实验 7 神经网络语言模型实验指导书

一、 实验目的

1. 掌握循环神经网络结构，包括 LSTM 等。
2. 掌握使用深度学习框架构建循环神经网络模型的方法。

二、 实验要求

1. 使用深度学习框架（本实验指导书以 TensorFlow 为例）构建一个规范的 LSTM 网络。
2. 在 PTB（Penn Treebank）语料库上进行神经网络语言模型的训练和评估，评价指标困惑度（Perplexity，PPL）低于 80。
3. 如果选择做此实验作业，按规定时间在课程网站提交实验报告、代码以及 PPT。

三、 实验原理

1. 语言模型

语言模型是一个为某一段词序列分配概率的模型。它对多种自然语言处理任务都有帮助。例如，在机器翻译任务中，需要由语言模型为系统输出打分，以提高输出目标语言输出的流畅性。在语音识别任务中，语言模型与声学模型一起预测下一个词。语言模型用来计算一个具有 N 个词的词序列概率，即：

$$\begin{aligned} P(s) &= P(w_1 w_2 \cdots w_N) \\ &= P(w_1 w_2 \cdots w_{N-1}) P(w_N \mid w_1 w_2 \cdots w_{N-1}) \\ &= P(w_1) P(w_2 \mid w_1) \cdots P(w_N \mid w_1 w_2 \cdots w_{N-1}) \end{aligned}$$

上式可知，它也可以分解成对给定前缀（一般叫做上下文）的下一词出现概率的乘积。神经语言模型就是用于估计每一个词出现的条件概率的。

2. 循环神经网络

循环神经网络对序列数据的处理有先天优势，它的结构使网络可以接受变长输入，当网络输入窗口被移位时，不需要重复计算。其网络结构如图 1 所示。

我们关注某一时刻τ的损失，它就等于此时刻之前所有时间步的损失之和。例如，对于时间τ时刻的损失$\mathrm{L}^{(\tau)}$，它的损失为

$$\mathrm{L}\left(\{\mathrm{x}^{(1)}, \cdots, \mathrm{x}^{(\tau)}\}, \{y^{(1)}, \cdots, y^{(\tau)}\}\right) = \sum_t L^{(t)} = -\sum_t \log p_{model}\left(y^{(t)} \middle| \{\mathrm{x}^{(1)}, \cdots, \mathrm{x}^{(t)}\}\right).$$

这种在时间步上展开进行反向传播的算法，称为基于时间的反向传播（Back-

Propagation Through Time，BPTT）。循环神经网络的训练就是通过 BPTT 算法进行的。

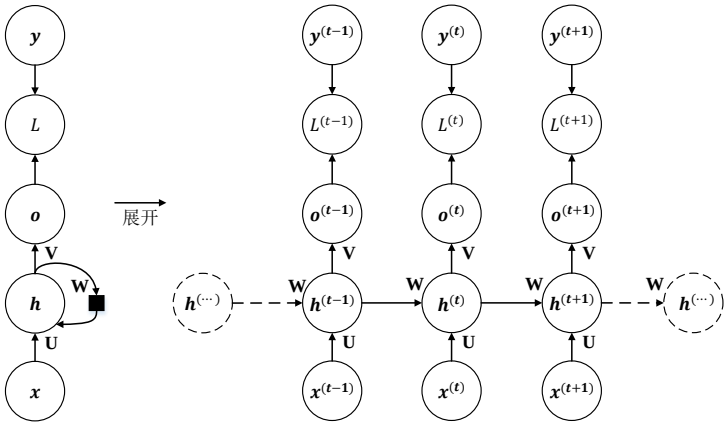

图 1 循环神经网络（RNN）结构

下面以图 1 为例，我们将通过 BPTT 算法来计算 RNN 的参数梯度。对于每一个节点 N，我们需要基于 N 后面的节点的梯度，递归地计算梯度$\nabla_{\mathrm{N}}L$。我们从最后一个节点的损失开始递归：

$$\frac{\partial \mathrm{L}}{\partial \mathrm{L}^{(\mathrm{t})}} = 1.$$

对于时间步 t 输出的梯度$\nabla_{\mathbf{o}^{(\mathrm{t})}}\mathrm{L}$的第 i 个元素为：

$$\left(\nabla_{\mathbf{o}^{(\mathrm{t})}}\mathrm{L}\right)_i = \frac{\partial \mathrm{L}}{\partial \mathrm{o}_i^{(\mathrm{t})}} = \frac{\partial \mathrm{L}}{\partial \mathrm{L}^{(\mathrm{t})}}\frac{\partial \mathrm{L}^{(\mathrm{t})}}{\partial \mathrm{o}_i^{(\mathrm{t})}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}.$$

从序列的最后时间步τ开始反向计算梯度。对于最后时间步τ，$\mathbf{h}^{(\tau)}$只由$\mathbf{o}^{(\mathbf{t})}$得到，则梯度为：

$$\nabla_{\boldsymbol{h}^{(\tau)}}L = \mathbf{V}^{\mathrm{T}}\nabla_{\mathbf{o}^{(\mathrm{t})}}\mathrm{L}.$$

根据这个梯度我们就可以依次计算时间步$\tau - 1$到时间步 1 的隐层节点的梯度。由于$\mathbf{h}^{(\mathbf{t})}(\mathrm{t} < \tau)$是同时有$\mathbf{o}^{(\mathbf{t})}$和$\boldsymbol{h}^{(\boldsymbol{t+1})}$两个后续节点。因此，对于每个时间步 t 的隐层节点的梯度为：

$$\nabla_{\boldsymbol{h}^{(t)}}L = \left(\frac{\partial \boldsymbol{h}^{(\mathbf{t+1})}}{\partial \mathbf{h}^{(\mathbf{t})}}\right)^{\mathrm{T}}\left(\nabla_{\boldsymbol{h}^{(t+1)}}L\right) + \left(\frac{\partial \boldsymbol{o}^{(\mathbf{t})}}{\partial \mathbf{h}^{(\mathbf{t})}}\right)^{T}\left(\nabla_{\mathbf{o}^{(\mathrm{t})}}\mathrm{L}\right)$$

$$= \mathbf{W}^{\mathrm{T}}\left(\nabla_{\boldsymbol{h}^{(t+1)}}L\right)diag\left(1 - \left(\boldsymbol{h}^{(\boldsymbol{t+1})}\right)^{2}\right) + \boldsymbol{V}^{\mathrm{T}}\left(\nabla_{\mathbf{o}^{(\mathrm{t})}}\mathrm{L}\right).$$

在得到了隐层节点的梯度后，我们就可以计算对于参数的梯度：

$$\nabla_{\mathbf{c}}\mathrm{L}=\sum_{\mathrm{t}}\left(\frac{\partial \boldsymbol{o}^{(\mathbf{t})}}{\partial \mathbf{c}}\right)^{T}\nabla_{\mathbf{o}^{(\mathbf{t})}}\mathrm{L}=\sum_{t}\nabla_{\mathbf{o}^{(\mathbf{t})}}\mathrm{L}$$

$$\nabla_{\mathbf{b}}\mathrm{L}=\sum_{\mathrm{t}}\left(\frac{\partial \boldsymbol{h}^{(\mathbf{t})}}{\partial \mathbf{b}^{(\mathbf{t})}}\right)^{T}\nabla_{\mathbf{h}^{(\mathbf{t})}}\mathrm{L}=\sum_{t}diag\left(1-\left(\boldsymbol{h}^{(\boldsymbol{t})}\right)^{2}\right)\nabla_{\mathbf{h}^{(\mathbf{t})}}\mathrm{L}$$

$$\nabla_{\mathbf{V}}\mathrm{L}=\sum_{\mathrm{t}}\sum_{i}\left(\frac{\partial \mathrm{L}}{\partial \mathrm{o}_{i}^{(\mathrm{t})}}\right)\nabla_{\boldsymbol{V}}o_{i}^{(t)}=\sum_{\mathrm{t}}\left(\nabla_{\mathbf{o}^{(\mathbf{t})}}L\right)\boldsymbol{h}^{(t)^{T}}$$

$$\nabla_{\mathbf{W}}\mathrm{L}=\sum_{\mathrm{t}}\sum_{i}\left(\frac{\partial \mathrm{L}}{\partial \mathrm{h}_{i}^{(\mathrm{t})}}\right)\nabla_{\boldsymbol{W}^{(\mathbf{t})}}h_{i}^{(t)}=\sum_{\mathrm{t}}diag\left(1-\left(\boldsymbol{h}^{(\boldsymbol{t})}\right)^{2}\right)\left(\nabla_{\mathbf{h}^{(\mathbf{t})}}L\right)\boldsymbol{h}^{(\boldsymbol{t}-\mathbf{1})^{T}}$$

$$\nabla_{\mathbf{U}}\mathrm{L}=\sum_{\mathrm{t}}\sum_{i}\left(\frac{\partial \mathrm{L}}{\partial \mathrm{h}_{i}^{(\mathrm{t})}}\right)\nabla_{\boldsymbol{U}^{(\boldsymbol{t})}}h_{i}^{(t)}=\sum_{\mathrm{t}}diag\left(1-\left(\boldsymbol{h}^{(\boldsymbol{t})}\right)^{2}\right)\left(\nabla_{\mathbf{h}^{(\mathbf{t})}}L\right)\boldsymbol{x}^{(\boldsymbol{t})^{T}}$$

其中，$\boldsymbol{W}^{(\boldsymbol{t})}$、$\boldsymbol{U}^{(\boldsymbol{t})}$、$\mathbf{b}^{(\mathbf{t})}$表示时刻 t 时，**W**、**U**、**b** 的副本，则$\nabla_{\boldsymbol{W}^{(\boldsymbol{t})}}$表示时刻 t 的时的梯度贡献，对于 **U** 和 **b** 是类似的。

长短期记忆网络（LSTM）是一种具有门结构的特殊循环神经网络。它是为了应对长期依赖的挑战而提出的。LSTM 网络结构如图 2 所示。

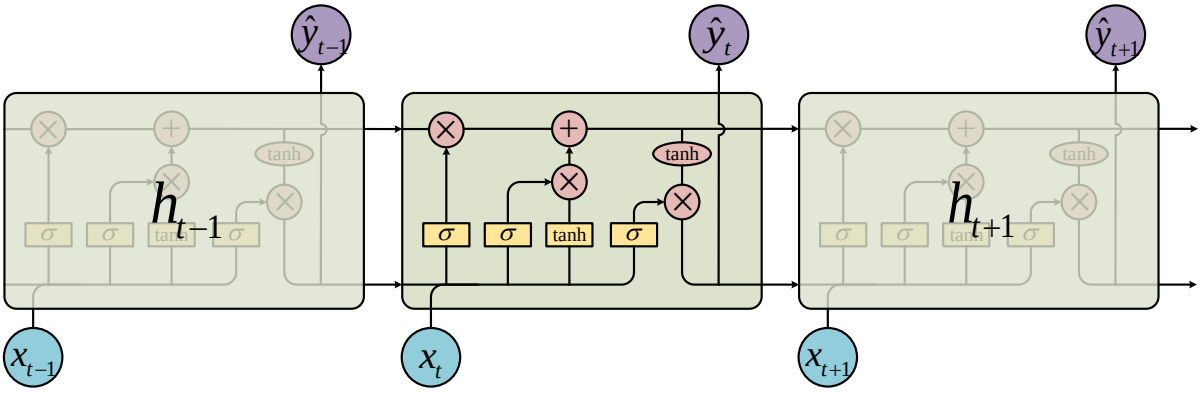

图 2 LSTM 网络结构

LSTM 引入了“门”机制对细胞状态信息进行添加或删除，由此实现长程记忆。“门”机制由一个 Sigmoid 激活函数层和一个向量点乘操作组成，Sigmoid 层的输出控制了信息传递的比例。每个 LSTM 基本单元包含遗忘门、输入门和输出门三个门结构。

1）遗忘门

LSTM 通过遗忘门（forget gate）实现对细胞状态信息遗忘程度的控制，输出当前状态的遗忘权重，取决于$\hat{y}_{t-1}$和x_t。

$$f_t = \sigma(W_f \cdot [\hat{y}_{t-1}, x_t] + b_f)$$

2）输入门

LSTM 通过输入门（input gate）实现对细胞状态输入接收程度的控制，输出当前输入信息的接受权重，取决于$\hat{y}_{t-1}$和x_t。

$$i_t = \sigma(W_i \cdot [\hat{y}_{t-1}, x_t] + b_i)$$

3）输出门

LSTM 通过输出门（output gate）实现对细胞状态输出认可程度的控制，输出当前输出信息的认可权重，取决于$\hat{y}_{t-1}$和x_t。

$$o_t = \sigma(W_o \cdot [\hat{y}_{t-1}, x_t] + b_o)$$

4）状态更新

$$\tilde{C}_t = tanh(W_C \cdot [\hat{y}_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$\hat{y}_t = o_t * tanh(C_t)$$

四、 实验所用工具及数据集

1. 工具

Anaconda、TensorFlow（Tensorflow 安装教程参考：Tensorflow 官网、Tensorflow 中文社区、https://github.com/tensorflow/tensorflow）

2. 数据集

Penn Treebank（PTB）语料库（下载地址：http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz）

五、 实验步骤与方法

1. 下载语料库

数据来源于 Tomas Mikolov 网站上的 PTB 数据集。该数据集已经预先处理过并且包含了全部的 10000 个不同的词语，其中包括语句结束标记符，以及标记稀有词语的特殊符号 (<unk>) 。

2. 加载及准备数据

```
#从文件读入的语料库数据上迭代生成训练、验证、训练数据集
#其中 raw_data 是需要将词转换成词序号，方便后续处理
```

```
def ptb_producer(raw_data, batch_size, num_steps, name=None):
    with tf.name_scope(name, "PTBProducer", [raw_data, batch_size, num_steps]):
        raw_data = tf.convert_to_tensor(raw_data, name="raw_data", dtype=tf.int32)
        #将数据进行切分，去掉多余数据
        data_len = tf.size(raw_data)
        batch_len = data_len // batch_size
        data = tf.reshape(raw_data[0 : batch_size * batch_len],
                                    [batch_size, batch_len])

        epoch_size = (batch_len - 1) // num_steps
        assertion = tf.assert_positive(
                epoch_size,
                message="epoch_size == 0, decrease batch_size or num_steps")
        with tf.control_dependencies([assertion]):
            epoch_size = tf.identity(epoch_size, name="epoch_size")
        #从数据中获得批数据集
        i = tf.train.range_input_producer(epoch_size, shuffle=False).dequeue()
        x = tf.strided_slice(data, [0, i * num_steps],
                                        [batch_size, (i + 1) * num_steps])
        x.set_shape([batch_size, num_steps])
        y = tf.strided_slice(data, [0, i * num_steps + 1],
                                        [batch_size, (i + 1) * num_steps + 1])
        y.set_shape([batch_size, num_steps])
        return x, y
```

3. 构建计算图

（1）构建 LSTM

```
with tf.device("/cpu:0"):
    embedding = tf.get_variable(
            "embedding", [vocab_size, size], dtype=data_type())
    inputs = tf.nn.embedding_lookup(embedding, input_.input_data)
#输入数据接 dropout 层
if is_training and config.keep_prob < 1:
    inputs = tf.nn.dropout(inputs, config.keep_prob)
#为每个 LSTM 单元加上 dropout 层
def make_cell():
    cell = self._get_lstm_cell(config, is_training)
    if is_training and config.keep_prob < 1:
        cell = tf.contrib.rnn.DropoutWrapper(
                cell, output_keep_prob=config.keep_prob)
    return cell
cell = tf.contrib.rnn.MultiRNNCell(
```

```
        [make_cell() for _ in range(config.num_layers)], state_is_tuple=True)
#初始单元状态为零状态
self._initial_state = cell.zero_state(config.batch_size, data_type())
state = self._initial_state
#构建 LSTM 网络
outputs = []
with tf.variable_scope("RNN"):
   for time_step in range(self.num_steps):
       #设置参数共享（重用）
       if time_step > 0: tf.get_variable_scope().reuse_variables()
       (cell_output, state) = cell(inputs[:, time_step, :], state)
       outputs.append(cell_output)
output = tf.reshape(tf.concat(outputs, 1), [-1, config.hidden_size])
#softmax 分类层
softmax_w = tf.get_variable(
       "softmax_w", [size, vocab_size], dtype=data_type())
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=data_type())
logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)
 # Reshape logits to be a 3-D tensor for sequence loss
logits = tf.reshape(logits, [self.batch_size, self.num_steps, vocab_size])
```

（2）训练与评估：

```
# 用在批上平均的序列损失求得损失
loss = tf.contrib.seq2seq.sequence_loss(
       logits,
       input_.targets,
       tf.ones([self.batch_size, self.num_steps], dtype=data_type()),
       average_across_timesteps=False,
       average_across_batch=True)
# 每一时间步损失求和
self._cost = tf.reduce_sum(loss)
#求得梯度
self._lr = tf.Variable(0.0, trainable=False)
       tvars = tf.trainable_variables()
       #使用全局梯度的范数进行梯度截断
       grads, _ = tf.clip_by_global_norm(tf.gradients(self._cost, tvars),
                                                 config.max_grad_norm)
       optimizer = tf.train.GradientDescentOptimizer(self._lr)
       self._train_op = optimizer.apply_gradients(
             zip(grads, tvars),
             global_step=tf.train.get_or_create_global_step())
       #更新学习率
       self._new_lr = tf.placeholder(
```

```
        tf.float32, shape=[], name="new_learning_rate")
    self._lr_update = tf.assign(self._lr, self._new_lr)
```

3. 创建会话，进行模型的训练与评估

```
#在批数据集上运行模型
def run_epoch(session, model, eval_op=None, verbose=False):
    start_time = time.time()
    costs = 0.0
    iters = 0
    state = session.run(model.initial_state)

    fetches = {
        "cost": model.cost,
        "final_state": model.final_state,
    }
    if eval_op is not None:
        fetches["eval_op"] = eval_op

    for step in range(model.input.epoch_size):
        feed_dict = {}
        for i, (c, h) in enumerate(model.initial_state):
            feed_dict[c] = state[i].c
            feed_dict[h] = state[i].h

        vals = session.run(fetches, feed_dict)
        cost = vals["cost"]
        state = vals["final_state"]

        costs += cost
        iters += model.input.num_steps

        if verbose and step % (model.input.epoch_size // 10) == 10:
            print("%.3f perplexity: %.3f speed: %.0f wps" %
                  (step * 1.0 / model.input.epoch_size, np.exp(costs / iters),
                   iters * model.input.batch_size * max(1, FLAGS.num_gpus) /
                   (time.time() - start_time)))

    return np.exp(costs / iters)、

with tf.Graph().as_default():
    sv = tf.train.Supervisor(logdir=FLAGS.save_path)
    config_proto = tf.ConfigProto(allow_soft_placement=soft_placement)
    with sv.managed_session(config=config_proto) as session:
```

```
for i in range(config.max_max_epoch):
    lr_decay = config.lr_decay ** max(i + 1 - config.max_epoch, 0.0)
    m.assign_lr(session, config.learning_rate * lr_decay)

    print("Epoch: %d Learning rate: %.3f" % (i + 1, session.run(m.lr)))
    train_perplexity = run_epoch(session, m, eval_op=m.train_op,
                                 verbose=True)
    print("Epoch: %d Train Perplexity: %.3f" % (i + 1, train_perplexity))
    valid_perplexity = run_epoch(session, mvalid)
    print("Epoch: %d Valid Perplexity: %.3f" % (i + 1, valid_perplexity))

test_perplexity = run_epoch(session, mtest)
print("Test Perplexity: %.3f" % test_perplexity)

if FLAGS.save_path:
    print("Saving model to %s." % FLAGS.save_path)
    sv.saver.save(session, FLAGS.save_path, global_step=sv.global_step)
```