基于 LSTM 的语言模型

汪仁喜

计算机 1901

摘 要 语言模型，就是对语言进行建模的模型。自然语言处理中讨论的语言模型，通常是指给定一句话的前 n 个单词，预测下一个单词概率分布的模型。由此首先出现了基于统计的语言模型，其中的典型是 n-gram 模型。之后又出现了基于神经网络的语言模型，例如循环神经网络(RNN)，长短期记忆模型(LSTM)。神经语言模型在自然语言处理任务中取得了巨大的成功。本项目主要完成了基于 pytorch 的深层 LSTM 语言模型，并对比了不同初始化方法，不同层数对训练的影响。

1 介绍

语言是传递信息的最自然的载体。人类对于语言的处理自古就有之。从两千年前的凯撒密码，到古代的诗歌、谜语，再到近代以来广泛应用于国防领域的语言加密技术，无不体现着人们对于语言的处理和应用。而随着计算机的发展和兴起，让计算机读懂语言，进而建立起具有人类智慧的机器模型，成了人类的目标。而让机器读懂语言的前提，就是对语言进行建模。

在自然语言处理领域，语言模型被定义为一个概率模型。对语言进行建模，就是在给定一句话的前 n 个单词的情况下，对下一个可能出现的单词建立概率模型。神经网络兴起之前，应用最广泛的便是基于统计的语言模型。例如 n-gram 模型，将在前 n-1 个单词出现的情况下，第 n 个单词出现的统计频率，作为这个单词的概率。利用大数定理可以知道，在数据量足够大的情况下，n-gram 得到的概率等于真实的概率。但现实情况是，语料库的数据量不足以建立一个完全的 n-gram 模型，基于统计的模型也没有考虑到语义之间的联系。随着词向量，循环神经网络的出现和成功应用，人们也将其应用到了语言模型的建立。其中为了解决神经网络记忆力短的问题而出现的长短期记忆模型(LSTM)获得了巨大的成功。本项目在深度学习

框架 pytorch 的基础上，实现了任意层深度的 LSTM 模型。

2 实现

2.1 LSTM 模型

一般的循环神经网络存在记忆力短等问题。由于循环神经网络通过循环计算实现前向传播，也就造成了反向传播时梯度消失的问题。为了使神经网络具有更长的记忆，LSTM 在模型中引入了门的概念：分别建立了“遗忘门”，“输入门”，“输出门”。其模型如图 1 所示。LSTM 前向传播的算法如下：

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

$$f_t = \sigma\left(w_f[h_{t-1}, x_t] + b_f\right)$$

$$g_t = tanh\left(w_g[h_{t-1}, x_t] + b_g\right)$$

$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot g_t$$

$$h_t = o_t \cdot tanh(c_t)$$

其中，i 是输入门，f 是遗忘门，o 是输出门。下标 t 表示第 t 个时间步，h 表示隐藏层状态，σ 表示 sigmoid 函数。LSTM 通过对记忆的显式建模，使模型获得了较好的记忆能力。

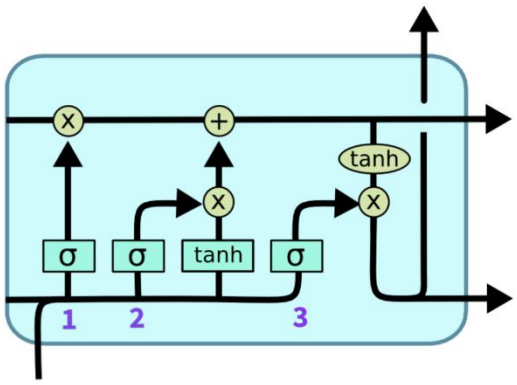


图 1 LSTM 模型，其中的 1，2，3 分别代表三个门

2.2 深层循环神经网络

LSTM 以及其他的循环神经网络，都可通过“堆叠”的方式，获得更深层的神经网络。一般而言，网络越深，提取到的特征也就越高级，从而效果也就越好。深层 RNN 模型如图 2 所示。深层 RNN 的前向传播计算，采用从左至又，从浅至深的方式。当某一单元所需要的上一层输入和前一时间步的隐藏层状态已被计算出时，就可对该单元进行计算。最终的输出结果为最深层的隐藏层状态。项目的代码实现中，采用循环机制，首先计算浅层网络，得到所有隐层状态后将其作为输入传递给深层网络。

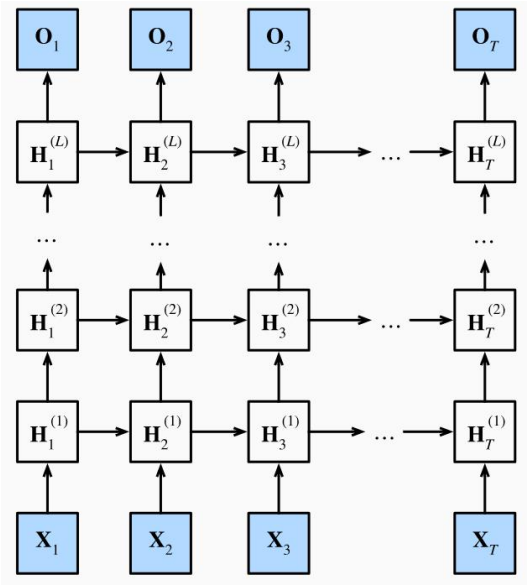


图 2 深层循环神经网络模型，其中 x 为输入，H 为隐藏层状态，o 为输出

3 实验

3.1 正确性

在将 LSTM 模型插入语言模型之前，需要验证模型的正确性。由于模型中的参数众多，且计算复杂，本次项目中采取了较为粗糙的方式来验证模型正确性：将模型输出与 PyTorch 库实现的 LSTM 输出进行对比，在初始参数相同的情况下，使用 torch.equal 函数对输出进行比较，若输出相等，可认为模型是正确的。验证正确性的代码保存在 lstm.py 文件中。

3.2 模型训练

3.2.1 数据

项目中采用的训练集大小为 505KB，共 4206 行，每行包含一句话。验证集大小 37KB，共 337 行，每行包含一句话。所有的数据均为英文语句，已经通过空格划分。所有数据在 data/dataset 目录下。

3.2.2 训练过程

训练过程首先需要读取数据，并将读取进来的数据转换为 token，并合并为一个批(batch)。每个批内都使用一个特殊的 padding 将句子填充为等长。训练时将一个批输入模型进行前向传播，得到输出后与目标计算交叉熵作为损失，同时输出模型的损失和困惑度(perplexity)。

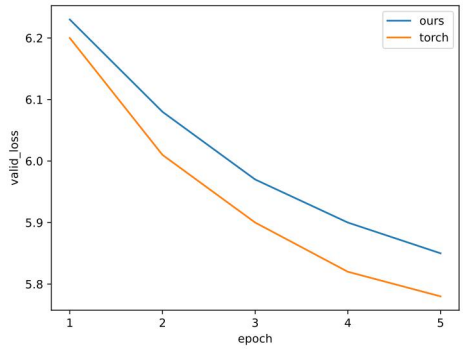
4 评估

4.1 正确性

在相同参数的情况下，项目实现的 LSTM 输出与 PyTorch 库实现的 LSTM 输出相同，可以认为模型是正确的，具体输出参照附录。注意在项目的实现中，将 batch 维度放在第一位，对应 PyTorch 中 LSTM 的 batch_first=True 选项。

4.2 结果比较

实验中采用轮数为 5，学习率为 0.0005，均采用默认初始化，层数均为 1 层，得到的结果如图 3 所示。



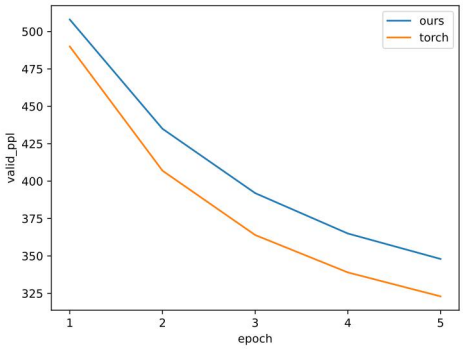
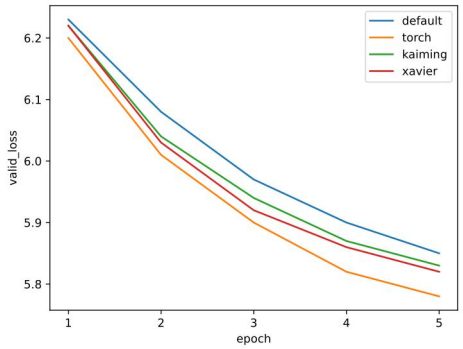


图 3 本项目实现的 LSTM 和 torch 库中 LSTM 在验证集上损失和困惑度对比

4.3 初始化

比较在测试集上的损失和困惑度可以发现，项目实现的 LSTM 均略差于 PyTorch 库中的 LSTM 模型，由于已经验证过模型的正确性，且 PyTorch 中模型的初始化大多是 kaiming 初始化或 xavier 初始化，但线性层是均匀初始化，而本项目中的参数均由线性层实现，因此再比较不同初始化方法对模型的影响，结果如图 4。通过结果可以发现，kaiming 初始化和 xavier 初始化收敛速度要好于默认初始化，但同 PyTorch 库实现还有一定差距。



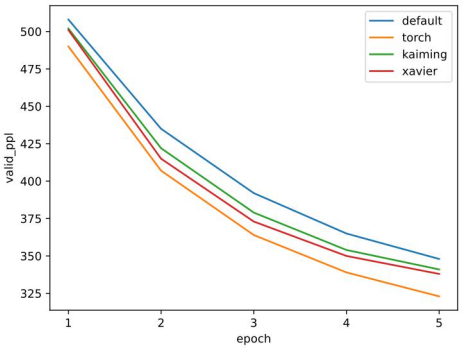
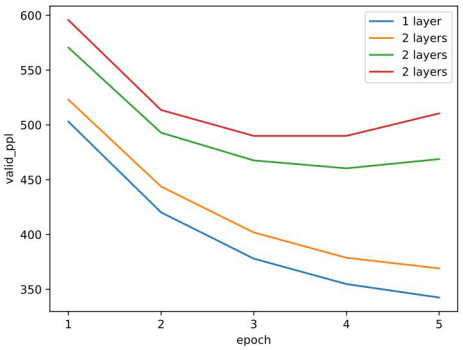


图 4 不同初始化方法和 torch 库中 LSTM 在验证集上损失和困惑度对比

4.3 模型深度

一般而言，模型深度对学习到的特征和学习效果有着较大影响。实验对不同深度模型进行了训练，结果如图 5 所示。



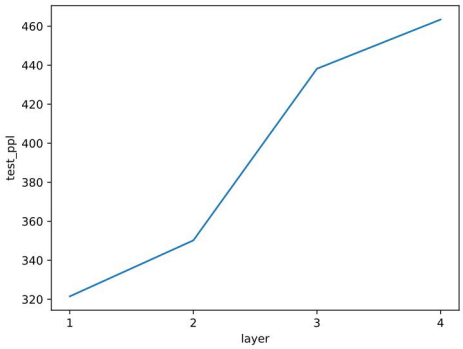


图 5 不同层数时的验证集困惑度和测试集困惑度

5 结论

本次项目采用 LSTM 模型对语言进行建模，实现了任意层深度的基于 LSTM 的语言模型。通过实验验证了模型的正确性，并通过对比实验，比较了不同初始化方法对收敛速度的影响。最后比较了不同深度语言模型的建模效果。

附录

1. 相同参数可输入时，模型输出和 PyTorch 库 LSTM 模型输出

```
tensor([[[0.0051, 0.0051, 0.0051, 0.0051, 0.0051, 0.0051],
         [0.0079, 0.0079, 0.0079, 0.0079, 0.0079, 0.0079]],
        [[0.0051, 0.0051, 0.0051, 0.0051, 0.0051, 0.0051],
         [0.0078, 0.0078, 0.0078, 0.0078, 0.0078, 0.0078]],
        [[0.0050, 0.0050, 0.0050, 0.0050, 0.0050, 0.0050],
         [0.0079, 0.0079, 0.0079, 0.0079, 0.0079, 0.0079]]],
       grad_fn=<StackBackward0>)
tensor([[[0.0051, 0.0051, 0.0051, 0.0051, 0.0051, 0.0051],
         [0.0079, 0.0079, 0.0079, 0.0079, 0.0079, 0.0079]],
        [[0.0051, 0.0051, 0.0051, 0.0051, 0.0051, 0.0051],
         [0.0078, 0.0078, 0.0078, 0.0078, 0.0078, 0.0078]],
        [[0.0050, 0.0050, 0.0050, 0.0050, 0.0050, 0.0050],
         [0.0079, 0.0079, 0.0079, 0.0079, 0.0079, 0.0079]]],
      grad_fn=<TransposeBackward0>)
```

input_size=4, hidden_size=6, 所有参数均初始化为 `0.01`，使用 batch_first 选项，即将 batch 维度放在第一维。