracy")
```

整个流程 TensorBoard 可视化网络如下图所示：



该部分详细代码见：build_model.py

6、 训练过程

既然神经网络模型已经搭建完毕，那么接下来就是传入数据，开始一步步的训练了。这部分除了需要初始化数据处理阶段和模型构建阶段需要定义的参数之外，还需要相应的配置模型的训练参数，主要包括：迭代次数、保存模型的

频率、评估模型的频率、优化损失过程的学习率等。训练过程也主要就是围绕上述三方面的参数来展开。

Train 的过程其实就是向模型里喂入数据，启动计算，不断优化损失函数，更新网络参数，直到几乎收敛，得到最佳参数的过程。不过除此之外，在这里我们还要使用批处理进行操作，也就是对数据进行批次迭代处理。

由于整个训练过程，包括模型的构建，都是基于 tensorflow 深度学习框架展开的，所以，必须搞清楚 tensorflow 最基本的概念和用法，这里再回顾下。Tensorflow 是一个图计算编程系统，使用图来表示计算任务。对于图我们自然会想到节点和边，而这里的节点被称之为 op(operation 的缩写)，一个 op 获得 0 个或多个 tensor（这里表示数据，是一个类型化的多维数组，），并返回 0 个或多个 tensor。而边在这里就是一种连接，数据在这些边上流动，比如两个 op 建立了联系，那么边上，就会有数据的传递，也就是 tensor 的 flow。整个过程的状态，需要变量来维护。而整个图需要在被称之为会话（Session）的上下文中执行。每个会话执行一个单一的图。如果你在创建变量和操作时，没有明确地使用一个会话，那么 TensorFlow 会创建一个当前默认会话。你可以在程序中使用多个图，但是大多数程序都只需要一个图。所以，整体来说 tensorflow 程序通常被组织成两个阶段（构建阶段和执行阶段），通常在构建阶段创建一个图来表示和训练神经网络（op 的执行步骤被描述为一个图），然后在执行阶段反复执行图中的训练 op。

整个训练过程具体流程如下：

1、定义图，创建会话

```
with tf.Graph().as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
```

其中 allow_soft_placement 参数的设置，允许 TensorFlow 回退到特定操作的设备，如果在优先设备不存在时。比如，如果我们的代码是运行在一个 GPU 上面的，但是我们的代码在一个没有 GPU 的机器上运行了。那么，如果不使用 allow_soft_placement 参数，程序就会报错。如果设置了 log_device_placement 参数，TensorFlow 会记录它运行操作的设备（CPU 或者 GPU）。

2、实例化模型，实例化一个构建好的模型，这样使得所有定义的变量和操作都将被放入我们创建的默认图和会话中。

```
cnn = build_model(
    sequence_length=x_train.shape[1],
    num_classes=y_train.shape[1],
    vocab_size=len(vocab_processor.vocabulary_),
    embedding_size=FLAGS.embedding_dim,
    filter_sizes=list(map(int, FLAGS.filter_sizes.split(","))),
    num_filters=FLAGS.num_filters,
    l2_reg_lambda=FLAGS.l2_reg_lambda)
```

3、优化损失函数，tensorflow 中有很多内嵌的优化函数，比如常用的优化函数：**GradientDescentOptimizer** 、**AdamOptimizer** 、**RMSPropOptimizer**，一般对于文本类型数据的处理，我们常用 Adam 优化器。

```
global_step = tf.Variable(0, name="global_step", trainable=False)
optimizer = tf.train.AdamOptimizer(1e-3)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

通过定义 global_step 变量并将它传递给优化器，对模型的参数进行梯度更新。每次执行 train_op 操作，就是一个训练步骤，并不返回任何东西，只是更

新网络参数。TensorFlow 会自动计算出哪些变量是“可训练”的，并计算它们的梯度。

4、单步训练，定义一个训练函数，用于单个训练步骤，在一批数据上进行评估，并且更新模型参数。后续在批量化循环训练的时候，就可以直接调用这个函数。

```
def train_step(x_batch, y_batch):
    """
    A single training step
    """
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: FLAGS.dropout_keep_prob
    }
    _, step, summaries, loss, accuracy = sess.run(
        [train_op, global_step, train_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
```

```
# Training loop. For each batch...
for batch in batches:
    x_batch, y_batch = zip(*batch)
    train_step(x_batch, y_batch)
    current_step = tf.train.global_step(sess, global_step)
```

其中，feed_dict 就是向之前定义的占位符中喂入数据，然后通过 sess.run 执行 train_op 方法，在循环训练过程中，每次批处理的过程都调用了一次 train_step 函数。然后通过 saver 保存训练的模型。当然训练之前所有变量要做好初始化。

```
saver.save(sess, checkpoint_prefix, global_step=current_step)
```

5、Saver 用法，我们经常在训练完一个模型之后希望保存训练的结果，这些结果指的是模型的参数，以便下次迭代的训练或者用作测试。Tensorflow 针对这一需求提供了 Saver 类。Saver 类提供了向 checkpoints 文件保存和从 checkpoints 文件中恢复变量的相关方法。Checkpoints 文件是一个二进制文件，它把变量名映射到对应的 tensor 值。只要提供一个计数器，当计数器触发时，Saver 类可以自动的生成 checkpoint 文件。这让我们可以在训练过程中保存多个中间结果。例如，我们可以保存每一步训练的结果。为了避免填满整个磁盘，

Saver 可以自动的管理 Checkpoints 文件。例如，我们可以指定保存最近的 N 个 Checkpoints 文件。**Saver** 将保存与图有关联的任何元数据。这就意味着，当我们恢复一个模型的时候，我们还同时恢复了所有与图相关的变量、操作和集合。

```
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.global_variables(), max_to_keep=FLAGS.num_checkpoints)
```

```
# Load the saved meta graph and restore variables
saver = tf.train.import_meta_graph("{}.meta".format(checkpoint_file))
saver.restore(sess, checkpoint_file)
```

num_checkpoints：表示训练多少次保存一下 checkpoints

checkpoint_dir：表示 checkpoints 文件的保存路径

6、TensorFlow 可视化：tensorflow 的可视化是使用 summary 和 tensorboard 合作完成的，它的作用就是可以把复杂的神经网络训练过程给可视化，允许你在训练和评估阶段来跟踪和可视化各种参数。可以更好地理解调试优化程序。

首先明确一点,summary 也是 op。summary 常见操作： tf.summary.scalar（当你想知道 learning rate、目标函数如何变化时，就可以通过向节点添加这个函数操作来分别输出学习速度和期望误差，给每个 scalary_summmay 分配一个有意义的标签为’learning_rate’和’loss function’，执行后可以看到可视化的图表）、

```
# Summaries for loss and accuracy
loss_summary = tf.summary.scalar("loss", cnn.loss)
acc_summary = tf.summary.scalar("accuracy", cnn.accuracy)
```

tf.summary.histogram（生成直方图 summary，当你想看 activations, gradients, weights 的分布时，可以用这个函数，它显示了每一步的分布，越靠前面面就是越新的步数的结果。），tf.summary.merge_all（将之前定义的 summary op 整合到一起，就像变量需要初始化一样，summary 也需要

merge），tf.summary.FileWriter(创建一个 file writer 用来向硬盘写 summary 数据)。

```
dev_summary_op = tf.summary.merge([loss_summary, acc_summary])
dev_summary_dir = os.path.join(out_dir, "summaries", "dev")
dev_summary_writer = tf.summary.FileWriter(dev_summary_dir, sess.graph)
```

启动 TensorBoard，如果已经通过 pip 安装了 TensorBoard，就可以通过执行更为简单地命令来访问 TensorBoard：

```
tensorboard –logdir = /path/to/log-directory
```

这里的参数 logdir 指向 SummaryWriter 序列化数据的存储路径。如果 logdir 目录的子目录中包含另一次运行时的数据，那么 TensorBoard 会展示所有运行的数据。一旦 TensorBoard 开始运行，你可以通过在浏览器中输入 localhost:6006 来查看 TensorBoard。

上述的这些过程可以看作是训练阶段的构建过程，为了能够循环不断的向前训练，采用了合适的优化函数，优化目标。为了可视化训练过层的状况，使用了 tensorboard，summary。为了能够保存训练的模型以及便于后续恢复模型，使用了 tensorflow 的 saver 类。而这些过程都需要在一个 Session 会话里完成。

以上就是训练的整个过程，具体代码见 train_model.py

7、 测试过程

在已经训练好模型或有 pre-trained 模型的情况下，对现有测试集进行测试的过程如下：

```
checkpoint_file = tf.train.latest_checkpoint(FLAGS.checkpoint_dir)
graph = tf.Graph()
with graph.as_default():
    session_conf = tf.ConfigProto(
      allow_soft_placement=FLAGS.allow_soft_placement,
      log_device_placement=FLAGS.log_device_placement)
    sess = tf.Session(config=session_conf)
    with sess.as_default():
        # Load the saved meta graph and restore variables
        saver = tf.train.import_meta_graph("{}.meta".format(checkpoint_file))
        saver.restore(sess, checkpoint_file)
```

```
for x_test_batch in batches:
    batch_predictions = sess.run(predictions, {input_x: x_test_batch, dropout_keep_prob: 1.0})
    all_predictions = np.concatenate([all_predictions, batch_predictions])
```

首先，启动 session，restore 训练好的模型，预测即可。

8、其他相关实现

https://github.com/yoonkim/CNN_sentence