Amazon Reviews for Sentiment Analysis

该数据集包含数百万条亚马逊客户评论（输入文本）和星级评分（输出标签），用于学习如何训练 fastText 进行情感分析。

这里的想法是，数据集不仅仅是一个玩具——合理规模的真实业务数据——而且可以在一台普通的笔记本电脑上在几分钟内进行训练。

> `CuDNNLSTM` 是一种在深度学习中使用的长短期记忆网络（LSTM）的变体
> 专门优化以在 NVIDIA GPU 上运行

1. **读取和预处理数据：**
   - 使用 bz2 库读取压缩的文本文件。
   - 将文本行解码成 UTF-8 格式，并进行初步的处理，如标签提取和文本清洗。
2. **文本数据清洗：**
   - 清洗包括将数字替换为0，以及将网址替换为 "<url>"。

将文本数据转换为深度学习模型可以处理的格式

1. **设置参数：**
   - `max_features` 是词汇表的大小，即最多考虑的单词数量，这里设为 20000。
   - `maxlen` 是文本序列的最大长度，这里设为 100。
     这意味着所有的文本序列将被截断或填充到这个长度。
2. **初始化和训练分词器：**
   - `tokenizer = text.Tokenizer(num_words=max_features)` 初始化一个 Keras 分词器，设置词汇表的最大大小。
   - `tokenizer.fit_on_texts(train_sentences)` 使用训练数据集 `train_sentences` 来训练分词器。
     这个步骤实际上是构建一个词汇表，
     将每个唯一的单词映射到一个唯一的整数。
3. **文本转换为序列：**
   - `tokenized_train = tokenizer.texts_to_sequences(train_sentences)` 将训练数据集中的每个文本转换为一系列整数。
     每个整数代表对应的单词在词汇表中的索引。
4. **序列填充和截断：**
   - `X_train = sequence.pad_sequences(tokenized_train, maxlen=maxlen)` 对转换后的序列进行填充或截断，使得它们的长度统一为 `maxlen`。
     填充通常在序列的开头进行，但也可以设置在末尾。
     这一步骤确保了所有输入到模型的文本数据都有相同的长度。

> 🔒 `X_train` 中看到很多0的原因？
>
> **填充序列：**
>
> - 当使用 `sequence.pad_sequences` 对序列进行填充以确保所有序列长度一致时，较短的序列会被0填充至指定的最大长度 (`maxlen`)。
>
> **限制词汇表大小（`max_features`）：**
>
> - `max_features` 参数限制了词汇表的大小。如果一个单词的索引超过了这个限制，它将不会被包含在最终的词汇表中。在文本转换为序列的过程中，不在词汇表中的单词通常会被忽略或替换为特定的标记（如未知词标记）。然而，这并不会直接导致0的增加，除非您的处理逻辑是将这些词替换为0。

3. **构建嵌入矩阵：**

▼ Code

```
EMBEDDING_FILE = '../input/glovetwitter100d/glove.twitter.27B.100d.txt'
def get_coefs(word, *arr):
    return word, np.asarray(arr, dtype='float32')

embeddings_index = dict(get_coefs(*o.rstrip().rsplit(' ')) for o in open(EMBEDDING_FILE))
all_embs = np.stack(embeddings_index.values())
emb_mean,emb_std = all_embs.mean(), all_embs.std()
embed_size = all_embs.shape[1]

word_index = tokenizer.word_index
nb_words = min(max_features, len(word_index))
#change below line if computing normal stats is too slow
embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size)) #embedding_
for word, i in word_index.items():
    if i >= max_features: continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None: embedding_matrix[i] = embedding_vector
```

4. **定义模型结构：**

- 使用 Keras 定义了一个包含 CuDNNLSTM 层的深度学习模型。模型包括嵌入层、卷积层、LSTM层和全连接层。

🔒 **定义模型参数：**

- `batch_size`：每批训练数据的大小，这里设为 2048。
- `epochs`：训练过程中整个数据集被遍历的次数，这里设为 7。
- `embed_size`：嵌入层中每个单词的向量维度，设为 100。
- `gc.collect()`：调用 Python 的垃圾回收器来释放内存。

1. **构建模型：**
   - `inp = Input(shape=(maxlen, ))`：定义模型的输入层，`maxlen` 是输入序列的长度。
   - `x = Embedding(max_features, embed_size, weights=[embedding_matrix], trainable=True)(inp)`：嵌入层，将输入的整数序列转换为嵌入向量。使用预先定义的嵌入矩阵 `embedding_matrix` 作为权重，这些权重在训练过程中是可训练的。
   - `x = Dropout(0.25)(x)`：Dropout层，以 0.25 的概率随机丢弃输入的一部分特征，防止过拟合。
   - `x = Conv1D(2*embed_size, kernel_size = 3)(x)`：一维卷积层，用于提取序列中的局部特征。
   - `prefilt = Conv1D(2*embed_size, kernel_size = 3)(x)`：另一个一维卷积层，紧接着前一个卷积层。
   - `for` 循环中的 `Conv1D`：使用不同的步长应用多个卷积层，以进一步提取特征。
   - `CuDNNLSTM` 层：
     两个 LSTM 层，一个处理正向信息（`x_f`），另一个处理反向信息（`x_b`）。这些层用于捕捉序列数据中的长期依赖关系。
   - `x = concatenate([x_f, x_b])`：将正向和反向 LSTM 层的输出合并。
   - `Dense(64, activation="relu")(x)`：全连接层，用 ReLU 激活函数，提供非线性转换。
   - `Dense(1, activation="sigmoid")(x)`：最后的全连接层，使用 sigmoid 激活函数，用于二分类问题的输出。
2. **编译模型：**
   - 使用 `binary_crossentropy` 作为损失函数，适用于二分类问题。
   - 优化器选择为 `adam`。
   - 性能指标使用 `binary_accuracy`。
3. **模型概览：**
   - `cudnnlstm_model.summary()`：输出模型的结构和参数。
4. **定义模型参数：**
   - `batch_size`：每批训练数据的大小，这里设为 2048。
   - `epochs`：训练过程中整个数据集被遍历的次数，这里设为 7。
   - `embed_size`：嵌入层中每个单词的向量维度，设为 100。
   - `gc.collect()`：调用 Python 的垃圾回收器来释放内存。
5. **构建模型：**
   - `inp = Input(shape=(maxlen, ))`：定义模型的输入层，`maxlen` 是输入序列的长度。
   - `x = Embedding(max_features, embed_size, weights=[embedding_matrix], trainable=True)(inp)`：嵌入层，将输入的整数序列转换为嵌入向量。使用预先定义的嵌入矩阵 `embedding_matrix` 作为权重，这些权重在训练过程中是可训练的。
   - `x = Dropout(0.25)(x)`：Dropout层，以 0.25 的概率随机丢弃输入的一部分特征，防止过拟合。
   - `x = Conv1D(2*embed_size, kernel_size = 3)(x)`：一维卷积层，用于提取序列中的局部特征。
   - `prefilt = Conv1D(2*embed_size, kernel_size = 3)(x)`：另一个一维卷积层，紧接着前一个卷积层。
   - `for` 循环中的 `Conv1D`：使用不同的步长应用多个卷积层，以进一步提取特征。
   - `CuDNNLSTM` 层：两个 LSTM 层，一个处理正向信息（`x_f`），另一个处理反向信息（`x_b`）。这些层用于捕捉序列数据中的长期依赖关系。
   - `x = concatenate([x_f, x_b])`：将正向和反向 LSTM 层的输出合并。

- `Dense(64, activation="relu")(x)`：全连接层，用 ReLU 激活函数，提供非线性转换。
- `Dense(1, activation="sigmoid")(x)`：最后的全连接层，使用 sigmoid 激活函数，用于二分类问题的输出。

6. **编译模型**：
   - 使用 `binary_crossentropy` 作为损失函数，适用于二分类问题。
   - 优化器选择为 `adam`。
   - 性能指标使用 `binary_accuracy`。
7. **模型概览**：
   - `cudnnlstm_model.summary()`：输出模型的结构和参数。

5. **模型编译和训练**：
   - 编译模型并使用训练数据进行训练。同时使用了回调函数，如模型检查点和提前停止，以优化训练过程。

> 🔒 **回调函数**
>
> 在训练过程中，可以使用回调函数来执行特定的任务，例如保存模型、提前停止训练等。
>
> - **模型检查点（ModelCheckpoint）**：这个回调函数在每个 epoch 结束后保存模型。通常，我们只保存在验证集上表现最好的模型。这就是 `save_best_only=True` 的作用。
> - **提前停止（EarlyStopping）**：这个回调函数可以在模型的验证损失在连续几个 epoch 中没有改善时提前终止训练。这有助于避免过拟合，并节省计算资源。

6. **模型评估**：
   - 在测试集上评估模型性能，打印测试得分和准确率。