Tensorflow实现的CNN文本分类 - somTian的博客 - 博客频道

分类：
tensorflow（3）
deeplearning论文学习（3）
深度学习笔记（5）
目录(?)[+]
翻译自博客：IMPLEMENTING A CNN FOR TEXT CLASSIFICATION IN TENSORFLOW

在这篇文章中，我们将实现一个类似于Kim Yoon的卷积神经网络语句分类的模型。 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，并已成为新的文本分类架构的标准基准。

本文假设你已经熟悉了应用于NLP的卷积神经网络的基础知识。 如果没有，建议先阅读Understanding Convolutional Neural Networks for NLP 以获得必要的背景。

1. 数据和预处理

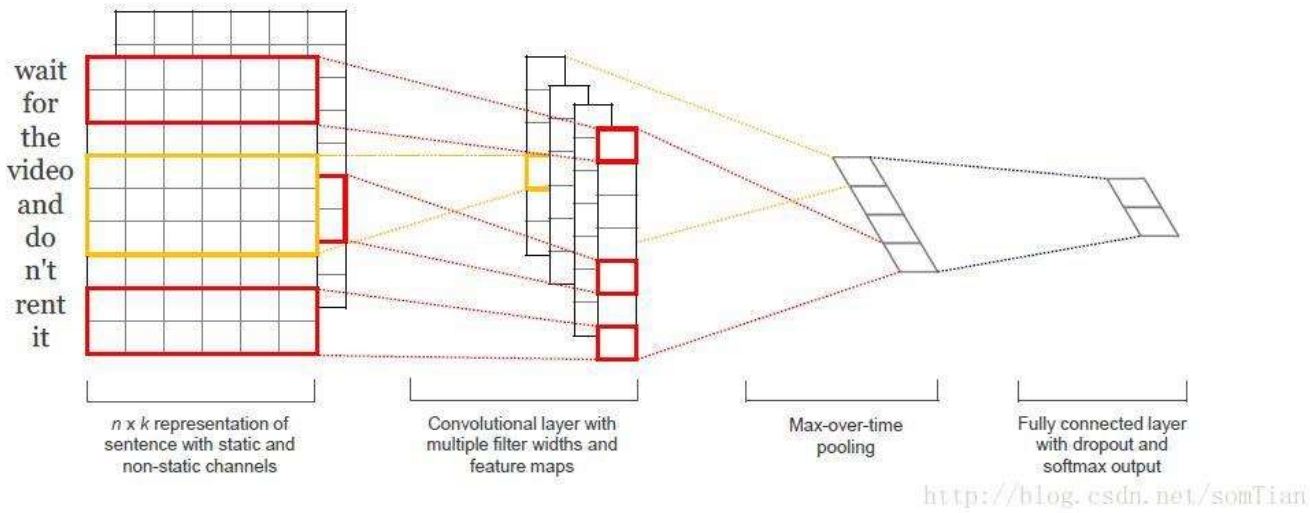
我们将在这篇文章中使用的数据集是 Movie Review data from Rotten Tomatoes，也是原始文献中使用的数据集之一。 数据集包含10,662个示例评论句子，正负向各占一半。 数据集的大小约为20k。 请注意，由于这个数据集很小，我们很可能会使用强大的模型。 此外，数据集不附带拆分的训练/测试集，因此我们只需将10%的数据用作 dev set。 原始文献展示了对数据进行10倍交叉验证的结果。

这里不讨论数据预处理代码，代码可以在 Github 上获得，并执行以下操作：

1. 从原始数据文件中加载正负向情感的句子。
2. 使用与原始文献相同的代码清理文本数据。
3. 将每个句子加到最大句子长度(59)。我们向所有其他句子添加特殊的操作，使其成为59个字。填充句子相同的长度是有用的，因为这样就允许我们有效地批量我们的数据，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（词库大小）。 每个句子都成为一个整数向量。

2. 模型

原始文献的网络结构如下图：



第一层将单词嵌入到低维向量中。 下一层使用多个过滤器大小对嵌入的字矢量执行卷积。 例如，一次滑过3，4或5个字。 接下来，我们将卷积层的max_pooling结果作为一个长的特征向量，添加dropout正则，并使用softmax层对结果进行分类。

因为这是一篇博客，所以对于原始文献的模型进行一下简化：

1. 我们不会对我们的词嵌入使用预先训练的word2vec向量。 相反，我们从头开始学习嵌入。
2. 我们不会对权重向量执行L2规范约束。 A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification 发现约束对最终结果几乎没有影响。
3. 原始实验用两个输入数据通道 - 静态和非静态字矢量。 我们只使用一个通道。

将这些扩展代码添加到这里是比较简单的（几十行代码）。 看看帖子结尾的练习。

3. 代码实现

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

```
import tensorflow as tf
import numpy as np

class TextCNN(object):
    def __init__(self,sequence_length, num_classes, vocab_size,
      embedding_size, filter_sizes, num_filters):
```

为了实例化类，我们传递以下参数：

- sequence_length - 句子的长度。注意：我们将所有句子填充到相同的长度（我们的数据集为59）。
- num_classes - 输出层中的类数，在我们的例子中为正（负）。
- vocab_size - 我们的词汇量的大小。 这需要定义我们的嵌入层的大小，它将具有[vocabulary_size，embedding_size]的形状。
- embedding_size - 嵌入的维度。

- filter_sizes - 我们想要卷积过滤器覆盖的字数。 我们将为此处指定的每个大小设置num_filters。 例如，[3，4，5]意味着我们将有一个过滤器，分别滑过3，4和5个字，总共有3 * num_filters过滤器。
- num_filters - 每个过滤器大小的过滤器数量（见上文）。

3.1 INPUT PLACEHOLDERS

首先定义网络的输入数据

tf.placeholder创建一个占位符变量，当我们在训练集或测试时间执行它时，我们将其馈送到网络。 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。 在我们的情况下，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。

将神经元保留在丢失层中的概率也是网络的输入，因为我们仅在训练期间使用dropout退出。 我们在评估模型时禁用它（稍后再说）。

3.2 EMBEDDING LAYER

我们定义的第一层是嵌入层，它将词汇词索引映射到低维向量表示中。 它本质上是一个从数据中学习的lookup table。

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

我们在这里使用了几个功能：

- tf.device（“/ cpu：0”）强制在CPU上执行操作。 默认情况下，TensorFlow将尝试将操作放在GPU上（如果有的话）可用，但是嵌入式实现当前没有GPU支持，并且如果放置在GPU上会引发错误。
- tf.name_scope创建一个名称范围，名称为“embedding”。 范围将所有操作添加到名为“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是我们在训练中学习的嵌入矩阵。 我们使用随机均匀分布来初始化它。 tf.nn.embedding_lookup创建实际的嵌入操作。 嵌入操作的结果是形状为[None，sequence_length，embedding_size]的三维张量。

TensorFlow的卷积转换操作具有对应于批次，宽度，高度和通道的尺寸的4维张量。 我们嵌入的结果不包含通道尺寸，所以我们手动添加，留下一层shape为[None，sequence_length，embedding_size，1]。

3.3 CONVOLUTION AND MAX-POOLING LAYERS

现在我们已经准备好构建卷积层，然后再进行max-pooling。 注意：我们使用不同大小的filter。 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，然后将结果合并成一个大特征向量。

```
pooled_outputs = []
for i,filter_size in enumerate(filter_sizes):
    with tf.name_scope("conv-maxpool-%s" %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="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")
        # 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="pool"
        )
        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。 每个过滤器在整个嵌入中滑动，但是它涵盖的字数有所不同。 “VALID”填充意味着我们在没有填充边缘的情况下将过滤器滑过我们的句子，执行给我们输出形状[1，sequence_length - filter_size + 1,1,1]的窄卷积。 在特定过滤器大小的输出上执行最大值池将留下一张张量的形状[batch_size，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。 一旦我们从每个过滤器大小得到所有的汇总输出张量，我们将它们组合成一个长形特征向量[batch_size，num_filters_total]。 在tf.reshape中使用-1可以告诉TensorFlow在可能的情况下平坦化维度。

3.4 DROPOUT LAYER

Dropout可能是卷积神经网络正则最流行的方法。Dropout背后的想法很简单。Dropout层随机地“禁用”其神经元的一部分。 这可以防止神经元共同适应（co-adapting），并迫使他们学习个别有用的功能。 我们保持启用的神经元的分数由我们网络的dropout_keep_prob输入定义。 在训练过程中，我们将其设置为0.5，在评估过程中设置为1（禁用Dropout）。

3.5 SCORES AND PREDICTIONS

使用max-pooling（with dropout ）的特征向量，我们可以通过执行矩阵乘法并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归一化概率，但这不会改变我们的最终预

测。

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

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

3.6 LOSS AND ACCURACY

使用分数我们可以定义损失函数。 损失是对我们网络错误的衡量，我们的目标是将其最小化。分类问题的标准损失函数是交叉熵损失 cross-entropy loss。

```
# Calculate mean cross-entropy loss
with tf.name_scope("loss"):
    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是一个方便的函数，计算每个类的交叉熵损失，给定我们的分数和正确的输入标签。 然后求损失的平均值。 我们也可以使用总和，但这比较难以比较不同批量大小和训练/测试集数据的损失。

我们还为精度定义一个表达式，这是在训练和测试期间跟踪的有用数值。

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

3.7 TRAINING PROCEDURE

在我们为网络定义训练程序之前，我们需要了解一些关于TensorFlow如何使用Sessions和Graphs的基础知识。如果您已经熟悉这些概念，请随时跳过本节。

在TensorFlow中， Session是正在执行graph 操作的环境，它包含有关变量和队列的状态。每个Session都在单个graph上运行。如果在创建变量和操作时未明确使用 Session，则使用TensorFlow创建的当前默认 Session。您可以通过在session.as_default（）块中执行命令来更改默认 Session（见下文）。

Graph包含操作和张量。您可以在程序中使用多个Graph，但大多数程序只需要一个Graph。您可以在多个Session中使用相同的Graph，但在一个 Session中不能使用多Graph。 TensorFlow始终创建一个默认

Graph，但您也可以手动创建一个Graph，并将其设置为新的默认Graph，如下图所示。显式创建 Session和Graph可确保在不再需要资源时正确释放资源。

```
FLAGS = tf.flags.FLAGS
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():
```

当优选设备不存在时，allow_soft_placement设置允许TensorFlow回退到具有特定操作的设备上。 例如，如果我们的代码在GPU上放置一个操作，并且我们在没有GPU的机器上运行代码，则不使用allow_soft_placement将导致错误。 如果设置了log_device_placement，TensorFlow会登录哪些设备（CPU或GPU）进行操作。 这对调试非常有用。 标记是我们程序的命令行参数。

3.8 INSTANTIATING THE CNN AND MINIMIZING THE LOSS

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置在上面创建的默认图和会话中。

```
cnn = TextCNN(
    sequence_length=x_train.shape[1],
    num_classes=y_train.shape[1],
    vocab_size=len(vocab_processor.vocabulary)
    embedding_size=FLAGS.num_filters,
    filter_sizes = map(int, FLAGS.filter_sizes.split(",")),
    num_filters = FLAGS.num_filters)
```

接下来，我们定义如何优化网络的损失函数。 TensorFlow有几个内置优化器。 我们正在使用Adam优化器。

```
# Define Training procedure
global_step = tf.Variable(0,name="global_step",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)
```

在这里，train_op这里是一个新创建的操作，我们可以运行它们来对我们的参数执行更新。 train_op的每次执行都是一个训练步骤。 TensorFlow自动计算哪些变量是“可训练的”并计算它们的梯度。 通过定义一个global_step变量并将其传递给优化器，让TensorFlow对训练步骤进行计数。 每次执行train_op时，global step 将自动递增1。

3.9 SUMMARIES

TensorFlow有一个概述（summaries），可以在训练和评估过程中跟踪和查看各种数值。 例如，您可能希望跟踪您的损失和准确性随时间的变化。您还可以跟踪更复杂的数值，例如图层激活的直方图。summaries是序列化对象，并使用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, "runs", timestamp))
print("Writing to {}\n".format(out_dir))

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

# Train Summaries
train_summary_op = tf.merge_summary([loss_summary, acc_summary])
train_summary_dir = os.path.join(out_dir, "summaries", "train")
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, "summaries", "dev")
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总结。 在我们的情况下，这些数值是相同的，但是您可能只有在训练过程中跟踪的数值（如参数更新值）。 tf.merge_summary是将多个摘要操作合并到可以执行的单个操作中的便利函数。

3.10 CHECKPOINTING

通常使用TensorFlow的另一个功能是checkpointing- 保存模型的参数以便稍后恢复。Checkpoints 可用于在以后的时间继续训练，或使用 early stopping选择最佳参数设置。 使用Saver对象创建Checkpoints。

```
# Checkpointing
checkpoint_dir = os.path.abspath(os.path.join(out_dir, "checkpoints"))
checkpoint_prefix = os.path.join(checkpoint_dir, "model")
# 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())
```

3.11 INITIALIZING THE VARIABLES

在训练模型之前，我们还需要在图中初始化变量。

```
# Initialize all variables
sess.run(tf.global_variables_initializer())
```

global_variables_initializer函数是一个方便函数，它运行我们为变量定义的所有初始值。也可以手动调用变量的初始化程序。 如果希望使用预先训练的值初始化嵌入，这很有用。

3.12 DEFINING A SINGLE TRAINING STEP

现在我们来定义一个训练步骤的函数，评估一批数据上的模型并更新模型参数。

```
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
    )
    time_str = datetime.datetime.now().isoformat()
    print("{}:step{},loss{:g},acc{:g}".format(time_str,step,loss,accuracy))
    train_summary_writer.add_summary(summaries,step)
```

feed_dict包含我们传递到我们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将抛出错误。使用输入数据的另一种方法是使用队列，但这超出了这篇文章的范围。

接下来，我们使用session.run执行我们的train_op，它返回我们要求它进行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，如果批量太小，训练批次的损失和准确性可能会在批次间显着变化。而且因为我们使用dropout，您的训练指标可能开始比您的评估指标更糟。

我们写一个类似的函数来评估任意数据集的丢失和准确性，例如验证集或整个训练集。本质上这个功能与上述相同，但没有训练操作。它也禁用退出。

```
def dev_step(x_batch, y_batch, writer=None):
    """
    Evaluates model on a dev set
    """
    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("{}: step {}, loss {:g}, acc {:g}".format(time_str, step, loss, accuracy))
    if writer:
        writer.add_summary(summaries, step)
```

3.13 TRAINING LOOP

最后，准备编写训练循环。 迭代数据的批次，调用每个批次的train_step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("\nEvaluation:")
        dev_step(x_dev, y_dev, writer=dev_summary_writer)
        print("")
    if current_step % FLAGS.checkpoint_every == 0:
        path = saver.save(sess, checkpoint_prefix, global_step=current_step)
        print("Saved model checkpoint to {}\n".format(path))
```

这里，batch_iter是一个批处理数据的帮助函数，而tf.train.global_step是返回global_step值的便利函数。

3.14 VISUALIZING RESULTS IN TENSORBOARD

我们的训练脚本将summaries写入输出目录，并将TensorBoard指向该目录，我们可以将图和我们创建的summaries可视化。

```
tensorboard --logdir `/path/`
```

有几件事情脱颖而出：

- 我们的训练指标并不平滑，因为我们使用小批量。 如果我们使用较大的批次（或在整个训练集上评

估），我们会得到一个更平滑的蓝线。

- 因为测试者的准确性显着低于训练准确度，我们的网络在训练数据似乎过拟合了，这表明我们需要更多的数据（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重量添加额外的L2正则，并且能够将准确度提高到76％，接近于原始文献。
- 因为使用了dropout，训练损失和准确性开始大大低于测试指标。

您可以使用代码进行操作，并尝试使用各种参数配置运行模型。 Github提供了代码和说明。

4. EXTENSIONS AND EXERCISES

以下是一些的练习，可以提高模型的性能：

- 使用预先训练的word2vec向量初始化嵌入。 为了能够起作用，您需要使用300维嵌入，并用预先训练的值初始化它们。
- 限制最后一层权重向量的L2范数，就像原始文献一样。 您可以通过定义一个新的操作，在每次训练步骤之后更新权重值。
- 将L2正规化添加到网络以防止过拟合，同时也提高dropout比率。 （Github上的代码已经包括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summaries，并在TensorBoard中进行可视化。

顶
0