WILDML
人工智能，深度学习和NLP

2015年12月11日作者：DENNY BRITZ

在TensorFlow中实现CNN进行文本分类

完整的代码可以在Github上找到。

在这篇文章中，我们将实现一个类似于Kim Yoon的用于句子分类的卷积神经网络的模型。本文提出的模型在一系列文本分类任务（如情感分析）中实现了良好的分类性能，并已成为新文本分类体系结构的标准基线。

我假设您已经熟悉应用于NLP的卷积神经网络的基础知识。如果没有，我建议首先阅读了解NLP的卷积神经网络，以获得必要的背景知识。

数据和预处理

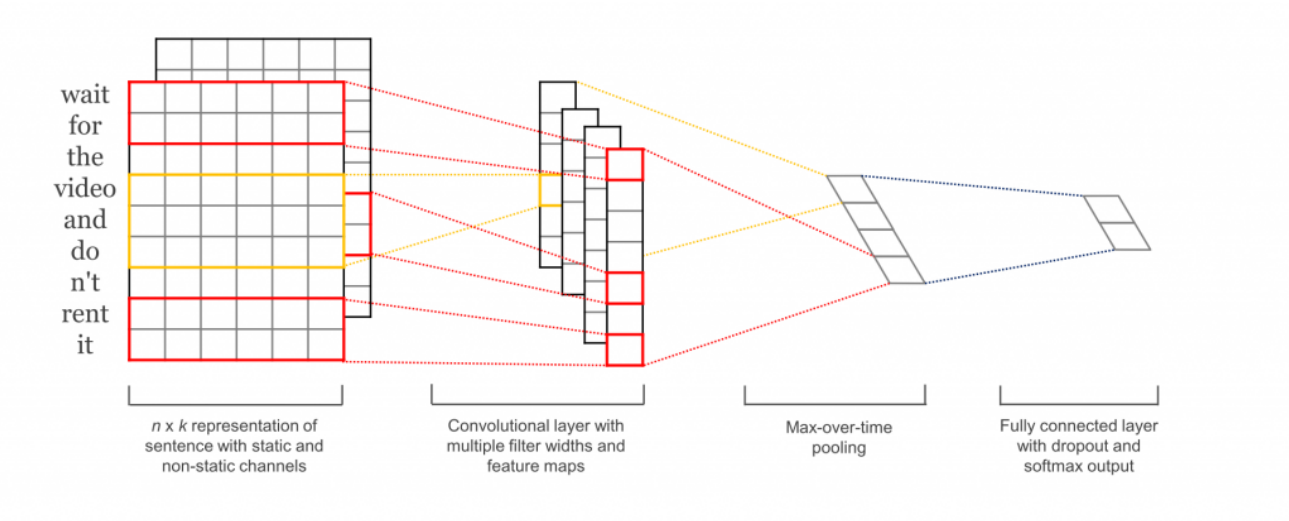
我们将在这篇文章中使用的数据集是来自烂番茄的电影评论数据 - 原始论文中也使用的数据集之一。该数据集包含10,662个例子评论句子，半正面和半负面。数据集的大小约为20k。请注意，由于此数据集非常小，我们可能会过度使用功能强大的模型。此外，数据集没有官方列车/测试拆分，因此我们只使用10％的数据作为开发集。原始论文报告了对数据进行10倍交叉验证的结果。

我不会在这篇文章中讨论数据预处理代码，但它可以在Github上获得并执行以下操作：

1. 从原始数据文件加载正面和负面的句子。
2. 使用与原始纸张相同的代码清洁文本数据。
3. 将每个句子填充到最大句子长度，结果为59.我们将特殊`<PAD>`标记附加到所有其他句子，使它们成为59个单词。将句子填充到相同长度是有用的，因为它允许我们有效地批量处理我们的数据，因为批处理中的每个示例必须具有相同的长度。
4. 构建词汇索引并将每个单词映射到0到18,765之间的整数（词汇量大小）。每个句子都成为整数的向量。

该模型

我们将在这篇文章中构建的网络大致如下：



第一层将单词嵌入到低维向量中。下一层使用多个滤波器大小对嵌入的字矢量执行卷积。例如，一次滑动3个，4个或5个字。接下来，我们将卷积层的结果最大化为长特征向量，添加丢失正则化，并使用softmax层对结果进行分类。

因为这是一篇教育性的帖子，所以我决定从原始论文中简化模型：

- 我们不会使用预先训练过的word2vec向量进行单词嵌入。相反，我们从头开始学习嵌入。
- 我们不会对权重向量强制执行L2范数约束。对句子分类的卷积神经网络（和从业者指南）的敏感性分析发现，约束对最终结果影响不大。
- 原始论文用两个输入数据通道进行实验 - 静态和非静态单词向量。我们只使用一个频道。

在这里向代码添加上述扩展是相对简单的（几十行代码）。看一下帖子末尾的练习。

让我们开始吧！

履行

为了允许各种超参数配置，我们将代码放入一个`TextCNN`类中，在`init`函数中生成模型图。

```
1 import tensorflow as tf
2 import numpy as np
3
```

```
class TextCNN(object):
    &quot;&quot;&quot;
    A CNN for text classification.
    Uses an embedding layer, followed by a convolutional, max-pooling and softmax la
yer.
    &quot;&quot;&quot;
    def __init__(
      self, sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters):
        # Implementation...
```

为了实例化该类，我们传递以下参数：

- `sequence_length` - 我们句子的长度。请记住，我们将所有句子填充为相同的长度（我们的数据集为59）。
- `num_classes` - 输出层中的类数，在我们的例子中为两个（正数和负数）。
- `vocab_size` - 我们词汇量的大小。这是定义嵌入层的大小所必需的，嵌入层将具有形状`[vocabulary_size, embedding_size]`。
- `embedding_size` - 嵌入的维度。
- `filter_sizes` - 我们希望卷积滤镜覆盖的单词数。我们将为`num_filters`此处指定的每个尺寸。例如，`[3, 4, 5]`意味着我们将分别为3个，4个和5个单词滑动过滤器，以获得总共`3 * num_filters`过滤器。
- `num_filters` - 每个过滤器大小的过滤器数量（见上文）。

输入占位符

我们首先定义传递给网络的输入数据：

```
# Placeholders for input, output and dropout
self.input_x = tf.placeholder(tf.int32, [None, sequence_length], name=&quot;input_x&q
uot;)
self.input_y = tf.placeholder(tf.float32, [None, num_classes], name=&quot;input_y&quo
t;)
self.dropout_keep_prob = tf.placeholder(tf.float32, name=&quot;dropout_keep_prob&quot
;)
```

`tf.placeholder`创建一个占位符变量，当我们在火车或测试时执行它时，我们将其提供给网络。第二个参数是输入张量的形状。`None`意味着该维度的长度可以是任何东西。在我们的例子中，第一个维度是批量大小，并且使用`None`允许网络处理任意大小的批次。

在丢失层中保持神经元的概率也是网络的输入，因为我们仅在训练期间启用丢失。我们在评估模型时禁用它（稍后会详细介绍）。

嵌入图层

我们定义的第一层是嵌入层，它将词汇单词索引映射到低维向量表示。它本质上是一个我们从数据中学习的查找表。

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

我们在这里使用了一些新功能，让我们来看看它们：

- `tf.device("/cpu:0")`强制在CPU上执行操作。默认情况下，TensorFlow会尝试将操作放在GPU上（如果有），但嵌入实现当前没有GPU支持，如果置于GPU上则会引发错误。
- `tf.name_scope`创建一个名为“embedding” 的新名称范围。范围将所有操作添加到称为“嵌入”的顶级节点中，以便在TensorBoard中可视化网络时获得良好的层次结构。

`W`是我们在培训期间学习的嵌入矩阵。我们使用随机均匀分布对其进行初始化。`tf.nn.embedding_lookup`创建实际的嵌入操作。嵌入操作的结果是三维形状张量`[None, sequence_length, embedding_size]`。

TensorFlow的卷积转换操作需要一个4维张量，其尺寸对应于批次，宽度，高度和通道。我们嵌入的结果不包含通道尺寸，因此我们手动添加它，为我们留下一层形状`[None, sequence_length, embedding_size, 1]`。

卷积和最大池层

现在我们已经准备好构建我们的卷积层，然后是最大池。请记住，我们使用不同大小的过滤器。因为每个卷积产生不同形状的张量，我们需要迭代它们，为它们中的每一个创建一个层，然后将结果合并为一个大的特征向量。

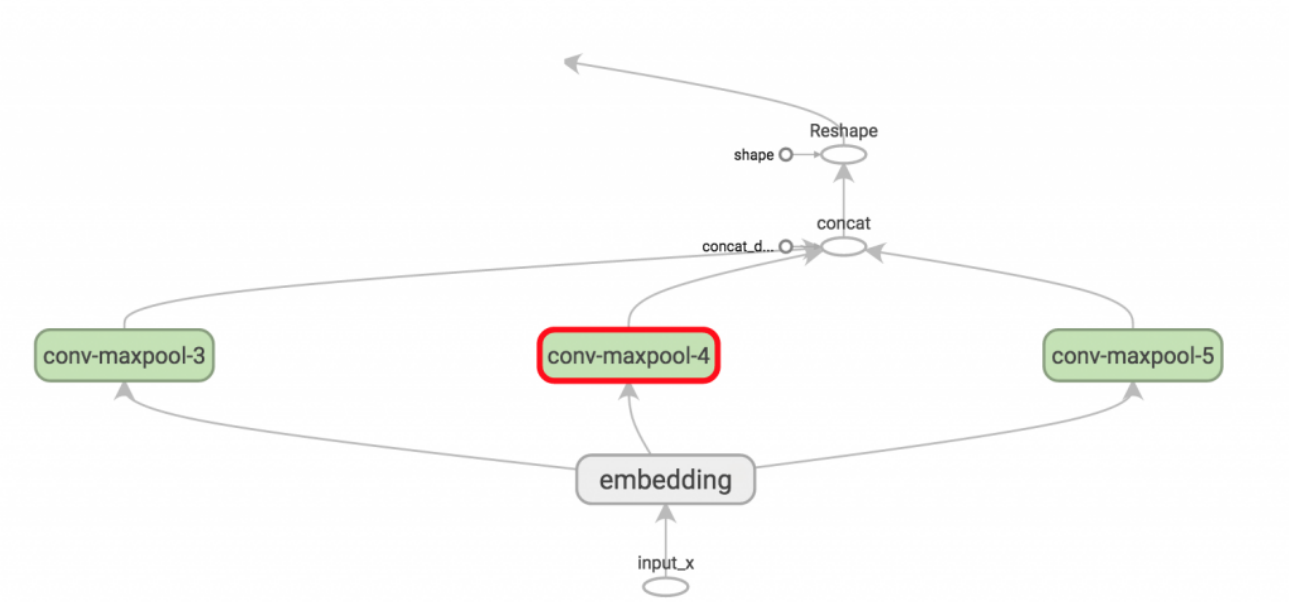
```
pooled_outputs = []
for i, filter_size in enumerate(filter_sizes):
    with tf.name_scope(&quot;conv-maxpool-%s&quot; % filter_size):
        # Convolution Layer
        filter_shape = [filter_size, embedding_size, 1, num_filters]
        W = tf.Variable(tf.truncated_normal(filter_shape, stddev=0.1), name=&quot;W&
quot;)
        b = tf.Variable(tf.constant(0.1, shape=[num_filters]), name=&quot;b&quot;)
        conv = tf.nn.conv2d(
            self.embedded_chars_expanded,
            W,
            strides=[1, 1, 1, 1],
            padding=&quot;VALID&quot;,
            name=&quot;conv&quot;)
        # Apply nonlinearity
        h = tf.nn.relu(tf.nn.bias_add(conv, b), name=&quot;relu&quot;)
        # Max-pooling over the outputs
        pooled = tf.nn.max_pool(
```

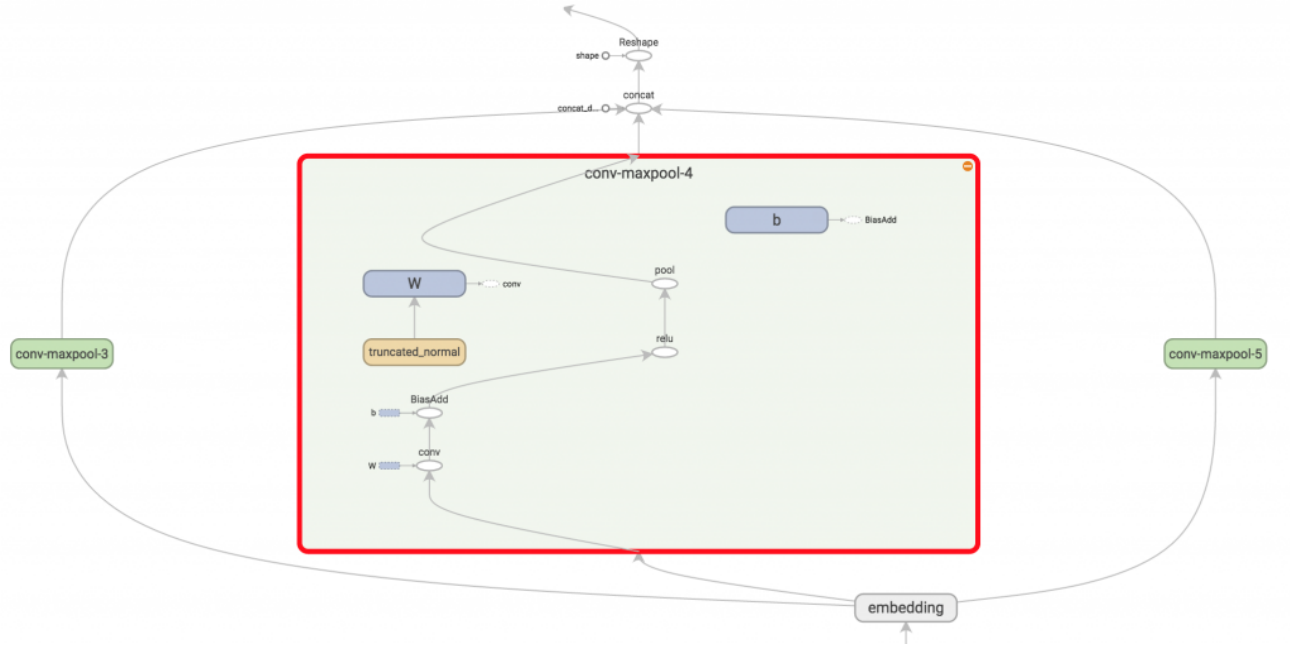
```
            h,
            ksize=[1, sequence_length - filter_size + 1, 1, 1],
            strides=[1, 1, 1, 1],
            padding='VALID',
            name=&quot;pool&quot;)
        pooled_outputs.append(pooled)

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

这里W是我们的滤波器矩阵，h是将非线性应用于卷积输出的结果。每个过滤器都滑过整个嵌入，但它覆盖的单词数量会有所不同。"VALID"padding意味着我们将滤镜滑过句子而不填充边缘，执行窄卷积，为我们提供形状输出`[1, sequence_length - filter_size + 1, 1, 1]`。在特定过滤器尺寸的输出上执行最大池化使我们具有张量形状`[batch_size, 1, 1, num_filters]`。这本质上是一个特征向量，其中最后一个维度对应于我们的特征。一旦我们从每个滤波器大小获得所有合并的输出张量，我们将它们组合成一个长形状的特征向量`[batch_size, num_filters_total]`。使用-1in `tf.reshape`告诉TensorFlow尽可能展平尺寸。

花些时间尝试了解每个操作的输出形状。您还可以参考了解NLP的卷积神经网络以获得一些直觉。在TensorBoard中可视化操作也可能有所帮助（对于特定的过滤器尺寸3,4和5）：





辍学层

辍学可能是最流行的卷积神经网络规范化方法。辍学背后的想法很简单。辍学层随机“禁用”其神经元的一部分。这可以防止神经元共同适应并迫使它们学习单独有用的特征。我们保持启用的神经元部分由dropout_keep_prob我们网络的输入定义。我们在训练期间将其设置为0.5，在评估期间设置为1（禁用丢失）。

```
# Add dropout
with tf.name_scope(&quot;dropout&quot;):
    self.h_drop = tf.nn.dropout(self.h_pool_flat, self.dropout_keep_prob)
```

分数和预测

使用max-pooling中的特征向量（应用了dropout），我们可以通过矩阵乘法和选择具有最高分数的类来生成预测。我们还可以应用softmax函数将原始分数转换为标准化概率，但这不会改变我们的最终预测。

```
with tf.name_scope(&quot;output&quot;):
    W = tf.Variable(tf.truncated_normal([num_filters_total, num_classes], stddev=0.1)
, name=&quot;W&quot;)
    b = tf.Variable(tf.constant(0.1, shape=[num_classes]), name=&quot;b&quot;)
    self.scores = tf.nn.xw_plus_b(self.h_drop, W, b, name=&quot;scores&quot;)
    self.predictions = tf.argmax(self.scores, 1, name=&quot;predictions&quot;)
```

这里tf.nn.xw_plus_b是执行$Wx + b$矩阵乘法的便利包装器。

损失和准确性

使用我们的分数，我们可以定义损失函数。损失是我们网络错误的衡量标准，我们的目标是最小化它。分类的标准损失函数问题是交叉熵损失。

```
# Calculate mean cross-entropy loss
with tf.name_scope(&quot;loss&quot;):
    losses = tf.nn.softmax_cross_entropy_with_logits(self.scores, self.input_y)
    self.loss = tf.reduce_mean(losses)
```

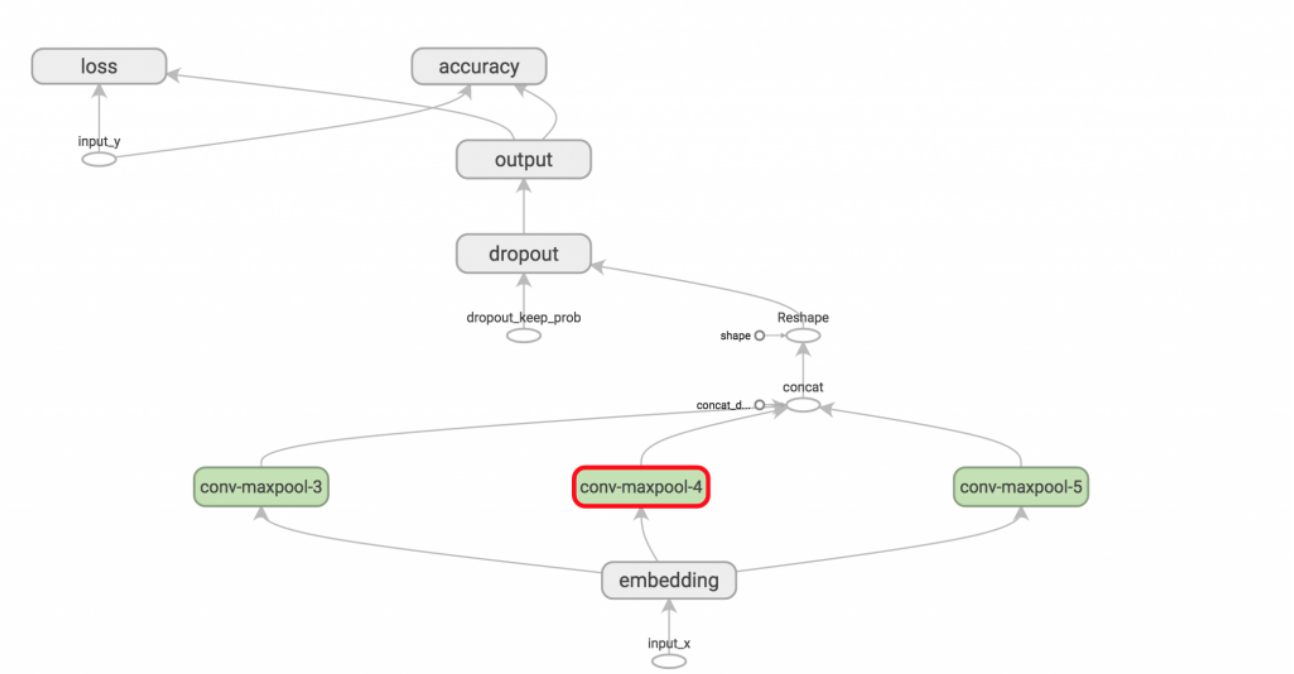
这里，tf.nn.softmax_cross_entropy_with_logits是一个便利函数，它根据我们的分数和正确的输入标签计算每个类的交叉熵损失。然后我们采取损失的平均值。我们也可以使用总和，但这使得比较不同批量大小和训练/开发数据的损失变得更加困难。

我们还定义了精度的表达式，这是在训练和测试期间跟踪的有用数量。

```
# Calculate Accuracy
with tf.name_scope(&quot;accuracy&quot;):
    correct_predictions = tf.equal(self.predictions, tf.argmax(self.input_y, 1))
    self.accuracy = tf.reduce_mean(tf.cast(correct_predictions, &quot;float&quot;), name=&quot;accuracy&quot;)
```

可视化网络

就是这样，我们完成了网络定义。此处提供完整的代码网络定义代码。为了了解全局，我们还可以在TensorBoard中可视化网络：



培训程序

在我们为网络定义培训过程之前，我们需要了解TensorFlow如何使用Sessions和的一些基础知识Graphs。如果您已经熟悉这些概念，请随意跳过本节。

在TensorFlow中，a Session是您正在执行图操作的环境，它包含有关变量和队列的状态。每个会话都在一个图表上运行。如果在创建变量和操作时未明确使用会话，则使用TensorFlow创建的当前默认会话。您可以通过执行session.as_default()块内的命令来更改默认会话（请参阅下文）。

A Graph包含操作和张量。您可以在程序中使用多个图形，但大多数程序只需要一个图形。您可以在多个会话中使用相同的图形，但不能在一个会话中使用多个图形。TensorFlow始终创建默认图形，但您也可以手动创建图形并将其设置为新默认图形，如下所示。显式创建会话和图表可确保在您不再需要资源时正确释放资源。

```
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():
        # Code that operates on the default graph and session comes here...
```

所述allow_soft_placement设置允许TensorFlow回落的设备上时，优选的设备不存在实现的某些操作。例如，如果我们的代码在GPU上放置操作并且我们在没有GPU的机器上运行代码，则不使用allow_soft_placement会导致错误。如果设置了log_device_placement，TensorFlow会记录它放置操作的设备（CPU或GPU）。这对调试很有用。FLAGS是我们程序的命令行参数。

实例化CNN并最大限度地减少损失

当我们实例化我们的TextCNN模型时，所有定义的变量和操作将被放入我们上面创建的默认图和会话中。

```
cnn = TextCNN(
    sequence_length=x_train.shape[1],
    num_classes=2,
    vocab_size=len(vocabulary),
    embedding_size=FLAGS.embedding_dim,
    filter_sizes=map(int, FLAGS.filter_sizes.split(&amp;quot;,&amp;quot;)),
    num_filters=FLAGS.num_filters)
```

接下来，我们定义如何优化网络的损失功能。TensorFlow有几个内置的优化器。我们正在使用Adam优化器。

```
global_step = tf.Variable(0, name=&amp;quot;global_step&amp;quot;, trainable=False)
optimizer = tf.train.AdamOptimizer(1e-4)
grads_and_vars = optimizer.compute_gradients(cnn.loss)
train_op = optimizer.apply_gradients(grads_and_vars, global_step=global_step)
```

这里，train_op这是一个新创建的操作，我们可以运行它来对我们的参数执行渐变更新。每次执行train_op都是一个训练步骤。TensorFlow自动确定哪些变量是“可训练的”并计算其梯度。通过定义global_step变量并将其传递给优化器，我们允许TensorFlow为我们处理训练步骤的计数。每次执行时，全局步骤将自动递增1 train_op。

摘要

TensorFlow具有摘要概念，使您可以在培训和评估过程中跟踪和可视化各种数量。例如，您可能希望跟踪损失和准确度随时间的变化情况。您还可以跟踪更复杂的数量，例如图层激活的直方图。摘要是序列化对象，它们使用SummaryWriter写入磁盘。

```
# Output directory for models and summaries
timestamp = str(int(time.time()))
out_dir = os.path.abspath(os.path.join(os.path.curdir, &amp;quot;runs&amp;quot;, timestamp))
print(&amp;quot;Writing to {}\n&amp;quot;.format(out_dir))

# Summaries for loss and accuracy
loss_summary = tf.scalar_summary(&amp;quot;loss&amp;quot;, cnn.loss)
acc_summary = tf.scalar_summary(&amp;quot;accuracy&amp;quot;, cnn.accuracy)

# Train Summaries
train_summary_op = tf.merge_summary([loss_summary, acc_summary])
train_summary_dir = os.path.join(out_dir, &amp;quot;summaries&amp;quot;, &amp;quot;train&amp;quot;)
train_summary_writer = tf.train.SummaryWriter(train_summary_dir, sess.graph_def)

# Dev summaries
dev_summary_op = tf.merge_summary([loss_summary, acc_summary])
dev_summary_dir = os.path.join(out_dir, &amp;quot;summaries&amp;quot;, &amp;quot;dev&amp;quot;)
dev_summary_writer = tf.train.SummaryWriter(dev_summary_dir, sess.graph_def)
```

在这里，我们分别跟踪培训和评估的摘要。在我们的例子中，这些数量相同，但您可能只有在训练期间要跟踪的数量（如参数更新值）。tf.merge_summary是一个便捷函数，它将多个汇总操作合并为一个我们可以执行的操作。

检查点

您通常要使用的另一个TensorFlow功能是检查点 - 保存模型的参数以便以后恢复它们。检查点可用于稍后继续训练，或使用提前停止选择最佳参数设置。使用Saver对象创建检查点。

```
# Checkpointing
checkpoint_dir = os.path.abspath(os.path.join(out_dir, &amp;quot;checkpoints&amp;quot;))
checkpoint_prefix = os.path.join(checkpoint_dir, &amp;quot;model&amp;quot;)
# Tensorflow assumes this directory already exists so we need to create it
if not os.path.exists(checkpoint_dir):
    os.makedirs(checkpoint_dir)
saver = tf.train.Saver(tf.all_variables())
```

初始化变量

在我们训练模型之前，我们还需要在图中初始化变量。

```
sess.run(tf.initialize_all_variables())
```

该initialize_all_variables功能是一个方便的功能，运行所有我们为我们的变量定义的初始化的。您也可以手动调用变量的初始值设定项。如果您想要使用预先训练的值初始化嵌入，这非常有用。

定义单个培训步骤

现在让我们为单个训练步骤定义一个函数，在一批数据上评估模型并更新模型参数。

```
def train_step(x_batch, y_batch):
    &quot;&quot;&quot;
    A single training step
    &quot;&quot;&quot;
    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)
    time_str = datetime.datetime.now().isoformat()
    print(&quot;{}: step {}, loss {:g}, acc {:g}&quot;.format(time_str, step, loss,
accuracy))
    train_summary_writer.add_summary(summaries, step)
```

feed_dict包含我们传递给网络的占位符节点的数据。您必须为所有占位符节点提供值，否则TensorFlow将引发错误。处理输入数据的另一种方法是使用队列，但这超出了本文的范围。

接下来，我们执行train_opusing session.run，它返回我们要求它评估的所有操作的值。请注意，train_op什么都不返回，它只是更新我们网络的参数。最后，我们打印当前培训批次的丢失和准确性，并将摘要保存到磁盘。请注意，如果批次较小，批次培训批次的损失和准确性可能会有很大差异。而且由于我们使用的是辍学，您的培训指标可能会比评估指标更差。

我们编写了一个类似的函数来评估任意数据集的损失和准确性，例如验证集或整个训练集。基本上这个功能与上面的功能相同，但没有训练操作。它还会禁用丢失。

```
def dev_step(x_batch, y_batch, writer=None):
    &quot;&quot;&quot;
    Evaluates model on a dev set
    &quot;&quot;&quot;
    feed_dict = {
      cnn.input_x: x_batch,
      cnn.input_y: y_batch,
      cnn.dropout_keep_prob: 1.0
```

```
    }
    step, summaries, loss, accuracy = sess.run(
        [global_step, dev_summary_op, cnn.loss, cnn.accuracy],
        feed_dict)
    time_str = datetime.datetime.now().isoformat()
    print(&quot;{}: step {}, loss {:g}, acc {:g}&quot;.format(time_str, step, loss,
accuracy))
    if writer:
        writer.add_summary(summaries, step)
```

训练循环

最后，我们准备编写训练循环了。我们迭代批量数据，train_step为每个批次调用函数，偶尔评估和检查我们的模型：

```
# Generate batches
batches = data_helpers.batch_iter(
    zip(x_train, y_train), FLAGS.batch_size, FLAGS.num_epochs)
# 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)
    if current_step % FLAGS.evaluate_every == 0:
        print(&quot;\nEvaluation:&quot;)
        dev_step(x_dev, y_dev, writer=dev_summary_writer)
        print(&quot;&quot;)
    if current_step % FLAGS.checkpoint_every == 0:
        path = saver.save(sess, checkpoint_prefix, global_step=current_step)
        print(&quot;Saved model checkpoint to {}\n&quot;.format(path))
```

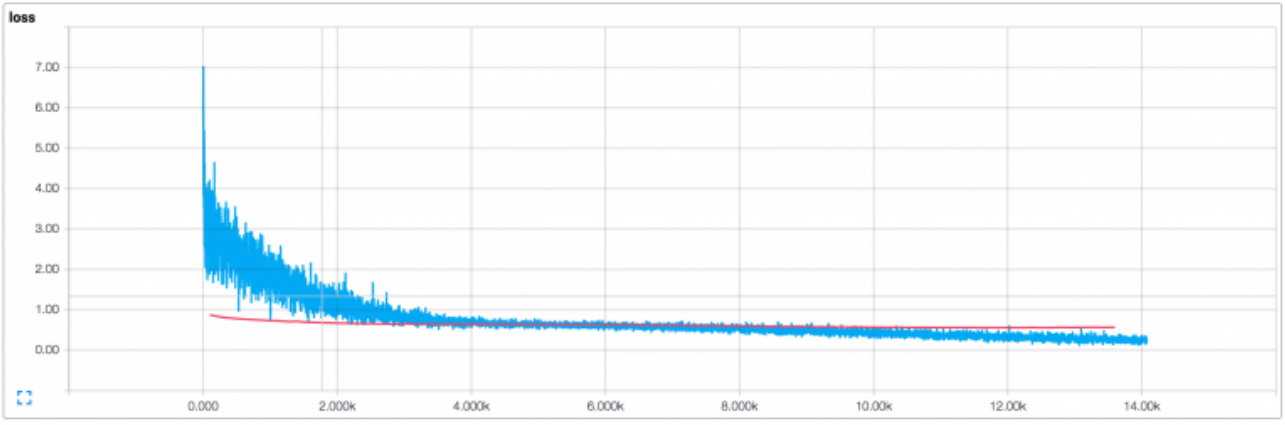
这里，batch_iter是我编写的批处理数据的辅助函数，tf.train.global_step是返回值的便捷函数global_step。**此处还提供完整的培训代码。**

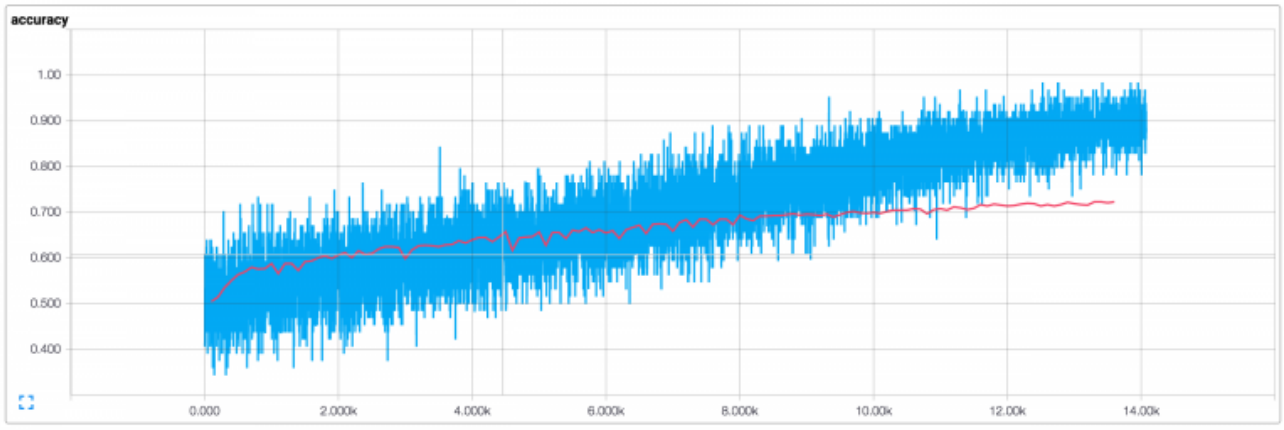
在TensorBoard中可视化结果

我们的训练脚本将摘要写入输出目录，通过将TensorBoard指向该目录，我们可以可视化图形和我们创建的摘要。

```
tensorboard --logdir /PATH_TO_CODE/runs/1449760558/summaries/
```

使用默认参数（128维嵌入，过滤器大小为3,4和5，每个过滤器大小丢失0.5和128个过滤器）运行训练过程会导致以下丢失和准确度图（蓝色是训练数据，红色是10％开发数据）。





有几件事情很突出：

- 我们的培训指标并不顺利，因为我们使用的是小批量。如果我们使用更大的批次（或在整个训练集上进行评估），我们将获得更平滑的蓝线。
- 由于开发精度明显低于训练精度，因此我们的网络似乎过度拟合训练数据，这表明我们需要更多数据（MR数据集非常小），更强的正则化或更少的模型参数。例如，我尝试在最后一层为权重添加额外的L2惩罚，并且能够将准确度提高到76％，接近原始论文中报告的。
- 由于应用了辍学，训练损失和准确性开始显着低于开发指标。

您可以使用代码并尝试使用各种参数配置运行模型。代码和说明可在Github上获得。

扩展和练习

以下是一些有用的练习，可以提高模型的性能：

- 使用预先训练的word2vec向量初始化嵌入。要完成这项工作，您需要使用300维嵌入并使用预先训练的值初始化它们。
- 与原始纸张一样，限制最后一层中权重向量的L2范数。您可以通过定义在每个训练步骤后更新权重值的新操作来完成此操作。

- 将L2正则化添加到网络以抵抗过度拟合，同时尝试提高丢失率。（Github上的代码已包含L2正则化，但默认情况下禁用）
- 添加权重更新和图层操作的直方图摘要，并在TensorBoard中显示它们。

请在评论中留下反馈和问题！

卷积神经网络，深度学习，神经网络，NLP