深度学习作业——神经网络语言模型

Clark

1 实验目的

本实验旨在基于 PyTorch 深度学习框架，构建并训练一个基于 LSTM（长短期记忆网络）的神经网络语言模型。通过本实验，主要达到以下目的：

1. 深入理解循环神经网络（RNN）及其变体 LSTM 在处理序列数据中的工作机制；
2. 掌握语言模型的基本原理，即如何根据历史上下文预测下一个词的概率分布；
3. 学习并实践文本数据的预处理流程，包括分词、构建词典、数据批处理（Batchify）等；
4. 掌握在训练过程中应用正则化技术（如 Locked Dropout、权重绑定）和优化策略（如梯度裁剪、学习率退火）以提升模型性能。

2 概述

语言模型是自然语言处理（NLP）中的核心任务之一，其目标是计算一个单词序列的概率。本实验使用经典的 Penn Treebank（PTB）数据集，构建一个多层 LSTM 网络来训练语言模型。实验涵盖了从数据下载、预处理、模型构建、训练循环到最终评估的全过程。通过计算困惑度（Perplexity, PPL）来衡量模型的性能，并分析不同训练策略对模型收敛速度和泛化能力的影响。

3 实验要求

1. 基于 Python 语言和 PyTorch 深度学习框架，完成 PTB 数据集的下载与读取；
2. 构建基于 LSTM 的语言模型，包含 Embedding 层、LSTM 层和全连接解码层；
3. 实现模型的训练与评估流程，使用困惑度（PPL）作为评价指标；
4. 应用 Dropout、梯度裁剪（Gradient Clipping）等技术防止过拟合和梯度爆炸；
5. 绘制训练损失和验证集困惑度的变化曲线，分析实验结果。

4 实验原理

4.1 语言模型

语言模型的目标是计算一个词序列 $w_1, w_2, \cdots, w_T$ 的联合概率 $P(w_1, w_2, \cdots, w_T)$。根据链式法则，该概率可以分解为条件概率的乘积：

$$P(w_1, w_2, \cdots, w_T) = \prod_{t=1}^{T} P(w_t | w_1, w_2, \cdots, w_{t-1}). \tag{1}$$

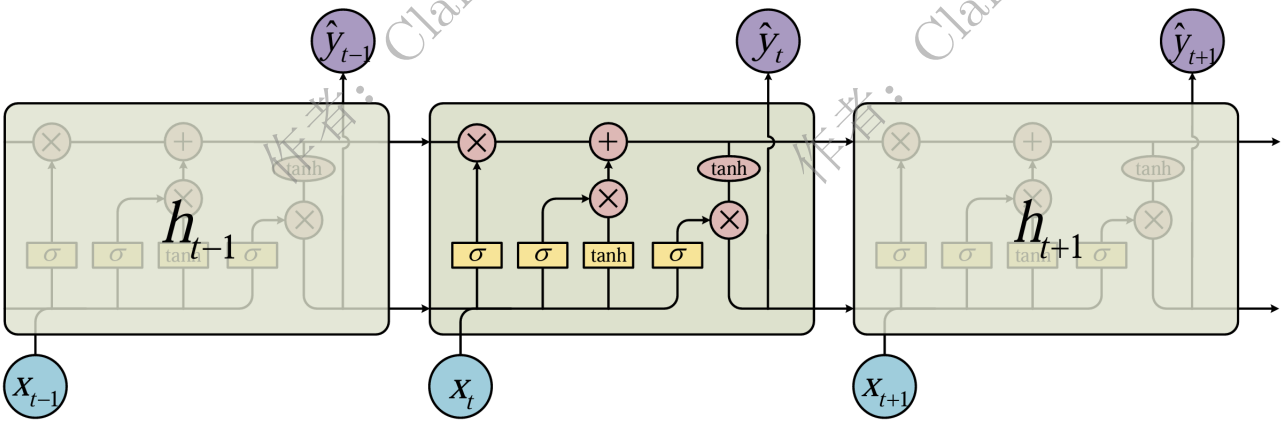
在神经网络语言模型中，我们通常使用 RNN 或 LSTM 来编码历史信息 $w_1, \cdots, w_{t-1}$ 为一个隐藏状态 h_{t-1}，然后预测下一个词 w_t 的概率分布。

4.2 长短期记忆网络 (LSTM)

LSTM 通过引入门控机制解决了传统 RNN 存在的梯度消失和梯度爆炸问题。一个标准的 LSTM 单元包含遗忘门 f_t、输入门 i_t 和输出门 o_t。其状态更新公式如下：

$$\begin{aligned}
i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi}), \\
f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf}), \\
g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg}), \\
o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho}), \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t, \\
h_t &= o_t \odot \tanh(c_t),
\end{aligned} \tag{2}$$

其中，x_t 是当前时刻的输入，h_{t-1} 是上一时刻的隐藏状态，c_t 是细胞状态，σ 是 Sigmoid 激活函数，$\odot$ 表示逐元素乘法。最后我们给出 LSTM 的网络结构图。



4.3 困惑度（Perplexity）

困惑度是评价语言模型好坏的常用指标，定义为交叉熵损失的指数形式：

$$\mathrm{PPL} = \mathrm{e}^{\mathrm{CrossEntropyLoss}} = \mathrm{e}^{-\frac{1}{N}\sum_{i=1}^{N}\ln P(w_i|w_{<i})}, \tag{3}$$

PPL 越低，表示模型对下一个词的预测越准确。

4.4 词嵌入与概率生成

在模型输入端，词嵌入层（Embedding Layer）将离散的单词索引映射为稠密的实值向量。数学上，这等价于一个矩阵乘法：

$$\boldsymbol{e}_t = \boldsymbol{W}_{\mathrm{emb}} \cdot \boldsymbol{x}_t, \tag{4}$$

其中 $\boldsymbol{x}_t \in \{0,1\}^{|V|}$ 是单词的 One-hot 向量，$\boldsymbol{W}_{\mathrm{emb}} \in \mathbb{R}^{d\times|V|}$ 是嵌入矩阵。在模型输出端，解码层将 LSTM 的隐藏状态 $\boldsymbol{h}_t$ 映射回词表空间，并通过 Softmax 函数生成概率分布：

$$P(w_{t+1}=k|w_{1:t}) = \frac{\mathrm{e}^{\boldsymbol{w}_k^T\boldsymbol{h}_t+b_k}}{\sum_{j=1}^{|V|}\mathrm{e}^{\boldsymbol{w}_j^T\boldsymbol{h}_t+b_j}}, \tag{5}$$

其中 $\boldsymbol{w}_k$ 是解码矩阵 $\boldsymbol{W}_{\mathrm{dec}}$ 的第 k 行。本实验中采用了权重绑定（Weight Tying）策略，即约束 $\boldsymbol{W}_{\mathrm{dec}} = \boldsymbol{W}_{\mathrm{emb}}^T$，这不仅减少了参数量，还赋予了嵌入向量更明确的语义解释。

4.5 随时间反向传播（BPTT）

循环神经网络的训练依赖于随时间反向传播算法（Backpropagation Through Time, BPTT）。BPTT 本质上是将 RNN 在时间维度上展开，将其视为一个层数等于时间步长 T 的深层前馈网络。总损失函数 L 是每个时间步损失 L_t 的总和：

$$L = \sum_{t=1}^{T} L_t(\theta), \tag{6}$$

其中 θ 是网络参数。参数的梯度是所有时间步梯度的累加：

$$\frac{\partial L}{\partial \theta} = \sum_{t=1}^{T}\frac{\partial L_t}{\partial \theta}. \tag{7}$$

在普通 RNN 中，梯度计算涉及连乘项 $\prod_{k=t}^{T}\frac{\partial h_k}{\partial h_{k-1}}$，这容易导致梯度消失或爆炸。LSTM 通过细胞状态 c_t 的加性更新规则（即 $c_t = f_t \odot c_{t-1} + \cdots$）引入了一条“恒等映射”路径，使得梯度能够更长时间地维持其幅度，从而有效缓解了梯度消失问题。

5 实验数据集及工具

5.1 数据预处理

实验使用 PTB 数据集。首先定义“Dictionary”类来构建词表，将单词映射为索引；“Corpus”类负责读取训练、验证和测试数据并进行 Tokenization。

```python
class Dictionary(object):
    def __init__(self):
self.word2idx = {}      self.idx2word = []

    def add_word(self, word):
        if word not in self.word2idx:
            self.idx2word.append(word)
            self.word2idx[word] = len(self.idx2word) - 1
        return self.word2idx[word]

    def __len__(self):
        return len(self.idx2word)

class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(os.path.join(path, 'ptb.train.txt'))
        self.valid = self.tokenize(os.path.join(path, 'ptb.valid.txt'))
        self.test = self.tokenize(os.path.join(path, 'ptb.test.txt'))

    def tokenize(self, path):
        """Tokenizes a text file."""
        assert os.path.exists(path)
        # Add words to the dictionary
        with open(path, 'r', encoding='utf-8') as f:
            tokens = 0
            for line in f:
                words = line.split() + ['<eos>']
                tokens += len(words)
                for word in words:
                    self.dictionary.add_word(word)

        # Tokenize file content
        with open(path, 'r', encoding='utf-8') as f:
            ids = torch.LongTensor(tokens)
            token = 0
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    ids[token] = self.dictionary.word2idx[word]
                    token += 1
        return ids
```

为了利用 GPU 进行并行计算，我们需要将数据整理成“batch_size”列的矩阵。“batchify”函数将长序

列数据切割并堆叠：

```
def batchify(data, bsz):
    # Work out how cleanly we can divide the dataset into bsz parts.
    nbatch = data.size(0) // bsz
    # Trim off any extra elements that wouldn't cleanly fit (remainders).
    data = data.narrow(0, 0, nbatch * bsz)
    # Evenly divide the data across the bsz batches.
    data = data.view(bsz, -1).t().contiguous()
    return data.to(device)
```

5.2 模型构建

本实验构建了一个基于多层 LSTM（Long Short-Term Memory）的神经网络语言模型。模型主要由三部分组成：编码器（Encoder）、LSTM 核心层和解码器（Decoder）。其中，LSTM 层负责捕捉序列数据中的长距离依赖关系。为了防止过拟合，模型特别采用了“LockedDropout”机制，这是一种专门针对 RNN 设计的 Dropout 变体。与在每个时间步独立采样掩码的标准 Dropout 不同，LockedDropout 在整个时间步序列中对相同的神经元应用相同的 Dropout 掩码，从而在破坏神经元间共适应性的同时，最大程度地保留了时间维度上的信息流。

```
class LockedDropout(nn.Module):
    def __init__(self):
        super(LockedDropout, self).__init__()

    def forward(self, x, dropout=0.5):
        if not self.training or not dropout:
            return x
        m = x.data.new(1, x.size(1), x.size(2)).bernoulli_(1 - dropout)
        mask = m.div_(1 - dropout)
        mask = mask.expand_as(x)
        return x * mask

class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers, dropout=0.5):
        super(LSTMModel, self).__init__()
        self.locked_drop = LockedDropout()
        self.drop = nn.Dropout(dropout)
        self.encoder = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, dropout=dropout)
        self.decoder = nn.Linear(hidden_size, vocab_size)

        # Tie weights if dimensions match
        if embed_size == hidden_size:
            self.decoder.weight = self.encoder.weight

```

```
        self.init_weights()
self.hidden_size = hidden_size        self.num_layers = num_layers

    def init_weights(self):
        initrange = 0.1
        self.encoder.weight.data.uniform_(-initrange, initrange)
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-initrange, initrange)

        # Initialize LSTM forget gate bias to 1
        for names in self.lstm._all_weights:
            for name in filter(lambda n: "bias" in n, names):
                bias = getattr(self.lstm, name)
                n = bias.size(0)
                start, end = n // 4, n // 2
                bias.data[start:end].fill_(1.)

    def forward(self, input, hidden):
        emb = self.encoder(input)
        emb = self.locked_drop(emb, dropout=self.drop.p)

        output, hidden = self.lstm(emb, hidden)

        output = self.locked_drop(output, dropout=self.drop.p)
        decoded = self.decoder(output.view(output.size(0)*output.size(1), output.size(2))
            )
        return decoded.view(output.size(0), output.size(1), decoded.size(1)), hidden

    def init_hidden(self, bsz):
        weight = next(self.parameters())
        return (weight.new_zeros(self.num_layers, bsz, self.hidden_size),
                weight.new_zeros(self.num_layers, bsz, self.hidden_size))
```

5.3 模型训练与评估

训练过程中使用了梯度裁剪（Gradient Clipping）来防止梯度爆炸，并实现了学习率退火策略。

```
def train(model, train_data, criterion, optimizer, epoch):
    model.train()
    total_loss = 0.
    start_time = time.time()
    hidden = model.init_hidden(BATCH_SIZE)

    batch_losses = []

```

```
    for batch, i in enumerate(range(0, train_data.size(0) - 1, SEQ_LEN)):
data, targets = get_batch(train_data, i)        # Starting each batch, we detach the
        hidden state from how it was previously produced.
        hidden = repackage_hidden(hidden)

        model.zero_grad()
        output, hidden = model(data, hidden)
        loss = criterion(output.view(-1, vocab_size), targets)
        loss.backward()

        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        torch.nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)
        optimizer.step()

        total_loss += loss.item()

        if batch % 200 == 0 and batch > 0:
            cur_loss = total_loss / 200
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches | lr {:02.5f} | ms/batch {:5.2f} |
                '
                  'loss {:5.2f} | ppl {:8.2f}'.format(
                    epoch, batch, len(train_data) // SEQ_LEN, LEARNING_RATE,
                    elapsed * 1000 / 200, cur_loss, math.exp(cur_loss)))
            batch_losses.append(cur_loss)
            total_loss = 0
            start_time = time.time()

    return batch_losses
```

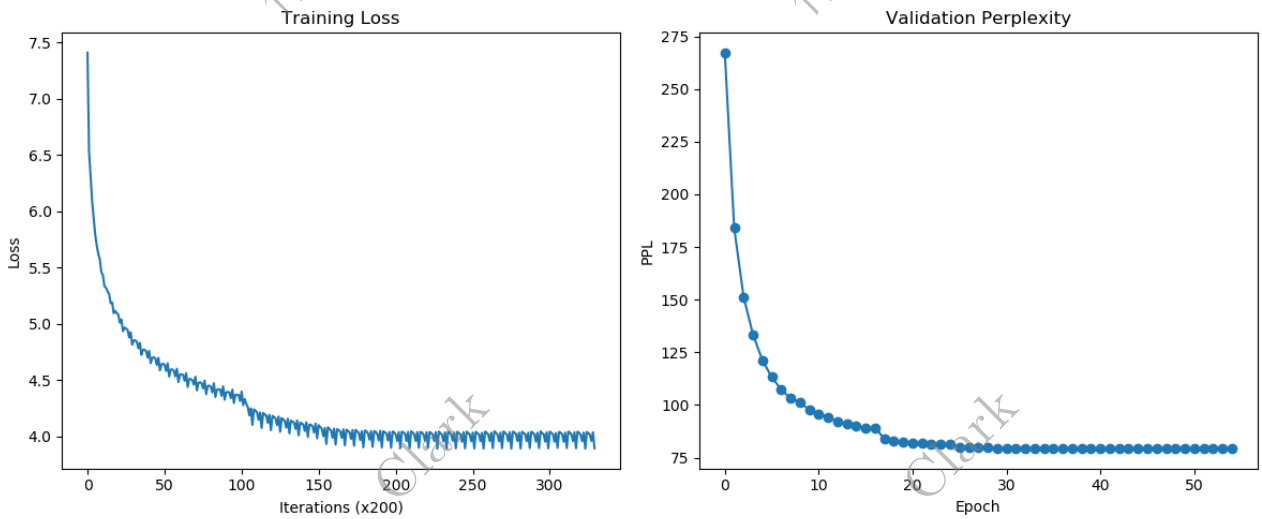
6 实验结果与分析

6.1 训练过程分析

最后实验的结果如下图所示，我们经过 55 个 Epoch 的训练，模型在训练集上的 ppl 得分为 79.22，损失函数为 4.37；在测试集上的 ppl 得分为 75.88，损失函数为 4.33。

```
-----------------------------------------------------------------------------------------
| epoch  55 |   200/ 1327 batches | lr 20.00000 | ms/batch 17.19 | loss  4.04 | ppl    56.66
| epoch  55 |   400/ 1327 batches | lr 20.00000 | ms/batch 17.12 | loss  4.03 | ppl    56.16
| epoch  55 |   600/ 1327 batches | lr 20.00000 | ms/batch 17.16 | loss  4.01 | ppl    55.24
| epoch  55 |   800/ 1327 batches | lr 20.00000 | ms/batch 17.19 | loss  3.96 | ppl    52.29
| epoch  55 |  1000/ 1327 batches | lr 20.00000 | ms/batch 17.19 | loss  4.03 | ppl    56.50
| epoch  55 |  1200/ 1327 batches | lr 20.00000 | ms/batch 17.21 | loss  3.89 | ppl    49.01
-----------------------------------------------------------------------------------------
| end of epoch  55 | time: 23.27s | valid loss  4.37 | valid ppl    79.22
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | test loss  4.33 | test ppl    75.88
=========================================================================================
Plot saved to result.png
chenpeng@amax:~/test$
```

实验设置了 55 个 Epoch，Batch Size 为 20，序列长度为 35。从生成的训练曲线可以看出，随着训练的进行，Training Loss 呈现稳步下降趋势，表明模型正在有效地拟合训练数据。同时，Validation Perplexity（验证集困惑度）也随之降低，说明模型的泛化能力在不断增强。



6.2 关键改进措施

- Locked Dropout：与标准的 Dropout 不同，Locked Dropout 在每个时间步采样相同的 Dropout 掩码。这对于 RNN 尤为重要，因为它避免了破坏循环连接中的长期依赖关系，从而允许模型更有效地学习长序列信息。
- 权重绑定（Weight Tying）：在“LSTMModel”中，当“embed_size”等于“hidden_size”时，我们将 Embedding 层的权重与输出解码层（Linear）的权重共享。这不仅显著减少了模型的参数量，降低了过拟合风险，还利用了 Embedding 层学到的语义信息来辅助解码。

- 梯度裁剪（Gradient Clipping）：在“train”函数中使用了“torch.nn.utils.clip_grad_norm_”，设定阈值为 0.25。这一措施有效防止了 RNN 训练中常见的梯度爆炸问题，确保了训练过程的数值稳定性。
- 学习率退火（Learning Rate Annealing）：在主循环中，如果验证集 Loss 没有下降，则将学习率除以 4。这种动态调整策略使得模型在训练初期能快速收敛，而在后期能进行更精细的参数搜索，从而达到更优的局部极小值。

7 实验总结

本次实验成功构建并训练了一个基于 LSTM 的神经网络语言模型。通过对 PTB 数据集的处理和分析，验证了 LSTM 在序列建模任务中的有效性。实验结果表明，结合 Locked Dropout、权重绑定和梯度裁剪等优化技术，模型能够达到较低的困惑度（Perplexity），具备良好的预测能力。

实验过程中，我深入理解了语言模型的评价指标 PPL 的物理含义，以及如何通过 Batchify 技术高效利用 GPU 资源。此外，通过对比不同正则化手段的作用，进一步掌握了深度学习模型调优的方法论。这些经验为后续进行更复杂的自然语言处理任务（如机器翻译、文本生成）奠定了坚实的基础。