

```

Nearest to no: any, thaler, boiled, gyroscopic, pontificia, grist, occupies,
michelob,
Nearest to UNK: cebus, agouti, cegep, dasyprocta, mitral, reginae, callithrix,
microcebus,
Nearest to used: referred, known, written, found, dasyprocta, able, shown, mitral,
Nearest to up: out, them, him, passes, eat, pianos, gaku, off,

```

下面定义一个用来可视化 Word2Vec 效果的函数。这里 low_dim_embs 是降维到 2 维的单词的空间向量，我们将在图表中展示每个单词的位置。我们使用 plt.scatter（一般将 matplotlib.pyplot 命名为 plt）显示散点图（单词的位置），并用 plt.annotate 展示单词本身。同时，使用 plt.savefig 保存图片到本地文件。

```

def plot_with_labels(low_dim_embs, labels, filename='tsne.png'):
    assert low_dim_embs.shape[0] >= len(labels), "More labels than embeddings"
    plt.figure(figsize=(18, 18))
    for i, label in enumerate(labels):
        x, y = low_dim_embs[i,:]
        plt.scatter(x, y)
        plt.annotate(label,
                     xy=(x, y),
                     xytext=(5, 2),
                     textcoords='offset points',
                     ha='right',
                     va='bottom')

    plt.savefig(filename)

```

我们使用 sklearn.manifold.TSNE 实现降维，这里直接将原始的 128 维的嵌入向量降到 2 维，再用前面的 plot_with_labels 函数进行展示。这里只展示词频最高的 100 个单词的可视化结果。

```

from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)

```

```
plot_only = 100

low_dim_embs = tsne.fit_transform(final_embeddings[:plot_only,:])

labels = [reverse_dictionary[i] for i in range(plot_only)]

plot_with_labels(low_dim_embs, labels)
```

图 7-3 所示即为可视化效果，可以看到其中距离相近的单词在语义上具有很高的相似性。例如，左上角为单个字母的聚集地；而冠词 the、an、a 和 another 则聚集在左边中部，稍微靠右一点则有 him、himself、its、itself 和 them 聚集；左下方有 will、could、would、then。这里我们只展示了部分截图，感兴趣的读者可以在程序画出来的大图中进行观察。对 Word2Vec 性能的评价，除了可视化观察，常用的方式还有 Analogical Reasoning，即直接预测语义、语境上的关系，例如让模型回答 “king is queen as father is to __” 这类问题。Analogical Reasoning 可以比较好地评测 Word2Vec 模型的准确性。在训练 Word2Vec 模型时，为了获得比较好的结果，我们可以使用大规模的语料库，同时需要对参数进行调试，选取最适合的值。

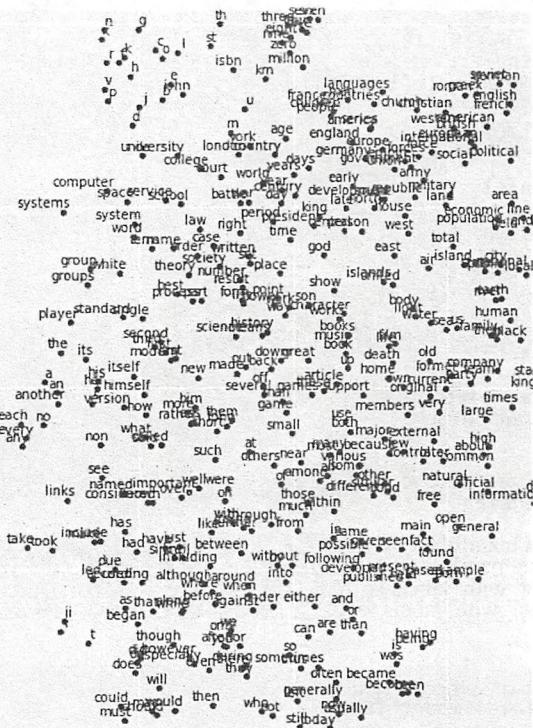


图 7-3 TSNE 降维后的 Word2Vec 的嵌入向量可视化图

7.2 TensorFlow 实现基于 LSTM 的语言模型

循环神经网络出现于 20 世纪 80 年代，在其发展早期，应用不是特别丰富。最近几年由于神经网络结构的进步和 GPU 上深度学习训练效率的突破，RNN 变得越来越流行。RNN 对时间序列数据非常有效，其每个神经元可通过内部组件保存之前输入的信息。

人每次思考时不会重头开始，而是保留之前思考的一些结果为现在的决策提供支持。例如我们对话时，我们会根据上下文的信息理解一句话的含义，而不是对每一句话重头进行分析。传统的神经网络不能实现这个功能，这可能是其一大缺陷。例如卷积神经网络虽然可以对图像进行分类，但是可能无法对视频中每一帧图像发生的事情进行关联分析，我们无法利用前一帧图像的信息，而循环神经网络则可以解决这个问题。RNN 的结构如图 7-4 所示，其最大特点是神经元的某些输出可作为其输入再次传输到神经元中，因此可以利用之前的信息。

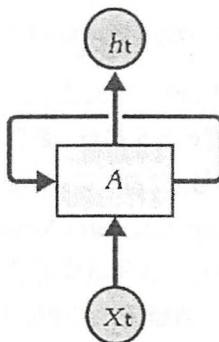


图 7-4 循环神经网络示例

如图 7-4 所示， x_t 是 RNN 的输入， A 是 RNN 的一个节点，而 h_t 是输出。我们对这个 RNN 输入数据 x_t ，然后通过网络计算并得到输出结果 h_t ，再将某些信息（state，状态）传到网络的输入。我们将输出 h_t 与 label 进行比较可以得到误差，有了这个误差之后，就能使用梯度下降（Gradient Descent）和 Back-Propagation Through Time（BPTT）方法对网络进行训练，BPTT 与训练前馈神经网络的传统 BP 方法类似，也是使用反向传播求解梯度并更新网络参数权重。另外，还有一种方法叫 Real-Time Recurrent Learning（RTRL），它可以正向求解梯度，不过其计算复杂度比较高。此外，还有介于 BPTT 和 RTRL 这两种方法之间的混合方法，可用来缓解因为时间序列间隔过长带来的梯度弥散的问题。

如果我们将 RNN 中的循环展开成一个个串联的结构，如图 7-5 所示，就可以更好地理解循环神经网络的结构了。RNN 展开后，类似于有一系列输入 x 和一系列输出 h 的串联的普通神经网络，上一层的神经网络会传递信息给下一层。这种串联的结构天然就非常适合时间序列数据的处理和分析。需要注意的是，展开后的每一个层级的神经网络，其参数都是相同的，我们并不需要训练成百上千层神经网络的参数，只需要训练一层 RNN 的参数，这就是它结构巧妙的地方，这里共享参数的思想和卷积网络中权值共享的方式也很类似。

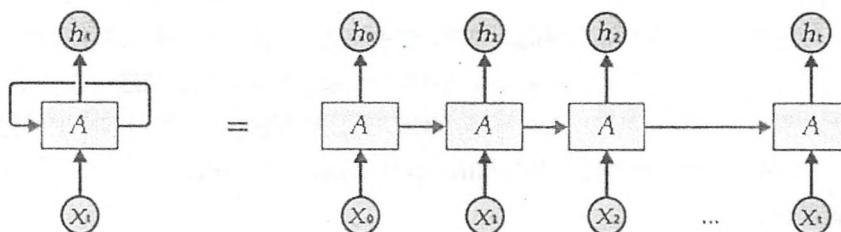


图 7-5 循环神经网络展开示意图

RNN 虽然被设计成可以处理整个时间序列信息，但是其记忆最深的还是最后输入的一些信号。而更早之前的信号的强度则越来越低，最后只能起到一点辅助的作用，即决定 RNN 输出的还是最后输入的一些信号。这样的缺陷导致 RNN 在早期的作用并不明显，慢慢淡出了大家的视野。而后随着 Long Short Term Memory (LSTM)⁵⁷ 的发现，循环神经网络重新回到了大家的视野，并逐渐在众多领域取得了很大的成功和突破，包括语音识别、文本分类、语言模型、自动对话、机器翻译、图像标注等领域。

对于某些简单的问题，可能只需要最后输入的少量时序信息即可解决。但对某些复杂问题，可能需要更早的一些信息，甚至是时间序列开头的信息，但间隔太远的输入信息，RNN 是难以记忆的，因此长程依赖 (Long-term Dependencies) 是传统 RNN 的致命伤。LSTM 由 Schmidhuber 教授于 1997 年提出，它天生就是为了解决长程依赖而设计的，不需要特别复杂地调试超参数，默认就可以记住长期的信息。LSTM 的内部结构相比 RNN 更复杂，如图 7-6 所示，其中包含了 4 层神经网络，其中小圆圈是 point-wise 的操作，比如向量加法、点乘等，而小矩形则代表一层可学习参数的神经网络。LSTM 单元上面的那条直线代表了 LSTM 的状态 state，它会贯穿所有串联在一起的 LSTM 单元，从第一个 LSTM 单元一直流向最后一个 LSTM 单元，其中只有少量的线性干预和改变。状态 state 在这条隧道中传递时，LSTM 单元可以对其添加或删减信息，这些对信息流的修改操作由

LSTM 中的 Gates 控制。这些 Gates 中包含了一个 Sigmoid 层和一个向量点乘的操作，这个 Sigmoid 层的输出是 0 到 1 之间的值，它直接控制了信息传递的比例。如果为 0 代表不允许信息传递，为 1 则代表让信息全部通过。每个 LSTM 单元中包含了 3 个这样的 Gates，用来维护和控制单元的状态信息。凭借对状态信息的储存和修改，LSTM 单元就可以实现长程记忆。

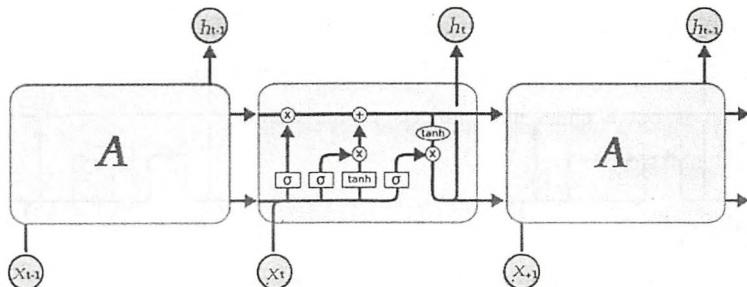


图 7-6 LSTM 结构示意图

在 RNN 的各种变种中，除了 LSTM，另一个非常流行的网络结构是 Gated Recurrent Unit (GRU)。GRU 的结构如图 7-7 所示，相比 LSTM，其结构更加简单，比 LSTM 减少了一个 Gate，因此计算效率更高（每个单元每次计算时可节约几个矩阵运算操作），同时占用的内存也相对较少。在实际使用中，LSTM 和 GRU 的差异不大，一般最后得到的准确率指标等都近似，但是相对来说，GRU 达到收敛状态时所需要的迭代数更少，也可以说是训练速度更快。

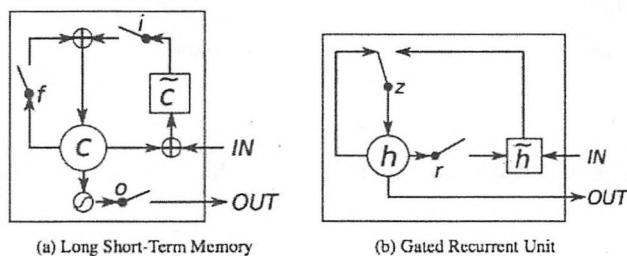


Illustration of (a) LSTM and (b) gated recurrent units. (a) i , f and o are the input, forget and output gates, respectively. c and \tilde{c} denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and \tilde{h} are the activation and the candidate activation.

图 7-7 LSTM 和 GRU 的结构

循环神经网络的应用非常广，不过用的最多的地方还是自然语言处理。用 RNN 训练

出的语言模型 (Language Modeling)，其效果令人惊叹。我们可以输入大量莎士比亚的剧本文字等信息给 RNN，训练得到的语言模型可以模仿莎士比亚的文字，自动生成类似的诗歌、剧本。下面的英文为语言模型生成的莎翁的诗歌，可以说是非常逼真，几乎可以以假乱真。

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

语言模型是 NLP 中非常重要的一个部分，同时也是语音识别、机器翻译和由图片生成标题等任务的基础和关键。语言模型是一个可以预测语句的概率模型。给定上文的语境，即历史出现的单词，语言模型可以预测下一个单词出现的概率。Penn Tree Bank (PTB)

是在语言模型训练中经常使用的一个数据集，它的质量比较高，可以用来评测语言模型的准确率，同时数据集不大，训练也比较快。下面我们就使用 LSTM 来实现一个语言模型，其网络结构来自论文 *Recurrent Neural Network Regularization*。

首先，我们下载 PTB 数据集并解压，确保解压后的文件路径和接下来 Python 的执行路径一致。这个数据集中已经做了一些预处理，它包含 1 万个不同的单词，有句尾的标记，同时将罕见的词汇统一处理为特殊字符。本节代码主要来自 TensorFlow 的开源实现⁵⁸。

```
wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
tar xvf simple-examples.tgz
```

我们下载 TensorFlow Models 库，并进入目录 models/tutorials/rnn/ptb。然后载入常用的库，和 TensorFlow Models 中的 PTB reader，借助它读取数据内容。读取数据内容的操作比较烦琐，主要是将单词转为唯一的数字编码，以便神经网络处理。这部分实现的细节我们不做讲解，感兴趣的读者可以阅读其源码。

```
git clone https://github.com/tensorflow/models.git
cd models/tutorials/rnn/ptb
import time
import numpy as np
import tensorflow as tf
import reader
```

下面定义语言模型处理输入数据的 class，PTBInput。其中只有一个初始化方法 __init__(), 我们读取参数 config 中的 batch_size、num_steps 到本地变量，这里 num_steps 是 LSTM 的展开步数(unrolled steps of LSTM)。然后计算每个 epoch 的 size，即每个 epoch 内需要多少轮训练的迭代，可以通过将数据长度整除 batch_size 和 num_steps 得到。我们使用 reader.ptb_producer 获取特征数据 input_data，以及 label 数据 targets，这里的 input_data 和 targets 都已经是定义好的 tensor 了，每次执行都会获取一个 batch 的数据。

```
class PTBInput(object):

    def __init__(self, config, data, name=None):
        self.batch_size = batch_size = config.batch_size
        self.num_steps = num_steps = config.num_steps
        self.epoch_size = ((len(data) // batch_size) - 1) // num_steps
```

```
self.input_data, self.targets = reader.ptb_producer(
    data, batch_size, num_steps, name=name)
```

接着定义语言模型的 class，PTBModel。首先依然是初始化函数 `_init_()`，其中包含三个参数，训练标记 `is_training`、配置参数 `config`，以及 `PTBInput` 类的实例 `input_`。我们读取 `input_` 中的 `batch_size` 和 `num_steps`，然后读取 `config` 中的 `hidden_size`、`vocab_size` 到本地变量。这里 `hidden_size` 是 LSTM 的节点数，`vocab_size` 是词汇表的大小。

```
class PTBModel(object):

    def __init__(self, is_training, config, input_):
        self._input = input_

        batch_size = input_.batch_size
        num_steps = input_.num_steps
        size = config.hidden_size
        vocab_size = config.vocab_size
```

接下来使用 `tf.contrib.rnn.BasicLSTMCell` 设置我们默认的 LSTM 单元，其中隐含节点数为前面提取的 `hidden_size`，`forget_bias`（即 forget gate 的 bias）为 0，`state_is_tuple` 也为 `True`，这代表接受和返回的 `state` 将是 2-tuple 的形式。同时，如果在训练状态且 `Dropout` 的 `keep_prob` 小于 1，则在前面的 `lstm_cell` 之后接一个 `Dropout` 层，这里的做法是调用 `tf.contrib.rnn.DropoutWrapper` 函数。最后使用 RNN 堆叠函数 `tf.contrib.rnn.MultiRNNCell` 将前面构造的 `lstm_cell` 多层堆叠得到 `cell`，堆叠次数为 `config` 中的 `num_layers`，这里同样将 `state_is_tuple` 设为 `True`，并用 `cell.zero_state` 设置 LSTM 单元的初始化状态为 0。这里需要注意，LSTM 单元可以读入一个单词并结合之前储存的状态 `state` 计算下一个单词出现的概率分布，并且每次读取一个单词后它的状态 `state` 会被更新。

```
def lstm_cell():
    return tf.contrib.rnn.BasicLSTMCell(
        size, forget_bias=0.0, state_is_tuple=True)
attn_cell = lstm_cell
if is_training and config.keep_prob < 1:
    def attn_cell():
        return tf.contrib.rnn.DropoutWrapper(
```

```

lstm_cell(), output_keep_prob=config.keep_prob)
cell = tf.contrib.rnn.MultiRNNCell(
    [attn_cell() for _ in range(config.num_layers)],
    state_is_tuple=True)

self._initial_state = cell.zero_state(batch_size, tf.float32)

```

我们创建网络的词嵌入 embedding 部分，embedding 即为将 one-hot 的编码格式的单词转化为向量表达形式，在 7.1 节 Word2Vec 中已经讲到了。因为这部分在 GPU 中还没有很好的实现，所以我们依然使用 with tf.device("/cpu:0") 将计算限定在 CPU 中进行。然后我们初始化 embedding 矩阵，其行数设为词汇表数 vocab_size，列数（即每个单词的向量表达的维数）设为 hidden_size，和 LSTM 单元中的隐含节点数一致。在训练过程中，embedding 的参数可以被优化和更新。接下来使用 tf.nn.embedding_lookup 查询单词对应的向量表达获得 inputs。同时，如果为训练状态则再添加上一层 Dropout。

```

with tf.device("/cpu:0"):
    embedding = tf.get_variable(
        "embedding", [vocab_size, size], dtype=tf.float32)
    inputs = tf.nn.embedding_lookup(embedding, input_.input_data)

    if is_training and config.keep_prob < 1:
        inputs = tf.nn.dropout(inputs, config.keep_prob)

```

接下来定义输出 outputs，我们先使用 tf.variable_scope 将接下来的操作的名称设为 RNN。一般为了控制训练过程，我们会限制梯度在反向传播时可以展开的步数为一个固定的值，而这个步数也就是 num_steps。这里我们设置一个循环，循环长度为 num_steps，来控制梯度的传播。并且从第 2 次循环开始，我们使用 tf.get_variable_scope.reuse_variables 设置复用变量。在每次循环内，我们传入 inputs 和 state 到堆叠的 LSTM 单元（即 cell）中。这里注意 inputs 有 3 个维度，第 1 个维度代表是 batch 中的第几个样本，第 2 个维度代表是样本中的第几个单词，第 3 个维度是单词的向量表达的维度，而 inputs[:, time_step, :] 代表所有样本的第 time_step 个单词。这里我们得到输出 cell_output 和更新后的 state。最后我们将结果 cell_output 添加到输出列表 outputs。

```

outputs = []
state = self._initial_state

```

```

with tf.variable_scope("RNN"):
    for time_step in range(num_steps):
        if time_step > 0: tf.get_variable_scope().reuse_variables()
        (cell_output, state) = cell(inputs[:, time_step, :], state)
        outputs.append(cell_output)

```

我们将 output 的内容用 tf.concat 串接到一起，并使用 tf.reshape 将其转为一个很长的一维向量。接下来是 Softmax 层，先定义权重 softmax_w 和偏置 softmax_b，然后使用 tf.matmul 将输出 output 乘上权重并加上偏置得到 logits，即网络最后的输出。然后定义损失 loss，这里直接使用 tf.contrib.legacy_seq2seq.sequence_loss_by_example 计算输出 logits 和 targets 的偏差，这里的 sequence_loss 即 target words 的 average negative log probability，其定义为 $loss = -\frac{1}{N} \sum_{i=1}^N \ln p_{target_i}$ 。然后使用 tf.reduce_sum 汇总 batch 的误差，再计算平均到每个样本的误差 cost。并且我们保留最终的状态为 final_state。此时，如果不是训练状态，则直接返回。

```

output = tf.reshape(tf.concat(outputs, 1), [-1, size])
softmax_w = tf.get_variable(
    "softmax_w", [size, vocab_size], dtype=tf.float32)
softmax_b = tf.get_variable("softmax_b", [vocab_size], dtype=tf.float32)
logits = tf.matmul(output, softmax_w) + softmax_b
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example(
    [logits],
    [tf.reshape(input_.targets, [-1])],
    [tf.ones([batch_size * num_steps], dtype=tf.float32)])
self._cost = cost = tf.reduce_sum(loss) / batch_size
self._final_state = state

if not is_training:
    return

```

下面定义学习速率的变量 lr，并将其设为不可训练。再使用 tf.trainable_variables 获取全部可训练的参数 tvars。这里针对前面得到的 cost，计算 tvars 的梯度，并用 tf.clip_by_global_norm 设置梯度的最大范数 max_grad_norm。这即是 Gradient Clipping 的方法，控制梯度的最大范数，某种程度上起到正则化的效果。Gradient Clipping 可以防止

Gradient Explosion 梯度爆炸的问题，如果对梯度不加限制，则可能会因为迭代中梯度过大导致训练难以收敛。然后定义优化器为 GradientDescent 优化器。再创建训练操作 _train_op，用 optimizer.apply_gradients 将前面 clip 过的梯度应用到所有可训练的参数 tvars 上，然后使用 tf.contrib.framework.get_or_create_global_step 生成全局统一的训练步数。

```
self._lr = tf.Variable(0.0, trainable=False)
tvars = tf.trainable_variables()
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars),
                                  config.max_grad_norm)
optimizer = tf.train.GradientDescentOptimizer(self._lr)
self._train_op = optimizer.apply_gradients(zip(grads, tvars),
                                           global_step=tf.contrib.framework.get_or_create_global_step())
```

这里设置一个名为_new_lr (new learning rate) 的 placeholder 用以控制学习速率，同时定义操作_lr_update，它使用 tf.assign 将_new_lr 的值赋给当前的学习速率_lr。再定义一个 assign_lr 的函数，用来在外部控制模型的学习速率，方式是将学习速率值传入_new_lr 这个 placeholder，并执行_update_lr 操作完成对学习速率的修改。

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

def assign_lr(self, session, lr_value):
    session.run(self._lr_update, feed_dict={self._new_lr: lr_value})
```

至此，模型定义的部分就完成了。我们再定义这个 PTBModel class 的一些 property，Python 中的@property 装饰器可以将返回变量设为只读，防止修改变量引发的问题。这里定义 input、initial_state、cost、final_state、lr、train_op 为 property，方便外部访问。

```
@property
def input(self):
    return self._input

@property
def initial_state(self):
```

```
    return self._initial_state

    @property
    def cost(self):
        return self._cost

    @property
    def final_state(self):
        return self._final_state

    @property
    def lr(self):
        return self._lr

    @property
    def train_op(self):
        return self._train_op
```

接下来定义几种不同大小的模型的参数。首先是小模型的设置，我们先解释各个参数的含义，这里的 init_scale 是网络中权重值的初始 scale；learning_rate 是学习速率的初始值；max_grad_norm 即前面提到的梯度的最大范数；num_layers 是 LSTM 可以堆叠的层数；num_steps 是 LSTM 梯度反向传播的展开步数；hidden_size 是 LSTM 内的隐含节点数；max_epoch 是初始学习速率可训练的 epoch 数，在此之后需要调整学习速率；max_max_epoch 是总共可训练的 epoch 数；keep_prob 是 dropout 层的保留节点的比例；lr_decay 是学习速率的衰减速度；batch_size 是每个 batch 中样本的数量。具体每个参数的值，在不同配置中对比才有意义，我们会在接下来的几个配置中讨论具体数值。

```
class SmallConfig(object):
    init_scale = 0.1
    learning_rate = 1.0
    max_grad_norm = 5
    num_layers = 2
    num_steps = 20
    hidden_size = 200
```

```
max_epoch = 4
max_max_epoch = 13
keep_prob = 1.0
lr_decay = 0.5
batch_size = 20
vocab_size = 10000
```

这里可以看到，在 MediumConfig 中型模型中，我们减小了 init_scale，即希望权重初值不要过大，小一些有利于温和的训练；学习速率和最大梯度范数不变，LSTM 层数也不变；这里将梯度反向传播的展开步数 num_steps 从 20 增大到 35；hidden_size 和 max_max_epoch 也相应地增大约 3 倍；同时，这里开始设置 dropout 的 keep_prob 到 0.5，而之前设为 1 即没有 dropout；因为学习的迭代次数增大，因此将学习速率的衰减速率 lr_decay 也减小了；batch_size 和词汇表 vocab_size 的大小都保持不变。

```
class MediumConfig(object):
    init_scale = 0.05
    learning_rate = 1.0
    max_grad_norm = 5
    num_layers = 2
    num_steps = 35
    hidden_size = 650
    max_epoch = 6
    max_max_epoch = 39
    keep_prob = 0.5
    lr_decay = 0.8
    batch_size = 20
    vocab_size = 10000
```

LargeConfig 大型模型进一步缩小了 init_scale；并大大放宽了最大梯度范数 max_grad_norm 到 10；同时将 hidden_size 提升到了 1500，并且 max_epoch、max_max_epoch 也相应地增大了；而 keep_drop 则因为模型复杂度的上升继续下降。学习速率的衰减速率 lr_decay 也进一步减小。

```
class LargeConfig(object):
    init_scale = 0.04
```

```
learning_rate = 1.0
max_grad_norm = 10
num_layers = 2
num_steps = 35
hidden_size = 1500
max_epoch = 14
max_max_epoch = 55
keep_prob = 0.35
lr_decay = 1 / 1.15
batch_size = 20
vocab_size = 10000
```

这里的 TestConfig 只是为测试用，参数都尽量使用最小值，只是为了测试可以完整运行模型。

```
class TestConfig(object):
    init_scale = 0.1
    learning_rate = 1.0
    max_grad_norm = 1
    num_layers = 1
    num_steps = 2
    hidden_size = 2
    max_epoch = 1
    max_max_epoch = 1
    keep_prob = 1.0
    lr_decay = 0.5
    batch_size = 20
    vocab_size = 10000
```

下面定义训练一个 epoch 数据的函数 run_epoch。我们记录当前时间，初始化损失 costs 和迭代数 iters，并执行 model.initial_state 来初始化状态并获得初始状态。接着创建输出结果的字典表 fetches，其中包括 cost 和 final_state，如果有评测操作 eval_op，也一并加入 fetches。接着我们进入训练循环中，次数即为 epoch_size。在每次循环中，我们生成训练用的 feed_dict，将全部 LSTM 单元的 state 加入 feed_dict 中，然后传入 feed_dict 并执行

fetches 对网络进行一次训练，并拿到 cost 和 state。这里我们累加 cost 到 costs，并累加 num_steps 到 iters。我们每完成约 10% 的 epoch，就进行一次结果的展示，依次展示当前 epoch 的进度、perplexity（即平均 cost 的自然常数指数，是语言模型中用来比较模型性能的重要指标，越低代表模型输出的概率分布在预测样本上越好）和训练速度（单词数每秒）。最后返回 perplexity 作为函数结果。

```
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 / (time.time() - start_time)))

return np.exp(costs / iters)

```

我们使用 reader.ptb_raw_data 直接读取解压后的数据，得到训练数据、验证数据和测试数据。这里定义训练模型的配置为 SmallConfig，读者也可自行测试其他大小的模型。需要注意的是测试配置 eval_config 需和训练配置一致，这里将测试配置的 batch_size 和 num_steps 修改为 1。

```

raw_data = reader.ptb_raw_data('simple-examples/data/')
train_data, valid_data, test_data, _ = raw_data

config = SmallConfig()
eval_config = SmallConfig()
eval_config.batch_size = 1
eval_config.num_steps = 1

```

我们创建默认的 Graph，并使用 tf.random_uniform_initializer 设置参数的初始化器，令参数范围在[-init_scale, init_scale]之间。然后使用 PTBInput 和 PTBModel 创建一个用来训练的模型 m，以及用来验证的模型 mvalid 和测试的模型 mtest，其中训练和验证模型直接使用前面的 config，测试模型则使用前面的测试配置 eval_config。

```

with tf.Graph().as_default():
    initializer = tf.random_uniform_initializer(-config.init_scale,
                                                config.init_scale)

    with tf.name_scope("Train"):
        train_input = PTBInput(config=config, data=train_data, name="TrainInput")
        with tf.variable_scope("Model", reuse=None, initializer=initializer):
            m = PTBModel(is_training=True, config=config, input_=train_input)

    with tf.name_scope("Valid"):
        valid_input = PTBInput(config=config, data=valid_data, name="ValidInput")

```

```

with tf.variable_scope("Model", reuse=True, initializer=initializer):
    mvalid = PTBModel(is_training=False, config=config, input_=valid_input)

with tf.name_scope("Test"):
    test_input = PTBInput(config=eval_config, data=test_data,
                          name="TestInput")
    with tf.variable_scope("Model", reuse=True, initializer=initializer):
        mtest = PTBModel(is_training=False, config=eval_config,
                          input_=test_input)

```

我们使用 `tf.train.Supervisor()` 创建训练的管理器 `sv`, 并使用 `sv.managed_session` 创建默认 `session`, 再执行训练多个 epoch 数据的循环。在每个 epoch 循环内, 我们先计算累计的学习速率衰减值, 这里只需计算超过 `max_epoch` 的轮数, 再求 `lr_decay` 的超出轮数次幂即可。然后将初始学习速率乘上累计的衰减, 并更新学习速率。然后在循环内执行一个 epoch 的训练和验证, 并输出当前的学习速率、训练和验证集上的 perplexity。在完成全部训练后, 计算并输出模型在测试集上的 perplexity。

```

sv = tf.train.Supervisor()
with sv.managed_session() 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)

```

我们来看 SmallConfig 小型模型的最后结果, 我们在 i7 6900K 和 GTX 1080 上的训

练速度可达 21000 单词每秒。同时在最后一个 epoch 中, 训练集上可达 36.9 的 perplexity, 而验证集和测试集上分别可达 122.3 和 116.7 的 perplexity。

```
Epoch: 13 Learnign rate: 0.004  
0.004 perplexity: 56.003 speed: 13005 wps  
0.104 perplexity: 41.096 speed: 21836 wps  
0.204 perplexity: 45.000 speed: 21891 wps  
0.304 perplexity: 43.224 speed: 21738 wps  
0.404 perplexity: 42.508 speed: 21529 wps  
0.504 perplexity: 41.803 speed: 21565 wps  
0.604 perplexity: 40.425 speed: 21470 wps  
0.703 perplexity: 39.768 speed: 21418 wps  
0.803 perplexity: 39.088 speed: 21480 wps  
0.903 perplexity: 37.753 speed: 21493 wps  
Epoch: 13 Train Perplexity: 36.949  
Epoch: 13 Valid Perplexity: 122.300  
Test Perplexity: 116.763
```

读者可以自行测试中型模型和大型模型, 在原论文中提到在中型模型上可以达到(训练集: 48.45, 验证集: 86.16, 测试集: 82.07)的效果, 在大型模型上, 可以达到(训练集: 37.87, 验证集: 82.62, 测试集: 78.29)的效果。本节我们实现了一个基于 LSTM 的语言模型, 读者应该了解到 LSTM 在处理文本等时序数据中的作用了。LSTM 可以存储状态, 并依靠状态对当前的输入进行处理分析和预测。RNN 和 LSTM 赋予了神经网络记忆和储存过往信息的能力, 可以模仿人类的一些简单的记忆和推理功能。而目前, 注意力(attention)机制是 RNN 和 NLP 领域研究的热点, 这种机制让机器可以更好地模拟人脑的功能。在图像标题生成任务中, 包含注意力机制的 RNN 可以对某一区域的图像进行分析, 并生成对应的文字描述, 有兴趣的读者可以阅读论文 *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* 了解这部分的相关信息。

7.3 TensorFlow 实现 Bidirectional LSTM Classifier

双向循环神经网络(Bidirectional Recurrent Neural Networks⁵⁹, Bi-RNN)是由 Schuster 和 Paliwal 于 1997 年首次提出的, 和 LSTM 是在同一年被提出的。Bi-RNN 的主要目标是

增加 RNN 可利用的信息。比如普通的 MLP 对数据长度等有限制，而 RNN 虽然可以处理不固定长度的时序数据，但是无法利用某个历史输入的未来信息。Bi-RNN 则正好相反，它可以同时使用时序数据中某个输入的历史及未来数据。其实现原理很简单，将时序方向相反的两个循环神经网络连接到同一个输出，通过这种结构，输出层就可以同时获取历史和未来信息了。

在需要上下文环境的情况下，Bi-RNN 将会非常有用，比如在手写文字识别时，如果有当前要识别的单词的前面和后面一个单词的信息，那么将非常有利于识别。同样，我们在阅读文章时，有时也需要通过下文的语境来推测文中某句话的准确含义。对 Language Modeling 这类问题，可能 Bi-RNN 并不合适，因为我们的目标就是通过前文预测下一个单词，这里不能将下文信息传给模型。对很多分类问题，比如手写文字识别、机器翻译、蛋白结构预测等，使用 Bi-RNN 将会大大提升模型效果。百度在其语音识别中也是通过 Bi-RNN 综合考虑上下文语境，将其模型准确率大大提升。

Bi-RNN 网络结构的核心是把一个普通的单向的 RNN 拆成两个方向，一个是随时序正向的，一个是逆着时序的反向的，如图 7-8 所示。这样当前时间节点的输出就可以同时利用正向、反向两个方向的信息，而不像普通 RNN 需要等到后面时间节点才可以获取未来信息。这两个不同方向的 RNN 之间不会共用 state，即正向 RNN 的输出 state 只会传给正向的 RNN，反向 RNN 的输出只会传给反向的 RNN，它们之间没有直接连接。如图 7-9 所示，每一个时间节点的输入会分别传到正向和反向的 RNN 中，它们根据各自的状态产生输出，这两份输出会一起连接到 Bi-RNN 的输出节点，共同合成最终输出。我们可以看到，Bi-RNN 的网络中虽然两个方向的 RNN 基本没有交集，但是因为它们共同合成了输出，所以它们对当前时间节点输出的贡献（或造成的 loss）就可以在训练中被计算出来，并且它们的参数会根据梯度被优化到合适的值。

Bi-RNN 在训练时和普通单向 RNN 非常类似，因为两个不同方向的 RNN 之间几乎没有交集，因此它们可以分别展开为普通的前馈网络。不过在使用 BPTT (back-propagation through time) 算法训练时，我们无法同时更新状态和输出。同时，正向 state 在 $t=1$ 时未知，且反向 state 在 $t=T$ 时未知，即 state 在各自方向的开始处未知，这里需要人工设置。此外，正向状态的导数在 $t=T$ 时未知，且反向 state 的导数在 $t=1$ 时未知，即 state 的导数在结尾处未知，这里一般需要设为 0 代表此时对参数更新不重要。然后正式开始训练步骤：第一步，我们对输入数据做 forward pass 操作，即 inference 的操作，我们先沿着 $1 \rightarrow T$ 方向计算正向 RNN 的 state，再沿着 $T \rightarrow 1$ 方向计算反向 RNN 的 state，然后获得输出 output；

第二步，我们进行 backward pass 操作，即对目标函数求导的操作，我们先对输出 output 求导，然后沿着 T→1 方向计算正向 RNN 的 state 的导数，再沿着 1→T 方向计算反向 RNN 的 state 的导数；第三步根据求得的梯度值更新模型参数，完成一次训练。

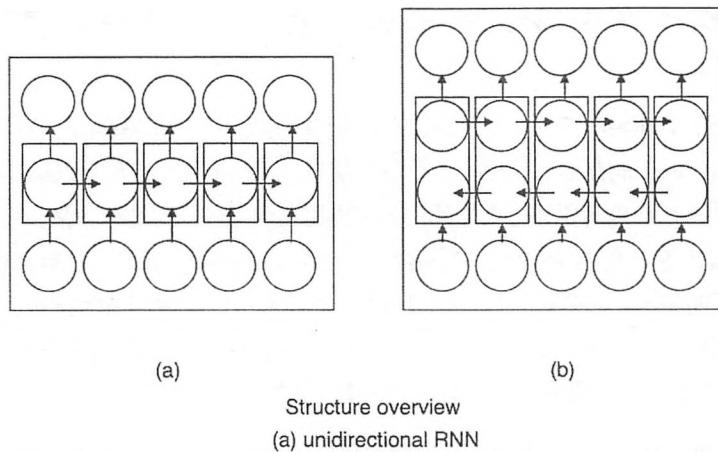


图 7-8 RNN 和 Bi-RNN 结构对比图

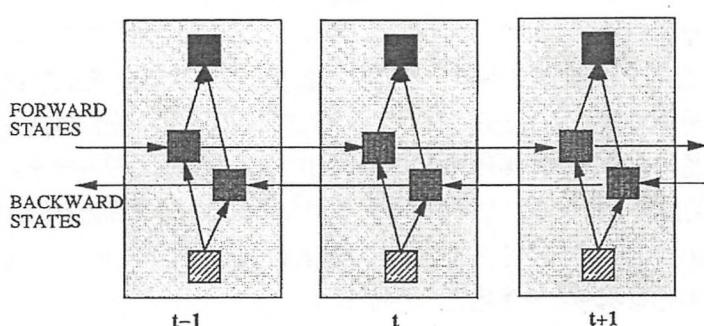


图 7-9 Bi-RNN 结构示意图

Bi-RNN 中的每个 RNN 单元既可以是传统的 RNN，也可以是 LSTM 单元或者 GRU 单元，思路是一致的，而且我们也可以在一层 Bi-RNN 上再叠加一层 Bi-RNN，即上一层 Bi-RNN 的输出再作为下一层 Bi-RNN 的输入，可以进一步抽象提炼特征。如果最后用作分类任务，我们可以将 Bi-RNN 的输出序列连接一个全连接层，或者连接全局平均池化 Global Average Pooling，最后再接 Softmax 层，这部分和使用卷积网络的输出进行分类的做法一样。

下面我们就使用 TensorFlow 实现一个 Bidirectional LSTM Classifier，并在 MNIST 数据集上进行测试。先载入 TensorFlow、NumPy，以及 TensorFlow 自带的 MNIST 数据读取器。与最开始的几章一样，我们直接使用 `input_data.read_data_sets` 下载并读取 MNIST 数据集。本节代码主要来自 TensorFlow-Examples 的开源实现⁶⁰。

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

然后设置训练参数。我们设置学习速率为 0.01（因为优化器将选择 Adam，所以学习速率较低），最大训练样本数为 40 万，`batch_size` 为 128，同时设置每间隔 10 次训练就展示一次训练情况。

```
learning_rate = 0.01
max_samples = 400000
batch_size = 128
display_step = 10
```

因为 MNIST 的图像尺寸为 28×28，因此输入 `n_input` 为 28（图像的宽），同时 `n_steps` 即 LSTM 的展开步数（unrolled steps of LSTM），也设置为 28（图像的高），这样图像的全部信息就都使用上了。和前一节使用 LSTM 处理文本数据时一次读取一个单词类似，这里是一次读取一行像素（28 个像素点），然后下一个时间点再传入下一行像素点。这里 `n_hidden`（LSTM 的隐藏节点数）设为 256，而 `n_classes`（MNIST 数据集的分类数目）则设为 10。

```
n_input = 28
n_steps = 28
n_hidden = 256
n_classes = 10
```

我们创建输入 `x` 和学习目标 `y` 的 `place_holder`。和使用卷积神经网络做分类时类似，这里输入 `x` 中每一个样本可直接使用二维的结构，而不必像 MLP 那样需要转为一维结构。不过这里的样本的二维的含义，和卷积网络中空间的二维不同，我们的样本被理解为一个时间序列，第一个维度是时间点 `n_steps`，第二个维度是每个时间点的数据 `n_input`。同时，我们设创建最后的 Softmax 层的 `weights` 和 `biases`，这里直接使用 `tf.random_normal` 初始化

这些参数。因为是双向 LSTM，有 forward 和 backwrad 两个 LSTM 的 cell，所以 weights 的参数量也翻倍，变为 $2 * n_hidden$ 。

```
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

weights = tf.Variable(tf.random_normal([2*n_hidden, n_classes]))
biases = tf.Variable(tf.random_normal([n_classes]))
```

下面就定义 Bidirectional LSTM 网络的生成函数。我们先对数据进行一些处理，把形状为 ($batch_size, n_steps, n_input$) 的输入变成长度为 n_steps 的列表，而其中元素形状为($batch_size, n_input$)。然后输入进行转置，使用 $tf.transpose(x, [1, 0, 2])$ 将第一个维度 $batch_size$ 和第二个维度 n_steps 进行交换。接着使用 $tf.reshape$ 将输入 x 变形为 $(n_steps*batch_size, n_input)$ 的形状，再使用 $tf.split$ 将 x 拆成长度为 n_steps 的列表，列表中每个 tensor 的尺寸都是($batch_size, n_input$)，这样符合 LSTM 单元的输入格式。下面使用 $tf.contrib.rnn.BasicLSTMCell$ 分别创建 forward 和 backward 的 LSTM 单元，它们的隐藏节点数都设为 n_hidden ，而 $forget_bias$ 都设为 1。然后直接将正向的 $lstm_fw_cell$ 和反向的 $lstm_bw_cell$ 传入 Bi-RNN 接口 $tf.nn.bidirectional_rnn$ 中，生成双向 LSTM，并传入 x 作为输入。最后对双向 LSTM 的输出结果 $outputs$ 做一个矩阵乘法并加上偏置，这里的参数即为前面定义的 $weights$ 和 $biases$ 。

```
def BiRNN(x, weights, biases):

    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, n_input])
    x = tf.split(x, n_steps)

    lstm_fw_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
    lstm_bw_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)

    outputs, _, _ = tf.contrib.rnn.static_bidirectional_rnn(lstm_fw_cell,
                                                          lstm_bw_cell, x, dtype=tf.float32)
    return tf.matmul(outputs[-1], weights) + biases
```

我们使用刚才定义好的函数生成我们的 Bidirectional LSTM 网络，对最后输出的结果

使用 `tf.nn.softmax_cross_entropy_with_logits` 进行 Softmax 处理并计算损失，然后使用 `tf.reduce_mean` 计算平均 cost。我们定义优化器为 Adam，学习速率即为前面定义的 `learning_rate`。再使用 `tf.argmax` 得到模型预测的类别，然后用 `tf.equal` 判断是否预测正确，最后用 `tf.reduce_mean` 求得平均准确率。

```
pred = BiRNN(x, weights, biases)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,
                                                               labels=y))

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(
    cost)

correct_pred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

init = tf.global_variables_initializer()
```

下面开始执行训练和测试操作。第一步是执行初始化参数，然后定义一个训练的循环，保持总训练样本数（迭代次数*batch_size）小于之前设定的值。在每一轮训练迭代中，我们使用 `mnist.train.next_batch` 拿到一个 batch 的数据并使用 `reshape` 改变其形状。接着，将包含输入 `x` 和训练目标 `y` 的 `feed_dict` 传入，执行一次训练操作并更新模型参数。每当迭代数为 `display_step` 的整数倍时，我们计算一次当前 batch 数据的预测准确率和 loss 并展示出来。

```
with tf.Session() as sess:
    sess.run(init)
    step = 1
    while step * batch_size < max_samples:
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, n_steps, n_input))
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        if step % display_step == 0:
            acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
            loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
```

```

print("Iter " + str(step*batch_size) + ", Minibatch Loss= " + \
      "{:.6f}".format(loss) + ", Training Accuracy= " + \
      "{:.5f}".format(acc))

step += 1
print("Optimization Finished!")

```

全部训练迭代结束后，我们使用训练好的模型，对 mnist.test.images 中全部的测试数据进行预测，并将准确率展示出来。

```

test_len = 10000
test_data = mnist.test.images[:test_len].reshape((-1, n_steps, n_input))
test_label = mnist.test.labels[:test_len]
print("Testing Accuracy:",
      sess.run(accuracy, feed_dict={x: test_data, y: test_label}))

```

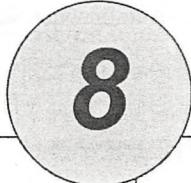
在完成了 40 万个样本的训练后，我们看一下模型在训练集和测试集上的表现。在训练集上我们的预测准确率非常高，基本都是 1，而在包含 10000 个样本的测试集上也有 0.983 的准确率。

```

Iter 394240, Minibatch Loss= 0.025686, Training Accuracy= 0.99219
Iter 395520, Minibatch Loss= 0.001847, Training Accuracy= 1.00000
Iter 396800, Minibatch Loss= 0.009049, Training Accuracy= 1.00000
Iter 398080, Minibatch Loss= 0.015611, Training Accuracy= 1.00000
Iter 399360, Minibatch Loss= 0.009190, Training Accuracy= 1.00000
Optimization Finished!
Testing Accuracy: 0.983

```

Bidirectional LSTM Classifier 在 MNIST 数据集上的表现虽然不如卷积神经网络，但也达到了一个很不错的水平。Bi-RNN 乃至双向 LSTM 网络在时间序列分类任务上能达到较好的表现，是因为它能做到同时利用时间序列的历史和未来信息，结合上下文信息，对结果进行综合判定。虽然在图片这种空间结构显著的数据上不如卷积神经网络，但在无空间结构的单纯的时间序列上，相信 Bi-RNN 和 Bi-LSTM 会更具优势。



8

TensorFlow 实现深度 强化学习

8.1 深度强化学习简介

强化学习（Reinforcement Learning）是机器学习的一个重要分支，主要用来解决连续决策的问题。强化学习可以在复杂的、不确定的环境中学习如何实现我们设定的目标。强化学习的应用场景非常广，几乎包括了所有需要做一系列决策的问题，比如控制机器人的电机让它执行特定任务，给商品定价或者库存管理、玩视频游戏或棋牌游戏等。强化学习也可以应用到有序列输出的问题中，因为它可以针对一系列变化的环境状态，输出一系列对应的行动。举个简单的例子，围棋（乃至全部棋牌类游戏）可以归结为一个强化学习问题，我们需要学习在各种局势下如何走出最好的招法。

一个强化学习问题包含三个主要概念，即环境状态（Environment State）、行动（Action）和奖励（Reward），而强化学习的目标就是获得最多的累计奖励。在围棋中，环境状态就是我们已经下出来的某个局势，行动是指我们在某个位置落子，奖励则是当前这步棋获得的目数（围棋中存在不确定性，在结束对弈后计算的目数是准确的，棋局中获得的目数是估计的），而最终目标就是在结束对弈时总目数超过对手，赢得胜利。我们要让强化学习

模型根据环境状态、行动和奖励，学习出最佳的策略，并以最终结果为目标，不能只看某个行动当下带来的利益（比如围棋中通过某一手棋获得的实地），还要看到这个行动未来能带来的价值（比如围棋中外势可以带来的潜在价值）。我们回顾一下，AutoEncoder 属于无监督学习，而 MLP、CNN 和 RNN 都属于监督学习，但强化学习跟这两种都不同。它不像无监督学习那样完全没有学习目标，也不像监督学习那样有非常明确的目标（即 label），强化学习的目标一般是变化的、不明确的，甚至可能存在绝对正确的标签。

强化学习已经有几十年的历史，但是直到最近几年深度学习技术的突破，强化学习才有了比较大的进展。Google DeepMind 结合强化学习与深度学习，提出 DQN⁶¹（Deep Q-Network，深度 Q 网络），它可以自动玩 Atari 2600 系列的游戏，并取得了超过人类的水平。而 DeepMind 的 AlphaGo⁶²结合了策略网络（Policy Network）、估值网络（Value Network，也即 DQN）与蒙特卡洛搜索树（Monte Carlo Tree Search），实现了具有超高水平的围棋对战程序，并战胜了世界冠军李世石。DeepMind 使用的这些深度强化学习模型（Deep Reinforcement Learning）本质上也是神经网络，主要分为策略网络和估值网络两种。深度强化学习模型对环境没有特别强的限制，可以很好地推广到其他环境，因此对强化学习的研究和发展具有非常重大的意义。下面我们来看看深度强化学习的一些实际应用例子。

无人驾驶是一个非常复杂、非常困难的强化学习任务，在深度学习出现之前，几乎不可能实现。如图 8-1 所示，无人驾驶汽车通过摄像头、雷达、激光测距仪、传感器等对环境进行观测，获取到许多丰富的环境信息，然后通过深度强化学习模型中的 CNN、RNN 等对环境信息进行处理、抽象和转化，再结合强化学习算法框架预测出最应该执行的动作（加速、减速、转换方向等），来实现自动驾驶。无人驾驶汽车每次执行的动作，都会让它到目的地的路程更短，这就是每次行动的奖励。当然，其最终目标是安全地顺利地到达目的地，这样可以获得最多的奖励。

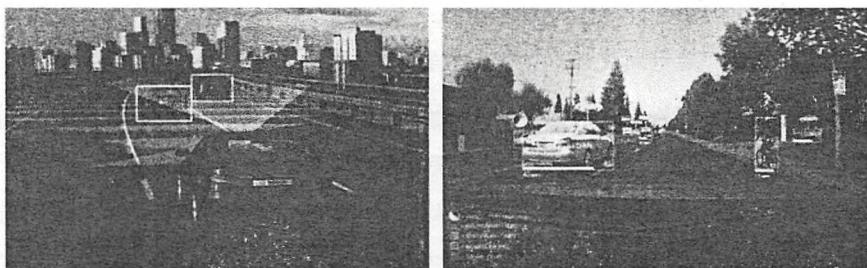


图 8-1 自动驾驶包含了对环境物体的识别及对汽车移动的连续控制

深度强化学习的另一个重要应用是操控复杂的机械装置。一般情况下，我们需要给机械装置编写逻辑非常复杂的控制代码来让它们执行具体的操作，比如控制机械臂拾取小零件。如果要拾取某个特定形状的小零件，需要单独设计一套逻辑，来控制电机进行一系列运转，进而驱动机械臂各个关节转动，最终拾取物体。但是这种做法拾取物体的成功率并不高，而且如果换了一个形状的零件，或者零件的位置发生比较大的变化，那就需要重新设计逻辑。利用深度强化学习算法，我们可以让机器自己学习如何拾取物体，如图 8-2 所示，省去了大量的编程工作。深度强化学习模型中前几层可使用卷积网络，然后可以使用卷积网络对摄像头捕获的图像进行处理和分析，让模型能“看见”环境并识别出物体位置，再通过强化学习框架，学习如何通过一系列动作来最高效地拾取物体。另外，当有新零件出现时，只需要再让机器学习一段时间，就可以掌握抓取新零件的方法，并且这个学习过程可以自动完成，无须人工干预。事实上，通过深度强化学习我们甚至可以让模型学会自动驾驶直升机，这是 Andrew Ng 在讲解强化学习时提到的例子。

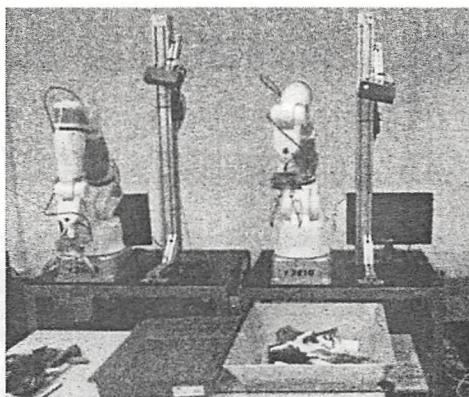


图 8-2 使用深度强化学习模型控制机械臂拾取小零件

同时，我们也可以使用深度强化学习自动玩游戏，如图 8-3 所示，用 DQN 可学习自动玩 *Flappy Bird*。DQN 前几层通常也是卷积层，因此具有了对游戏图像像素 (raw pixels) 直接进行学习的能力。前几层卷积可理解和识别游戏图像中的物体，后层的神经网络则对 Action 的期望价值进行学习，结合这两个部分，可以得到能根据游戏像素自动玩 *Flappy Bird* 的强化学习策略。而且，不仅是这类简单的游戏，连非常复杂的包含大量战术策略的《星际争霸 2》也可以被深度强化学习模型掌握。目前，DeepMind 就在探索如何通过深度强化学习训练一个可以战胜《星际争霸 2》世界冠军的人工智能，这之后的进展让我们拭目以待。

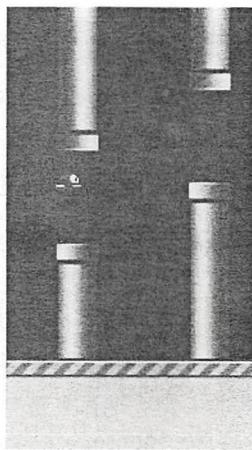


图 8-3 使用深度强化学习自动玩 Flappy Bird

深度强化学习最具有代表性的一个里程碑自然是 AlphaGo。在 2016 年，Google DeepMind 的 AlphaGo 以 4:1 的比分战胜了人类的世界冠军李世石，如图 8-4 所示。围棋可以说是棋类游戏中最为复杂的， 19×19 的棋盘给它带来了 3^{361} 种状态，除去其中非法的违反游戏规则的状态，也有远超整个宇宙中原子数目的状态数。因此，计算机是无法通过像深蓝那样的暴力搜索来战胜人类的，要在围棋这个项目上战胜人类，就必须给计算机抽象思维的能力，而 AlphaGo 做到了这一点。



图 8-4 AlphaGo 代表了深度强化学习技术的巅峰

在 AlphaGo 中使用了快速走子 (Fast Rollout)、策略网络、估值网络和蒙特卡洛搜索树等技术。图 8-5 所示为 AlphaGo 的几种技术单独使用时的表现，横坐标为步数，纵坐标为预测的误差 (可以理解为误差越低模型效果越好)，其中简单的快速走子策略虽然效果比较一般，但是已经远胜随机策略。估值网络和策略网络的效果都非常好，相对来说，策

略网络的性能更胜一筹。AlphaGo 融合了所有这些策略，取得了比单一策略更好的性能，在实战中表现出了惊人的水平。

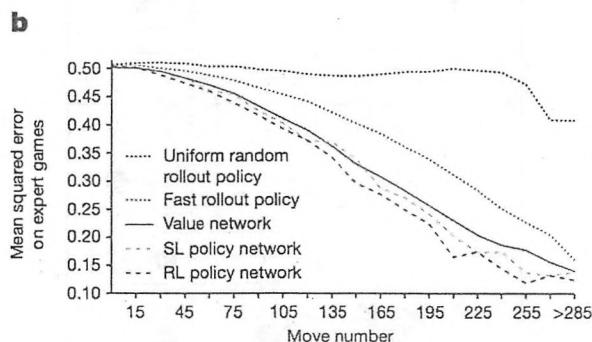


图 8-5 AlphaGo 中随机策略、快速走子、估值网络和策略网络（SL 和 RL 两种）的性能表现

Policy-Based（或者 Policy Gradients）和 Value-Based（或者 Q-Learning）是强化学习中最重要的两类方法，其主要区别在于 Policy-Based 的方法直接预测在某个环境状态下应该采取的 Action，而 Value Based 的方法则预测某个环境状态下所有 Action 的期望价值（Q 值），之后可以通过选择 Q 值最高的 Action 执行策略。这两种方法的出发点和训练方式都有不同，一般来说，Value Based 方法适合仅有少量离散取值的 Action 的环境，而 Policy-Based 方法则更通用，适合 Action 种类非常多或者有连续取值的 Action 的环境。而结合深度学习后，Policy-Based 的方法就成了 Policy Network，而 Value-Based 的方法则成了 Value Network。

图 8-6 所示为 AlphaGo 中的策略网络预测出的当前局势下应该采取的 Action，图中标注的数值为策略网络输出的应该执行某个 Action 的概率，即我们应该在某个位置落子的概率。

图 8-7 所示为 AlphaGo 中估值网络预测出的当前局势下每个 Action 的期望价值。估值网络不直接输出策略，而是输出 Action 对应的 Q 值，即在某个位置落子可以获得的期望价值。随后，我们可以直接选择期望价值最大的位置落子，或者选择其他位置进行探索。

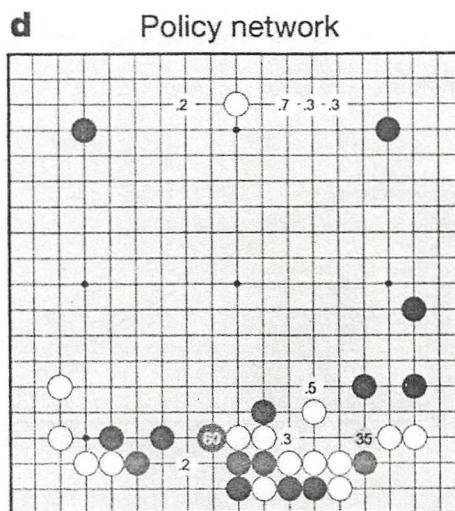


图 8-6 AlphaGo 中的策略网络，输出在某个位置落子的概率

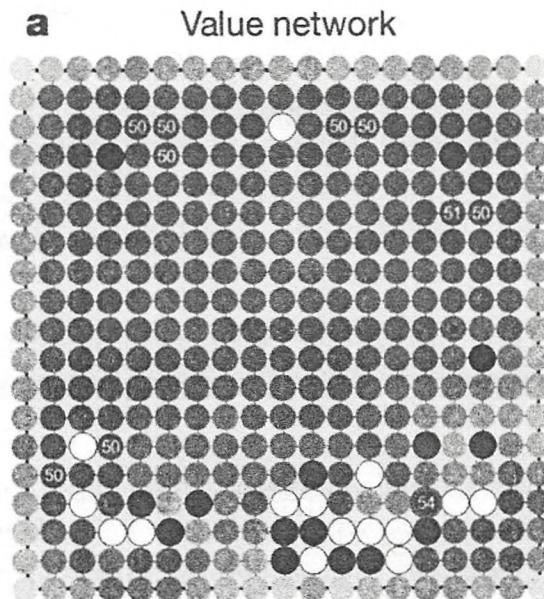


图 8-7 AlphaGo 中的估值网络，输出在某个位置落子的期望价值

在强化学习中，我们也可以建立额外的 model 对环境状态的变化进行预测。普通的强化学习直接根据环境状态预测出行动策略，或行动的期望价值。如果根据环境状态和采取

的行动预测接下来的环境状态，并利用这个信息训练强化学习模型，那就是 model-based RL。对于复杂的环境状态，比如视频游戏的图像像素，要预测这么大量且复杂的环境信息是非常困难的。如果环境状态是数量不大的一些离散值（m），并且可采取的行动也是数量较小的一些离散值（n），那么环境 model 只是一个简单的 $m \times n$ 的转换矩阵。对于一个普通的视频游戏环境，假设图像像素为 $64 \times 64 \times 3$ ，可选行动有 18 种，那么我们光存储这个转换矩阵就需要大的难以想象的内存空间 ($256^{64 \times 64 \times 3} \times 18$)。对于更复杂的环境，我们就更难使用 model 预测接下来的环境状态。而 model-free 类型的强化学习则不需要对环境状态进行任何预测，也不考虑行动将如何影响环境。model-free RL 直接对策略或者 Action 的期望价值进行预测，因此计算效率非常高。当然，如果有一个良好的 model 可以高效、准确地对环境进行预测，会对训练 RL 带来益处；但是一个不那么精准的 model 反而会严重干扰 RL 的训练。因此，对大多数复杂环境，我们主要使用 model-free RL，同时供给更多的样本给 RL 训练，用来弥补没有 model 预测环境状态的问题。

8.2 TensorFlow 实现策略网络

前面提到了强化学习中非常重要的 3 个要素是 Environment State、Action 和 Reward。在环境中，强化学习模型的载体是 Agent，它负责执行模型给出的行动。环境是 Agent 无法控制的，但是可以进行观察；根据观察的结果，模型给出行动，交由 Agent 来执行；而 Reward 是在某个环境状态下执行了某个 Action 而获得的，是模型要争取的目标。在很多任务中，Reward 是延迟获取的（Delayed），即某个 Action 除了可以即时获得 Reward，也可能跟未来获得的 Reward 有很大关系。

所谓策略网络，即建立一个神经网络模型，它可以通过观察环境状态，直接预测出目前最应该执行的策略（Policy），执行这个策略可以获得最大的期望收益（包括现在的和未来的 Reward）。与普通的监督学习不同，在强化学习中，可能没有绝对正确的学习目标，样本的 feature 不再和 label 一一对应。对某一个特定的环境状态，我们并不知道它对应的最好的 Action 是什么，只知道当前 Action 获得的 Reward 还有试验后获得的未来的 Reward。我们需要让强化学习模型通过试验样本自己学习什么才是某个环境状态下比较好的 Action，而不是告诉模型什么才是比较好的 Action，因为我们也不知道正确的答案（即样本没有绝对正确的 label，只有估算出的 label）。我们的学习目标是期望价值，即当前获得的 Reward，加上未来潜在的可获取的 reward。为了更好地让策略网络理解未来的、潜在的 Reward，策略网络不只是使用当前的 Reward 作为 label，而是使用 Discounted Future

Reward，即把所有未来奖励依次乘以衰减系数 γ 。这里的衰减系数一般是一个略小于但接近1的数，防止没有损耗地积累导致 Reward 目标发散，同时也代表了对未来奖励的不确定性的估计。

$$r = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{n-1} r_n$$

我们使用被称为 Policy Gradients 的方法来训练策略网络。Policy Gradients 指的是模型通过学习 Action 在 Environment 中获得的反馈，使用梯度更新模型参数的过程。在训练过程中，模型会接触到好 Action 及它们带来的高期望价值，和差 Action 及它们带来的低期望价值，因此通过对这些样本的学习，我们的模型会逐渐增加选择好 Action 的概率，并降低选择坏 Action 的概率，这样就逐渐完成了我们对策略的学习。和 Q-Learning 或估值网络不同，策略网络学习的不是某个 Action 对应的期望价值 Q，而是直接学习在当前环境应该采取的策略，比如选择每个 Action 的概率（如果是有限个可选 Action，好的 Action 应该对应较大概率，反之亦然），或者输出某个 Action 的具体数值（如果 Action 不是离散值，而是连续值）。因此策略网络是一种 End-to-End（端对端）的方法，可以直接产生最终的策略。

Policy Based 的方法相比于 Value-Based，有更好的收敛性（通常可以保证收敛到局部最优，且不会发散），同时对高维或者连续值的 Action 非常高效（训练和输出结果都更高效），同时能学习出带有随机性的策略。例如，在石头剪刀布的游戏中，任何有规律的策略都会被别人学习到并且被针对，因此完全随机的策略反而可以立于不败之地（起码不会输给别的策略）。在这种情况下，可以利用策略网络学到随机出剪刀、石头、布的策略（三个 Action 的概率相等）。

我们需要使用 Gym⁶³ 辅助我们进行策略网络的训练。Gym 是 OpenAI 推出的开源的强化学习的环境生成工具。OpenAI 是 Tesla 和 Space X 的老板马斯克发起的非营利性的人工智能研究机构。其主要任务是研究安全、开放的人工智能技术，并且确保人工智能技术可以被广泛地、公平地普及，并服务社会。Gym 是 OpenAI 贡献出来的非常重要的开源项目，它的主要作用是为研究者和开发者提供一个方便的强化学习任务环境，例如文字游戏、棋类游戏、视频图像游戏等，并且让用户可以和其他人的强化学习算法进行效率、性能上的比较。

对于强化学习的研究，之前主要受制于两个因素。其一是缺乏高质量的 Benchmark，对于图像识别、监督学习等问题，我们有 ImageNet 这样的经过标注的超大规模数据集，

可以让各种算法在上面进行测试。在强化学习中同样需要大量的、丰富的任务环境，而目前任务环境不仅稀缺，而且设置一个环境的过程也非常繁琐；其二是我们没有一个通用的环境标准，强化学习的相关论文很难进行横向比较，不同任务使用的环境定义、reward 的函数、可用的 Action 都会有区别，而且不同任务的难度可能差异非常大，比如围棋就比国际象棋难很多。Gym 则非常好地解决了这两个问题，提供了大量的标准化的环境，可以用来公平地横向对比强化学习模型的性能。Gym 的用户可以上传模型效果和训练日志到 OpenAI Gym Service 的接口，随后可以参与某个任务的排名，和其他研究者比较模型的效果，并分享算法的思路给其他研究者。

OpenAI Gym 对用户开发模型的方式没有任何限制，它跟其他机器学习库，例如 TensorFlow 和 Theano，都完全兼容。用户可以使用 Python 语言和任何 Python 的 Library 编写强化学习模型的 Agent，比如可以创建一些简单的经验规则，或者使用 State-Action 一一对应的策略表，当然也可以使用深度神经网络模型来做训练模型。

在 Gym 中，有两个核心的概念，一个是 Environment，指我们的任务或者问题，另一个就是 Agent，即我们编写的策略或算法。Agent 会将执行的 Action 传给 Environment，Environment 接受某个 Action 后，再将结果 Observation（即环境状态）和 Reward 返回给 Agent。Gym 中提供了完整的 Environment 的接口，而 Agent 则是完全由用户编写。目前，Gym 一共包含了几个大类的环境，分别是 Algorithmic（算法）、Atari 游戏（使用了 Arcade Learning Environment）、Board Games（棋牌类游戏，其中围棋包含了 9×9 和 19×19 两种规模，目前使用的对抗程序为 Pachi）、Box2D（二维的物理引擎）、Classic Control（经典的控制类问题）、MuJoCo（另一个高效的物理引擎，可以实现非常细节的物理模拟，包括碰撞，可以用来控制 2D 或者 3D 的机器人执行一些任务操作），以及 Toy Text（文本类型）的任务。其中某些任务环境需要额外安装一些依赖库或者程序，我们可以执行 full install 来安装全部环境的依赖程序。

Gym 中环境的接口是 Env 类，其中有几个重要的方法。使用 `env=gym.make('Copy-v0')` 创建某个任务的环境；使用 `env.reset()` 初始化环境，并返回初始的 observation，即 state；使用 `env.step(action)` 在当前状态下执行一步 Action，并返回 observation、reward、done（完成标记）、info（调试信息，但一般不应让 Agent 使用该信息）；使用 `env.render()` 方法可以渲染出一帧的任务图像，很多任务的 observation 就是一帧图像，此时 Agent 直接从图像像素中学习信息和策略。

下面我们就以 Gym 中的 CartPole 环境作为具体例子。CartPole 任务最早由论文

Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem 提出, 是一个经典的可用强化学习来解决的控制问题。如图 8-8 所示, CartPole 的环境中有一辆小车, 在一个一维的无阻力轨道上行动, 在车上绑着一个连接不太结实的杆, 这个杆会左右摇晃。我们的环境信息 observation 并不是图像像素, 而只是一个有 4 个值的数组, 包含了环境中的各种信息, 比如小车位置、速度、杆的角度、速度等。我们并不需要知道每个数值对应的具体物理含义, 因为我们不是要根据这些数值自己编写逻辑控制小车, 而是设计一个策略网络让它自己从这些数值中学习到环境信息, 并制定最佳策略。我们可以采取的 Action 非常简单, 给小车施加一个正向的力或者负向的力。我们有一个 Action Space 的概念, 即 Action 的离散数值空间, 比如在 CartPole 里 Action Space 就是 Discrete(2), 即只有 0 或 1, 其他复杂一点的游戏可能有更多可以选择的值。我们并不需要知道这里的数值会具体对应哪个 Action, 只要模型可以学习到采取这个 Action 之后将会带来的影响就可以, 因此 Action 都只是一个编码。CartPole 的任务目标很简单, 就是尽可能地保持杆竖直不倾倒, 当小车偏离中心超过 2.4 个单位的距离, 或者杆的倾角超过 15 度时, 我们的任务宣告失败, 并自动结束。在每坚持一步后, 我们会获得+1 的 reward, 我们只需要坚持尽量长的时间不导致任务失败即可。任务的 Reward 恒定, 对任何 Action, 只要不导致任务结束, 都可以获得+1 的 Reward。但是我们的模型必须有远见, 要可以考虑到长远的利益, 而不只是学习到当前的 Reward。

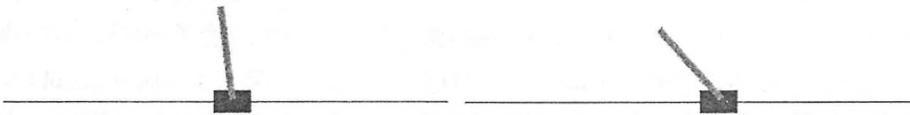


图 8-8 CartPole 环境中包含一个可以控制移动方向的小车和不稳的杆

当我们使用 `env.reset()` 方法后, 就可以初始化环境, 并获取到环境的第一个 Observation。此后, 根据 Observation 预测出应该采取的 Action, 并使用 `env.step(action)` 在环境中执行 Action, 这时会返回 Observation (在 CartPole 中是 4 维的抽象的特征, 在其他任务中可能是图像像素)、reward (当前这步 Action 获得的即时奖励)、done (任务是否结束的标记, 在 CartPole 中是杆倾倒或者小车偏离中心太远, 其他游戏中可能是被敌人击中。如果为 True, 应该 reset 任务) 和 info (额外的诊断信息, 比如标识了游戏中一些随机事件的概率, 但是不应该用来训练 Agent)。这样我们就进入 Action-Observation 的循环, 执行 Action, 获得 Observation, 再执行 Action, 如此往复直到任务结束, 并期望在结束时获得尽可能高的奖励。我们可执行的 Action 在 CartPole 中是离散的数值空间, 即有限的几种可能,

在别的任务中可能是连续的数值，例如在赛车游戏任务中，我们执行的动作是朝某个方向移动，这样我们就有了 0~360 度的连续数值空间可以选择。同时，我们的环境名称后面都带有版本号，比如 V0、V1 等。当环境发生更新或者变化时，我们不会修改之前的环境，而是创建新的版本，这样可以让 Agent 的性能被公平的比较。同时，我们可以调用 env.monitor 方法，对模型的训练过程进行监控和记录，这样之后我们就可以方便地使用 gym.upload 将训练日志上传到 gym service 进行展示，并与他人的算法进行比较。一般来说，对比较简单的问题，我们的评测标准是需要多少步训练就可以稳定地达到理想的分数，并希望需要的训练步数越少越好；对于比较复杂的问题，我们并不知道理想的分数是多少，因此一般是希望获得的分数越高越好。用户可以上传算法到 gym 并让同行审议，其中如果提出非常有效的新算法、新技巧，并且能被其他研究者复现，那对相关领域的研究会有很大价值。

下面就使用 TensorFlow 创建一个基于策略网络的 Agent 来解决 CartPole 问题。我们先安装 OpenAI Gym。本节代码主要来自 DeepRL-Agents⁶⁴ 的开源实现。

```
pip install gym
```

接着，载入 NumPy、TensorFlow 和 gym。这里用 gym.make('CartPole-v0') 创建 CartPole 问题的环境 env。

```
import numpy as np
import tensorflow as tf
import gym
env = gym.make('CartPole-v0')
```

先测试在 CartPole 环境中使用随机 Action 的表现，作为接下来对比的 baseline。首先，我们使用 env.reset() 初始化环境，然后进行 10 次随机试验，这里调用 env.render() 将 CartPole 问题的图像渲染出来。使用 np.random.randint(0,2) 产生随机的 Action，然后用 env.step() 执行随机的 Action，并获取返回的 observation、reward 和 done。如果 done 标记为 True，则代表这次试验结束，即倾角超过 15 度或者偏离中心过远导致任务失败。在一次试验结束后，我们展示这次试验累计的奖励 reward_sum 并重启环境。

```
env.reset()
random_episodes = 0
reward_sum = 0
while random_episodes < 10:
```

```

env.render()
observation, reward, done, _ = env.step(np.random.randint(0,2))
reward_sum += reward
if done:
    random_episodes += 1
    print("Reward for this episode was:",reward_sum)
    reward_sum = 0
    env.reset()

```

可以看到随机策略获得的奖励总值差不多在 10~40 之间，均值应该在 20~30，这将作为接下来用来对比的基准。我们将任务完成的目标设定为拿到 200 的 Reward，并希望通过尽量少次数的试验来完成这个目标。

```

Reward for this episode was: 12.0
Reward for this episode was: 17.0
Reward for this episode was: 20.0
Reward for this episode was: 44.0
Reward for this episode was: 28.0
Reward for this episode was: 19.0
Reward for this episode was: 13.0
Reward for this episode was: 30.0
Reward for this episode was: 20.0
Reward for this episode was: 26.0

```

我们的策略网络使用简单的带有一个隐含层的 MLP。先设置网络的各个超参数，这里隐含节点数 H 设为 50，batch_size 设为 25，学习速率 learning_rate 为 0.1，环境信息 observation 的维度 D 为 4，gamma 即 Reward 的 discount 比例设为 0.99。在估算 Action 的期望价值（即估算样本的学习目标）时会考虑 Delayed Reward，会将某个 Action 之后获得的所有 Reward 做 discount 并累加起来，这样可以让模型学习到未来可能出现的潜在 Reward。注意，一般 discount 比例要小于 1，防止 Reward 被无损耗地不断累加导致发散，这样也可以区分当前 Reward 和未来 Reward 的价值（当前 Action 直接带来的 Reward 不需要 discount，而未来的 Reward 因存在不确定性所以需要 discount）。

```

H = 50
batch_size = 25

```

```
learning_rate = 1e-1
D = 4
gamma = 0.99
```

下面定义策略网络的具体结构。这个网络将接受 observations 作为输入信息，最后输出一个概率值用以选择 Action（我们只有两个 Action，向左施加力或者向右施加力，因此可以通过一个概率值决定）。我们创建输入信息 observations 的 placeholder，其维度为 D。然后使用 tf.contrib.layers.xavier_initializer 初始化算法创建隐含层的权重 W1，其维度为 [D, H]。接着用 tf.matmul 将环境信息 observation 乘上 W1 再使用 ReLU 激活函数处理得到隐含层输出 layer1，这里注意我们并不需要加偏置。同样用 xavier_initializer 算法创建最后 Sigmoid 输出层的权重 W2，将隐含层输出 layer1 乘以 W2 后，使用 Sigmoid 激活函数处理得到最后的输出概率。

```
observations = tf.placeholder(tf.float32, [None,D] , name="input_x")
W1 = tf.get_variable("W1", shape=[D, H],
                     initializer=tf.contrib.layers.xavier_initializer())
layer1 = tf.nn.relu(tf.matmul(observations,W1))
W2 = tf.get_variable("W2", shape=[H, 1],
                     initializer=tf.contrib.layers.xavier_initializer())
score = tf.matmul(layer1,W2)
probability = tf.nn.sigmoid(score)
```

这里模型的优化器使用 Adam 算法。我们分别设置两层神经网络参数的梯度的 placeholder——W1Grad 和 W2Grad，并使用 adam.apply_gradients 定义我们更新模型参数的操作 updateGrads。之后计算参数的梯度，当积累到一定样本量的梯度，就传入 W1Grad 和 W2Grad，并执行 updateGrads 更新模型参数。这里注意，深度强化学习的训练和其他神经网络一样，也使用 batch training 的方式。我们不逐个样本地更新参数，而是累计一个 batch_size 的样本的梯度再更新参数，防止单一样本随机扰动的噪声对模型带来不良影响。

```
adam = tf.train.AdamOptimizer(learning_rate=learning_rate)
W1Grad = tf.placeholder(tf.float32, name="batch_grad1")
W2Grad = tf.placeholder(tf.float32, name="batch_grad2")
batchGrad = [W1Grad,W2Grad]
updateGrads = adam.apply_gradients(zip(batchGrad,tvars))
```

下面定义函数 discount_rewards，用来估算每一个 Action 对应的潜在价值 discount_r。因为 CartPole 问题中每次获得的 Reward 都和前面的 Action 有关，属于 delayed reward。因此需要比较精准地衡量每一个 Action 实际带来的价值时，不能只看当前这一步的 Reward，而要考虑后面的 Delayed Reward。那些能让 Pole 长时间保持在空中竖直的 Action，应该拥有较大的期望价值，而那些最终导致 Pole 倾倒的 Action，则应该拥有较小的期望价值。我们判断越靠后的 Action 的期望价值越小，因为它们更可能是导致 Pole 倾倒的原因，并且判断越靠前的 Action 的期望价值越大，因为它们长时间保持了 Pole 的竖直，和倾倒的关系没有那么大。我们倒推整个过程，从最后一个 Action 开始计算所有 Action 应该对应的期望价值。输入数据 r 为每一个 Action 实际获得的 Reward，在 CartPole 问题中，除了最后结束时的 Action 为 0，其余均为 1。下面介绍具体的计算方法，我们定义每个 Action 除直接获得的 Reward 外的潜在价值为 running_add，running_add 是从后向前累计的，并且需要经过 discount 衰减。而每一个 Action 的潜在价值，即为后一个 Action 的潜在价值乘以衰减系数 gamma 再加上它直接获得的 reward，即 running_add*gamma+r[t]。这样从最后一个 Action 开始不断向前累计计算，即可得到全部 Action 的潜在价值。这种对潜在价值的估算方法符合我们的期望，越靠前的 Action 潜在价值越大。

```
def discount_rewards(r):
    discounted_r = np.zeros_like(r)
    running_add = 0
    for t in reversed(range(r.size)):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

我们定义人工设置的虚拟 label（下文会讲解其生成原理，其取值为 0 或 1）的 placeholder——input_y，以及每个 Action 的潜在价值的 placeholder——advantages。这里 loglik 的定义略显复杂，我们来看一下 loglik 到底代表什么。Action 取值为 1 的概率为 probability（即策略网络输出的概率），Action 取值为 0 的概率为 1-probability，label 取值与 Action 相反，即 label=1-Action。当 Action 为 1 时，label 为 0，此时 loglik=tf.log(probability)，Action 取值为 1 的概率的对数；当 Action 为 0 时，label 为 1，此时 loglik=tf.log(1-probability)，即 Action 取值为 0 的概率的对数。所以，loglik 其实就是当前 Action 对应的概率的对数，我们将 loglik 与潜在价值 advantages 相乘，并取负数作为损失，即优化目标。我们使用优化器优化时，会让能获得较多 advantages 的 Action 的概率变大，并让能获得较少 advantages

的 Action 的概率变小，这样能让损失变小。通过不断的训练，我们便能持续加大能获得较多 advantages 的 Action 的概率，即学习到一个能获得更多潜在价值的策略。最后，使用 `tf.trainable_variables()` 获取策略网络中全部可训练的参数 `tvars`，并使用 `tf.gradients` 求解模型参数关于 loss 的梯度。

```
input_y = tf.placeholder(tf.float32,[None,1], name="input_y")
advantages = tf.placeholder(tf.float32, name="reward_signal")
loglik = tf.log(input_y*(input_y - probability) + \
                  (1 - input_y)*(input_y + probability))
loss = -tf.reduce_mean(loglik * advantages)

tvars = tf.trainable_variables()
newGrads = tf.gradients(loss,tvars)
```

在正式进入训练过程前，我们先定义一些参数，`xs` 为环境信息 `observation` 的列表，`ys` 为我们定义的 `label` 的列表，`drs` 为我们记录的每一个 Action 的 Reward。我们定义累计的 Reward 为 `reward_sum`，总试验次数 `total_episodes` 为 10000，直到达到获取 200 的 Reward 才停止训练。

```
xs,ys,drs = [],[],[]
reward_sum = 0
episode_number = 1
total_episodes = 10000
```

我们创建默认的 Session，初始化全部参数，并在一开始将 `render` 的标志关闭。因为 `render` 会带来比较大的延迟，所以一开始不太成熟的模型没必要去观察。先初始化 CartPole 的环境并获得初始状态。然后使用 `sess.run` 执行 `tvars` 获取所有模型参数，用来创建储存参数梯度的缓冲器 `gradBuffer`，并把 `gradBuffer` 全部初始化为零。接下来的每次试验中，我们将收集参数的梯度存储到 `gradBuffer` 中，直到完成了一个 `batch_size` 的试验，再将汇总的梯度更新到模型参数。

```
with tf.Session() as sess:
    rendering = False
    init = tf.global_variables_initializer()
    sess.run(init)
```

```

observation = env.reset()

gradBuffer = sess.run(tvars)
for ix, grad in enumerate(gradBuffer):
    gradBuffer[ix] = grad * 0

```

下面进入试验的循环，最大循环次数即为 total_episodes。当某个 batch 的平均 Reward 达到 100 以上时，即 Agent 表现良好时，调用 env.render() 对试验环境进行展示。先使用 tf.reshape 将 observation 变形为策略网络输入的格式，然后传入网络中，使用 sess.run 执行 probability 获得网络输出的概率 tfprob，即 Action 取值为 1 的概率。接下来我们在(0, 1) 间随机抽样，若随机值小于 tfprob，则令 Action 取值为 1，否则令 Action 取值为 0，即代表 Action 取值为 1 的概率为 tfprob。

```

while episode_number <= total_episodes:

    if reward_sum/batch_size > 100 or rendering == True :
        env.render()
        rendering = True

    x = np.reshape(observation,[1,D])

    tfprob = sess.run(probability,feed_dict={observations: x})
    action = 1 if np.random.uniform() < tfprob else 0

```

然后将输入的环境信息 observation 添加到列表 xs 中。这里我们制造虚拟的 label——y，它取值与 Action 相反，即 $y=1-\text{action}$ ，并将其添加到列表 ys 中。然后使用 env.step 执行一次 Action，获取 observation、reward、done 和 info，并将 reward 累加到 reward_sum，同时将 reward 添加到列表 drs 中。

```

xs.append(x)
y = 1 - action
ys.append(y)

observation, reward, done, info = env.step(action)
reward_sum += reward

```

```
drs.append(reward)
```

当 done 为 True，即一次试验结束时，将 episode_number 加 1。同时使用 np.vstack 将几个列表 xs、ys、drs 中的元素纵向堆叠起来，得到 epx、epy 和 epr，并将 xs、ys、drs 清空以备下次试验使用。这里注意，epx、epy、drs 即为一次试验中获得的所有 observation、label、reward 的列表。我们使用前面定义好的 discount_rewards 函数计算每一步 Action 的潜在价值，并进行标准化(减去均值再除以标准差)，得到一个零均值标准差为 1 的分布。这么做是因为 discount_reward 会参与到模型损失的计算，而分布稳定的 discount_reward 有利于训练的稳定。

```
if done:
    episode_number += 1
    epx = np.vstack(xs)
    epy = np.vstack(ys)
    epr = np.vstack(drs)
    xs,ys,drs = [],[],[]

    discounted_epr = discount_rewards(epr)
    discounted_epr -= np.mean(discounted_epr)
    discounted_epr /= np.std(discounted_epr)
```

我们将 epx、epy 和 discounted_epr 输入神经网络，并使用操作 newGrads 求解梯度。再将获得的梯度累加到 gradBuffer 中去。

```
tGrad = sess.run(newGrads, feed_dict={observations: epx,
                                       input_y: epy, advantages: discounted_epr})
for ix, grad in enumerate(tGrad):
    gradBuffer[ix] += grad
```

当进行试验的次数达到 batch_size 的整倍数时，gradBuffer 中就累计了足够多的梯度，因此使用 updateGrads 操作将 gradBuffer 中的梯度更新到策略网络的模型参数中，并清空 gradBuffer，为计算下一个 batch 的梯度做准备。这里注意，我们是使用一个 batch 的梯度更新参数，但是每一个梯度是使用一次试验中全部样本（一个 Action 对应一个样本）计算出来的，因此一个 batch 中的样本数实际上是 25 (batch_size) 次试验的样本数之和。

同时，我们展示当前的试验次数 episode_number，和 batch 内每次试验平均获得的 reward。当我们 batch 内每次试验的平均 reward 大于 200 时，我们的策略网络就成功完成了任务，并将终止循环。如果没有达到目标，则清空 reward_sum，重新累计下一个 batch 的总 reward。同时，在每次试验结束后，将任务环境 env 重置，方便下一次试验。

```

if episode_number % batch_size == 0:
    sess.run(updateGrads, feed_dict={W1Grad: gradBuffer[0],
                                     W2Grad: gradBuffer[1]})

    for ix, grad in enumerate(gradBuffer):
        gradBuffer[ix] = grad * 0

    print('Average reward for episode %d : %f.' % \
          (episode_number, reward_sum/batch_size))

    if reward_sum/batch_size > 200:
        print("Task solved in", episode_number, 'episodes!')
        break

    reward_sum = 0

observation = env.reset()

```

下面是我们模型的训练日志，可以看到策略网络在仅经历了 200 次试验，即 8 个 batch 的训练和参数更新后，就实现了我们的目标，达到了 batch 内平均 230 的 reward，顺利完成预设的目标。有兴趣的读者可以尝试修改策略网络的结构、隐含节点数、batch_size、学习速率等参数来尝试优化策略网络的训练，加快其学习到好策略的速度。

```

Average reward for episode 25 : 19.200000.
Average reward for episode 50 : 30.680000.
Average reward for episode 75 : 41.360000.
Average reward for episode 100 : 52.160000.
Average reward for episode 125 : 70.680000.
Average reward for episode 150 : 84.520000.
Average reward for episode 175 : 153.320000.

```

```
Average reward for episode 200 : 230.400000.
```

```
Task solved in 200 episodes!
```

8.3 TensorFlow 实现估值网络

在强化学习中，除了 Policy Based 直接选择 Action 的方法，还有一种学习 Action 对应的期望价值（Expected Utility）的方法，称为 Q-Learning⁶⁵。Q-Learning 最早于 1989 年由 Watkins 提出，其收敛性于 1992 年由 Watkins 和 Dayan 共同证明。Q-Learning 学习中的期望价值指从当前的这一步到所有后续步骤，总共可以期望获取的最大价值（即 Q 值，也可称为 Value）。有了这个 Action→Q 的函数，我们的最佳策略就是在每一个 state 下，选择 Q 值最高的 Action。和 Policy Based 方法一样，Q-Learning 不依赖环境模型。在有限马尔科夫决策过程（Markov Decision Process）中，Q-Learning 被证明最终可以找到最优的策略。

Q-Learning 的目标是求解函数 $Q(s_t, a_t)$ ，即根据当前环境状态，估算 Action 的期望价值。Q-Learning 训练模型的基本思路也非常简单，它以（状态、行为、奖励、下一个状态）构成的元组 $(s_t, a_t, r_{t+1}, s_{t+1})$ 为样本进行训练，其中 s_t 为当前的状态， a 为当前状态下执行的 Action， r_{t+1} 为在执行 Action 后获得的奖励， s_{t+1} 为下一个状态。其中特征是 (s_t, a_t) ，而学习目标（即期望价值）则是 $r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$ ，这个学习目标即是当前 Action 获得的 Reward 加上下一步可获得的最大期望价值。学习目标中包含了 Q-Learning 的函数本身，所以这其中使用了递归求解的思想。下一步可获得的最大期望价值被乘以一个 γ ，即衰减系数 discount factor，这个参数决定了未来奖励在学习中的重要性。如果 discount factor 为 0，那么模型将学习不到任何未来奖励的信息，将会变得短视，只关注当前的利益；如果 discount factor 大于等于 1，那算法很可能无法收敛，期望价值将被不断累加并且没有衰减（即 discount），这样期望价值很可能会发散。因此，discount factor 一般会被设为一个比 1 稍小的值。我们可以把整个 Q-Learning 学习的过程写成下面这个式子：

$$Q_{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q_{old}(s_t, a_t) + \alpha \cdot (r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a))$$

简单描述这个公式是，将旧的 Q-Learning 函数 $Q_{old}(s_t, a_t)$ ，向着学习目标（当前获得的 Reward 加上下一步可获得的最大期望价值）按一个较小的学习速率 α 学习，得到新的 Q-Learning 函数 $Q_{new}(s_t, a_t)$ 。其中学习速率决定了我们使用新获取的样本信息覆盖之前掌握到的信息的比率，通常设为一个比较小的值，可以保证学习过程的稳定，同时确保最后

的收敛性。同时，Q-Learning 需要一个初始值 Q_0 ，而比较高的初始值可以鼓励模型多进行探索。

我们用来学习 Q-Learning 的模型可以是神经网络，这样得到的模型即是估值网络。如果其中的神经网络比较深，那就是 DQN。DQN 这一说法，是由 Google DeepMind 发表于 Nature 的论文 *Human-level control through deep reinforcement learning* 提出的，在这篇论文中 DeepMind 使用 DQN 创建了达到人类专家水平的可以玩 Atari 2600 系列游戏的 Agent。相比于早期 Q-Learning 使用的简单模型，DeepMind 的 DQN 有了很多方面的改进。下面我们将逐一介绍目前 state of the art 的 DQN 中的一些 Trick。

第 1 个 Trick，我们需要在 DQN 中引入卷积层。我们不再是输入一些数值类的特征让模型学习，而是直接让模型通过 Atari 这类游戏的视频图像了解环境信息并学习策略。这样就必须让 DQN 能理解它所接收到的图像，即具有一定的图像识别能力，因此我们就需要用到前几章提到的卷积神经网络。卷积神经网络的具体原理前面几章讲解过，它利用可提取空间结构信息的卷积层来抽取特征。卷积层可以提取图像中重要目标的特征并传给后面的层来做分类或者回归，比如第 6 章中的 VGG Net 和 Inception Net。但 DQN 不同，它使用卷积层不是用来对图像做分类，而是进行强化学习的训练，其目标是根据环境图像输出决策。通常在设计 DQN 时，如果输入是图像，那么最前面几层一般都会设置成卷积层，如图 8-9 所示。本节将要实现的 DQN 的前 4 层也都是卷积层。

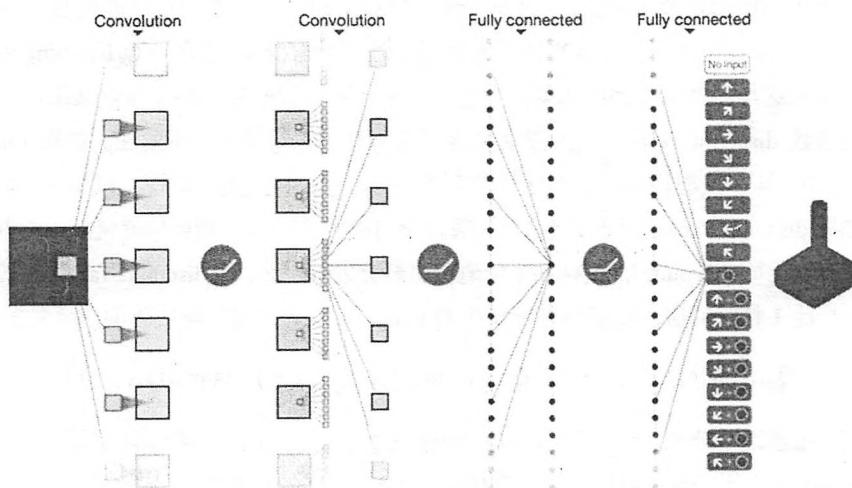


图 8-9 Deep Q-Network 中的多层卷积结构

第 2 个 Trick 是 Experience Replay。因为深度学习需要大量的样本，所以传统的 Q-Learning 的 online update 的方法（逐一对新样本学习的方式）可能不太适合 DQN。因此，我们需要增大样本量，并且像 VGGNet 或 Inception Net 那样进行多个 epoch 的训练，对图像进行反复利用。我们引入一种被称为 Experience Replay 的技术，它的主要思想就是储存 Agent 的 Experience（即样本），并且每次训练时随机抽取一部分样本供给网络学习。这样我们能比较稳定地完成学习任务，避免只短视地学习到最新接触到的样本，而是综合地、反复地利用过往的大量样本进行学习。我们会创建一个用来储存 Experience 的缓存 buffer，它里面可以储存一定量的比较新的样本。当容量满了以后，会用新样本替换最旧的样本，这可以保证大部分样本有相近的概率被抽到，如果不替换旧的，那么从一开始就获得的旧样本，在整个训练过程中被抽到的概率会比新样本高很多。每次需要训练样本时，就直接从 buffer 中随机抽取一定量的样本给 DQN 训练，这样可以保持对样本较高的利用率，同时可以让模型学习到比较新的一批样本。

第 3 个 Trick，我们可以再使用第二个 DQN 网络来辅助训练，这个辅助网络一般称为 target DQN，它的意义是辅助我们计算目标 Q 值，即提供学习目标公式里的 $\max_a Q(s_{t+1}, a)$ 。我们之所以要拆分为两个网络，一个用来制造学习目标，一个来进行实际训练，原因很简单，是为了让 Q-Learning 训练的目标保持平稳。强化学习及 Q-Learning 不像普通的监督学习，它的学习目标每次都是变化的，因为学习目标的一部分是模型本身输出的。每次更新模型参数都会导致我们的学习目标发生变化，如果更新很频繁、幅度很大，我们的训练过程就会非常不稳定并且失控。这样 DQN 的训练就会陷入目标 Q 值与预测 Q 值的反馈循环中（陷入震荡发散，难以收敛）。为了降低这种影响，需要让目标 Q 值尽量平稳，因此需要一个比较稳定的 target DQN 辅助网络计算目标 Q 值。我们让 target DQN 进行低频率或者缓慢的学习，这样它输出的目标 Q 值的波动也会比较小，可以减小对训练过程的影响。

第 4 个 Trick，如果在分拆出 target DQN 的方法上更进一步，那就是 Double DQN。DeepMind 的研究者在论文 *Deep Reinforcement Learning with Double Q-Learning* 中发现，传统的 DQN 通常会高估 Action 的 Q 值。如果这种高估不是均匀的，可能会导致本来最优的某个 Action 总是被高估而超过了最优的 Action，那将给训练和选择 Action 带来很大的麻烦，我们可能永远都发现了不了最优的 Action。因此，在 DeepMind 这篇论文中提出了可以在 DQN 中也使用 Double Q-Learning 的方法。我们之前是让 target DQN 完全负责生成目标 Q 值，即先产生 $Q(s_{t+1}, a)$ ，再通过 \max_a 选择最大的 Q 值。Double DQN 则是修