



---

# 李宏毅深度学习教程

## LeeDL Tutorial

---

<https://github.com/datawhalechina/leedl-tutorial>

Qi Wang Yiyuan Yang Ji Jiang 编著

版本：1.0.1

2023年2月23日

# 目录

<b>第 1 章 深度学习基础</b>	<b>5</b>
1.1 局部极小值与鞍点 . . . . .	5
1.1.1 临界点及其种类 . . . . .	5
1.1.2 判断临界值种类的方法 . . . . .	6
1.1.3 逃离鞍点的方法 . . . . .	10
1.2 批量和动量 . . . . .	13
1.2.1 批量大小对梯度下降法的影响 . . . . .	13
1.2.2 动量法 . . . . .	18
1.3 自适应学习率 . . . . .	22
1.3.1 AdaGrad . . . . .	25
1.3.2 RMSProp . . . . .	26
1.3.3 Adam . . . . .	28
1.4 学习率调度 . . . . .	29
1.5 优化总结 . . . . .	30
<b>第 2 章 卷积神经网络</b>	<b>32</b>
2.1 观察 1: 检测模式不需要整张图像 . . . . .	34
2.2 简化 1: 感受野 . . . . .	35
2.3 观察 2: 同样的模式可能会出现在图像的不同区域 . . . . .	38
2.4 简化 2: 共享参数 . . . . .	39
2.5 简化 1 和 2 的总结 . . . . .	41
2.6 观察 3: 下采样不影响模式检测 . . . . .	45
2.7 简化 3: 汇聚 . . . . .	46
2.8 卷积神经网络的应用: 下围棋 . . . . .	49
<b>第 3 章 自注意力机制</b>	<b>53</b>
3.1 输入是向量序列的情况 . . . . .	53
3.1.1 类型 1: 输入与输出数量相同 . . . . .	56

3.1.2	类型 2：输入是一个序列，输出是一个标签 . . . . .	57
3.1.3	类型 3：序列到序列 . . . . .	58
3.2	自注意力的运作原理 . . . . .	58
3.3	多头注意力 . . . . .	67
3.4	位置编码 . . . . .	69
3.5	截断自注意力 . . . . .	72
3.6	自注意力与卷积神经网络对比 . . . . .	73
3.7	自注意力与循环神经网络对比 . . . . .	75
<b>第 4 章 Transformer</b>		<b>79</b>
4.1	序列到序列模型 . . . . .	79
4.2	序列到序列模型的应用 . . . . .	79
4.2.1	语音识别、机器翻译与语音翻译 . . . . .	79
4.2.2	语音合成 . . . . .	80
4.2.3	聊天机器人 . . . . .	80
4.2.4	问答任务 . . . . .	81
4.2.5	句法分析 . . . . .	82
4.2.6	多标签分类 . . . . .	82
4.3	Transformer 结构 . . . . .	84
4.4	Transformer 编码器 . . . . .	84
4.5	Transformer 解码器 . . . . .	87
4.5.1	自回归解码器 . . . . .	87
4.5.2	非自回归解码器 . . . . .	91
4.6	编码器-解码器注意力 . . . . .	94
4.7	Transformer 的训练过程 . . . . .	95
4.8	序列到序列模型训练常用技巧 . . . . .	97
4.8.1	复制机制 . . . . .	97
4.8.2	引导注意力 . . . . .	98
4.8.3	束搜索 . . . . .	100

4.8.4 加入噪声 . . . . .	101
4.8.5 使用强化学习训练 . . . . .	101
4.8.6 计划采样 . . . . .	102
<b>第 5 章 自监督学习</b>	<b>104</b>
5.1 自监督学习基础 . . . . .	104
5.2 来自 Transformers 的双向编码器表示 (BERT) . . . . .	106
5.2.1 BERT 的使用方式 . . . . .	110
5.2.2 BERT 有用的原因 . . . . .	120
5.2.3 BERT 的变种 . . . . .	125
5.3 生成式预训练 (GPT) . . . . .	129
<b>第 6 章 迁移学习</b>	<b>134</b>
6.1 领域偏移 . . . . .	134
6.2 领域自适应 . . . . .	135
6.3 领域泛化 . . . . .	142
<b>第 7 章 强化学习</b>	<b>144</b>
7.1 监督学习与强化学习的关系 . . . . .	144
7.2 强化学习应用 . . . . .	145
7.2.1 玩视频游戏 . . . . .	145
7.2.2 下围棋 . . . . .	146
7.3 强化学习框架 . . . . .	147
7.3.1 第 1 步: 未知函数 . . . . .	148
7.3.2 第 2 步: 定义损失 . . . . .	149
7.3.3 第 3 步: 优化 . . . . .	149
7.4 评价动作的标准 . . . . .	154
7.4.1 版本 0 . . . . .	154
7.4.2 版本 1 . . . . .	155
7.4.3 版本 2 . . . . .	155
7.4.4 版本 3 . . . . .	157

7.4.5 版本 3.5 . . . . .	160
7.4.6 版本 4 . . . . .	165

# 第 1 章 深度学习基础

## 1.1 局部极小值与鞍点

### 1.1.1 临界点及其种类

我们在优化的时候，怎样将梯度下降做得更好？首先，要理解为什么优化会失败。我们在做优化的时候经常会发现，随着参数不断更新，训练的损失不会再下降，但是我们对这个损失仍然不满意。把深层网络（deep network）跟线性模型（linear model）和浅层网络（shallow network）做比较，可以发现深层网络没有做得更好——深层网络没有发挥出它完整的力量，所以优化是有问题的。但有时候，模型一开始就训练不起来，不管我们怎么更新参数，损失都降不下去。这个时候到底发生了什么事情？过去常见的一个猜想是我们优化到某个地方，这个地方参数对损失的微分为零，如图 1.1 所示。图 1.1 中的两条曲线对应两个神经网络训练的过程。当参数对损失微分为零的时候，梯度下降就不能再更新参数了，训练就停下来了，损失就不再下降了。

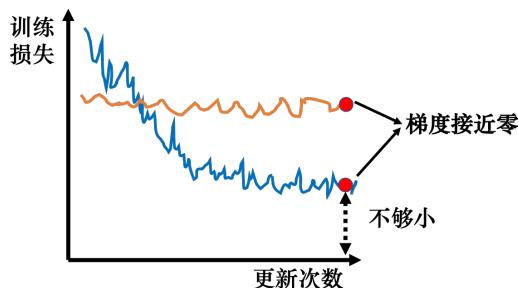


图 1.1 梯度下降失效的情况

提到梯度为零的时候，大家最先想到的可能就是局部极小值（local minimum），如图 1.2a 所示。所以经常有人说，做深度学习时使用梯度下降会收敛在局部极小值，梯度下降不起作用。但其实损失不是只在局部极小值的梯度是零，还有其他可能会让梯度是零的点，比如鞍点（saddle point）。鞍点其实就是梯度是零且区别于局部极小值和局部极大值（local maximum）的点。图 1.2b 红色的点在  $y$  轴方向是比较高的，在  $x$  轴方向是比较低的，这就是一个鞍点。鞍点的叫法是因为其形状像马鞍。鞍点的梯度为零，但它不是局部极小值。我们把梯度为零的点统称为临界点（critical point）。损失没有办法再下降，也许是因为收敛在了临界点，但不一定收敛在局部极小值，因为鞍点也是梯度为零的点。

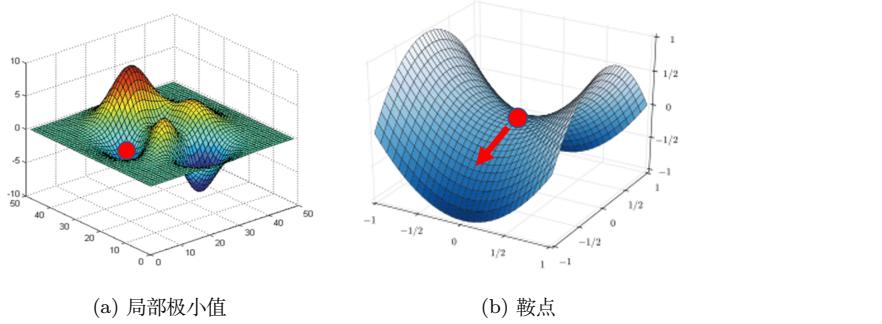


图 1.2 局部极小值与鞍点

但是如果一个点的梯度真的很接近零，我们走到临界点的时候，这个临界点到底是局部极小值还是鞍点，是一个值得去探讨的问题。因为如果损失收敛在局部极小值，我们所在的位置已经是损失最低的点了，往四周走损失都会比较高，就没有路可以走了。但鞍点没有这个问题，旁边还是有路可以让损失更低的。只要逃离鞍点，就有可能让损失更低。

### 1.1.2 判断临界值种类的方法

判断一个临界点到底是局部极小值还是鞍点需要知道损失函数的形状。可是怎么知道损失函数的形状？网络本身很复杂，用复杂网络算出来的损失函数显然也很复杂。虽然我们没有办法完整知道整个损失函数的样子，但是如果给定某一组参数，比如  $\theta'$ ，在  $\theta'$  附近的损失函数是有办法写出来的——虽然  $L(\theta)$  完整的样子写不出来。 $\theta'$  附近的  $L(\theta)$  可近似为

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta'). \quad (1.1)$$

式 (1.1) 是泰勒级数近似 (Taylor series approximation)。其中，第一项  $L(\theta')$  告诉我们，当  $\theta$  跟  $\theta'$  很近的时候， $L(\theta)$  应该跟  $L(\theta')$  还蛮靠近的；第二项  $(\theta - \theta')^T g$  中， $g$  代表梯度，它是一个向量，可以弥补  $L(\theta')$  跟  $L(\theta)$  之间的差距。有时候梯度  $g$  会写成  $\nabla L(\theta')$ 。 $g_i$  是向量  $g$  的第  $i$  个元素，就是  $L$  关于  $\theta$  的第  $i$  个元素的微分，即

$$g_i = \frac{\partial L(\theta')}{\partial \theta_i}. \quad (1.2)$$

光看  $g$  还是没有办法完整地描述  $L(\theta)$ ，还要看式 (1.1) 的第三项  $\frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$ 。第三项跟海森矩阵 (Hessian matrix)  $H$  有关。 $H$  里面放的是  $L$  的二次微分，它第  $i$  行，第  $j$  列的值  $H_{ij}$  就是把  $\theta$  的第  $i$  个元素对  $L(\theta')$  作微分，再把  $\theta$  的第  $j$  个元素对  $\frac{\partial L(\theta')}{\partial \theta_i}$  作微

分后的结果，即

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L(\boldsymbol{\theta}'). \quad (1.3)$$

总结一下，损失函数  $L(\boldsymbol{\theta})$  在  $\boldsymbol{\theta}'$  附近可近似为式 (1.1)，式 (1.1) 跟梯度和海森矩阵有关，梯度就是一次微分，海森矩阵里面有二次微分的项。

在临界点，梯度  $\mathbf{g}$  为零，因此  $(\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{g}$  为零。所以在临界点的附近，损失函数可被近似为

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}') + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}'); \quad (1.4)$$

我们可以根据  $\frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}')$  来判断在  $\boldsymbol{\theta}'$  附近的误差表面 (error surface) 到底长什么样子。知道误差表面的“地貌”，我们就可以判断  $L(\boldsymbol{\theta}')$  是局部极小值、局部极大值，还是鞍点。为了符号简洁，我们用向量  $\mathbf{v}$  来表示  $\boldsymbol{\theta} - \boldsymbol{\theta}'$ ， $(\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}')$  可改写为  $\mathbf{v}^\top \mathbf{H} \mathbf{v}$ ，有如下三种情况。

(1) 如果对所有  $\mathbf{v}$ ， $\mathbf{v}^\top \mathbf{H} \mathbf{v} > 0$ 。这意味着对任意  $\boldsymbol{\theta}$ ， $L(\boldsymbol{\theta}) > L(\boldsymbol{\theta}')$ 。只要  $\boldsymbol{\theta}$  在  $\boldsymbol{\theta}'$  附近， $L(\boldsymbol{\theta})$  都大于  $L(\boldsymbol{\theta}')$ 。这代表  $L(\boldsymbol{\theta}')$  是附近的一个最低点，所以它是局部极小值。

(2) 如果对所有  $\mathbf{v}$ ， $\mathbf{v}^\top \mathbf{H} \mathbf{v} < 0$ 。这意味着对任意  $\boldsymbol{\theta}$ ， $L(\boldsymbol{\theta}) < L(\boldsymbol{\theta}')$ ， $\boldsymbol{\theta}'$  是附近最高的一点， $L(\boldsymbol{\theta}')$  是局部极大值。

(3) 如果对于  $\mathbf{v}$ ， $\mathbf{v}^\top \mathbf{H} \mathbf{v}$  有时候大于零，有时候小于零。这意味着在  $\boldsymbol{\theta}'$  附近，有时候  $L(\boldsymbol{\theta}) > L(\boldsymbol{\theta}')$ ，有时候  $L(\boldsymbol{\theta}) < L(\boldsymbol{\theta}')$ 。因此在  $\boldsymbol{\theta}'$  附近， $L(\boldsymbol{\theta}')$  既不是局部极大值，也不是局部极小值，而是鞍点。

有一个问题，通过  $\frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}')$  判断临界点是局部极小值还是鞍点还是局部极大值，需要代入所有的  $\boldsymbol{\theta}$ 。但我们不可能把所有的  $\mathbf{v}$  都拿来试试看，所以有一个更简便的方法来判断  $\mathbf{v}^\top \mathbf{H} \mathbf{v}$  的正负。算出一个海森矩阵后，不需要把它跟所有的  $\mathbf{v}$  都乘乘看，只要看  $\mathbf{H}$  的特征值。若  $\mathbf{H}$  的所有特征值都是正的， $\mathbf{H}$  为正定矩阵，则  $\mathbf{v}^\top \mathbf{H} \mathbf{v} > 0$ ，临界点是局部极小值。若  $\mathbf{H}$  的所有特征值都是负的， $\mathbf{H}$  为负定矩阵，则  $\mathbf{v}^\top \mathbf{H} \mathbf{v} < 0$ ，临界点是局部极大值。若  $\mathbf{H}$  的特征值有正有负，临界点是鞍点。

如果  $n$  阶对称矩阵  $\mathbf{A}$  对于任意非零的  $n$  维向量  $\mathbf{x}$  都有  $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ ，则称矩阵  $\mathbf{A}$  为正定矩阵。如果  $n$  阶对称矩阵  $\mathbf{A}$  对于任意非零的  $n$  维向量  $\mathbf{x}$  都有  $\mathbf{x}^\top \mathbf{A} \mathbf{x} < 0$ ，则称矩阵  $\mathbf{A}$  为负定矩阵。

举个例子，我们有一个简单的神经网络，它只有两个神经元，而且这个神经元还没有激活

函数和偏置. 输入  $x$ ,  $x$  乘上  $w_1$  以后输出, 然后再乘上  $w_2$ , 接着再输出, 最终得到的数据就是  $y$ .

$$y = w_1 w_2 x. \quad (1.5)$$

我们还有一个简单的训练数据集, 这个数据集只有一组数据  $(1,1)$ , 也就是  $x = 1$  的标签是 1. 所以输入 1 进去, 我们希望最终的输出跟 1 越接近越好, 如图 1.3 所示.

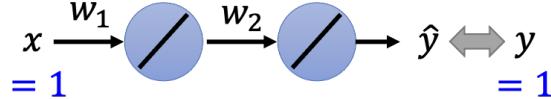


图 1.3 简单的神经网络

可以直接画出这个神经网络的误差表面, 如图 1.4 所示, 可以取  $[-2,2]$  之间的  $w_1$  跟  $w_2$  的数值, 算出这个范围内  $w_1, w_2$  数值所带来的损失, 四个角落的损失是高的. 我们用黑色的点来表示临界点, 原点  $(0,0)$  是临界点, 另外两排点是临界点. 我们可以进一步地判断这些临界点是鞍点还是局部极小值. 原点是鞍点, 因为我们往某个方向走, 损失可能会变大, 也可能会变小. 而另外两排临界点都是局部极小值. 这是我们取  $[-2,2]$  之间的参数得到的损失函数以后, 得到的损失的值后, 画出误差表面后得到的结论.

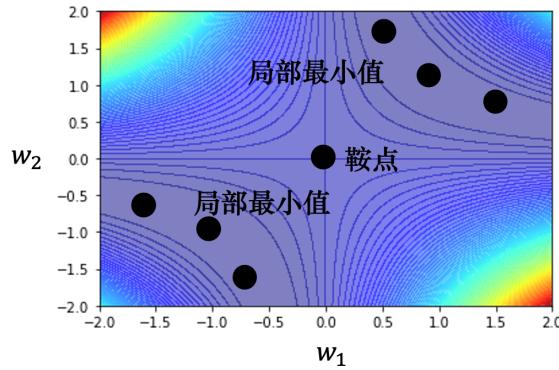


图 1.4 误差表面

除了尝试取所有可能的损失, 我们还有其他的方法, 比如把损失的函数写出来. 对于图 1.3 所示的神经网络, 损失函数  $L$  是正确答案  $y$  减掉模型的输出  $\hat{y} = w_1 w_2 x$  后取平方误差 (square error), 这里只有一组数据, 因此不会对所有的训练数据进行加和. 令  $x = 1, y = 1$ , 损失函数为

$$L = (y - w_1 w_2 x)^2 = (1 - w_1 w_2)^2. \quad (1.6)$$

可以求出损失函数的梯度  $\mathbf{g} = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}]$ :

$$\begin{cases} \frac{\partial L}{\partial w_1} = 2(1 - w_1 w_2)(-w_2); \\ \frac{\partial L}{\partial w_2} = 2(1 - w_1 w_2)(-w_1). \end{cases} \quad (1.7)$$

那么, 什么时候梯度会为零 (也就是到一个临界点) 呢? 比如, 在原点时,  $w_1 = 0, w_2 = 0$ , 此时的梯度为零, 原点就是一个临界点, 但通过海森矩阵才能判断它是哪种临界点. 刚才我们通过取  $[-2, 2]$  之间的  $w_1$  和  $w_2$  来判断出原点是一个鞍点, 但是假设我们还没有取所有可能的损失, 我们要看看能不能够用海森矩阵来判断原点是什么临界点.

海森矩阵  $\mathbf{H}$  收集了  $L$  的二次微分:

$$\begin{cases} H_{1,1} = \frac{\partial^2 L}{\partial w_1^2} = 2(-w_2)(-w_2); \\ H_{1,2} = \frac{\partial^2 L}{\partial w_1 \partial w_2} = -2 + 4w_1 w_2; \\ H_{2,1} = \frac{\partial^2 L}{\partial w_2 \partial w_1} = -2 + 4w_1 w_2; \\ H_{2,2} = \frac{\partial^2 L}{\partial w_2^2} = 2(-w_1)(-w_1). \end{cases} \quad (1.8)$$

对于原点, 只要把  $w_1 = 0, w_2 = 0$  代进去, 有海森矩阵

$$\mathbf{H} = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix}. \quad (1.9)$$

通过海森矩阵来判断原点是局部极小值还是鞍点, 要看它的特征值, 这个矩阵有两个特征值: 2 和 -2, 特征值有正有负, 因此原点是鞍点.

如果我们当前处于鞍点, 就不用那么害怕了.  $\mathbf{H}$  不只可以帮助我们判断是不是在一个鞍点, 还指出了参数可以更新的方向. 之前我们参数更新的时候, 都是看梯度  $\mathbf{g}$ , 但是我们走到某个地方以后发现  $\mathbf{g}$  变成  $\mathbf{0}$  了, 就不能再看  $\mathbf{g}$  了,  $\mathbf{g}$  不见了. 但如果临界点是一个鞍点, 还可以再看  $\mathbf{H}$ , 怎么再看  $\mathbf{H}$  呢,  $\mathbf{H}$  怎么告诉我们怎么更新参数呢?

设  $\lambda$  为  $\mathbf{H}$  的一个特征值  $\lambda$ ,  $\mathbf{u}$  为其对应的特征向量. 对于我们的优化问题, 可令  $\mathbf{u} = \boldsymbol{\theta} - \boldsymbol{\theta}'$ , 则

$$\mathbf{u}^\top \mathbf{H} \mathbf{u} = \mathbf{u}^\top (\lambda \mathbf{u}) = \lambda \|\mathbf{u}\|^2. \quad (1.10)$$

若  $\lambda < 0$ , 则  $\lambda \|\mathbf{u}\|^2 < 0$ . 所以  $\frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}')^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}') < 0$ . 此时,  $L(\boldsymbol{\theta}) < L(\boldsymbol{\theta}')$ , 且

$$\boldsymbol{\theta} = \boldsymbol{\theta}' + \mathbf{u}. \quad (1.11)$$

沿着  $\mathbf{u}$  的方向更新  $\boldsymbol{\theta}$ , 我们就可以让损失变小. 因为根据式 (1.10) 和式 (1.11), 只要  $\boldsymbol{\theta} = \boldsymbol{\theta}' + \mathbf{u}$ , 沿着特征向量  $\mathbf{u}$  的方向去更新参数, 损失就会变小. 所以虽然临界点的梯度为零, 如果我们是在一个鞍点, 只要找出负的特征值, 再找出这个特征值对应的特征向量, 将其与  $\boldsymbol{\theta}'$  相加, 就可以找到一个损失更低的点.

在前面的例子中, 原点是一个临界点, 此时的海森矩阵如式 (1.9) 所示, 该海森矩阵有一个负的特征值:  $-2$ , 特征值  $-2$  对应的特征向量有无穷多个, 不妨取  $\mathbf{u} = [1, 1]^T$ , 作为  $-2$  对应的特征向量. 我们其实只要顺着  $\mathbf{u}$  的方向去更新参数, 就可以找到一个比鞍点的损失还要更低的点. 以这个例子来看, 原点是鞍点, 其梯度为零, 所以梯度不会告诉我们要怎么更新参数. 但海森矩阵的特征向量告诉我们只要往  $[1, 1]^T$  的方向更新, 损失就会变得更小, 就可以逃离鞍点.

所以从这个角度来看, 鞍点似乎并没有那么可怕. 但实际上, 我们几乎不会真的把海森矩阵算出来, 因为海森矩阵需要算二次微分, 计算这个矩阵的运算量非常大, 何况我们还要把它的特征值跟特征向量找出来. 所以几乎没有人用这个方法来逃离鞍点. 还有一些其他逃离鞍点的方法的运算量都比要算海森矩阵小很多.

### 1.1.3 逃离鞍点的方法

讲到这边我们就会有一个问题: 鞍点跟局部极小值谁比较常见? 鞍点其实并没有很可怕, 如果我们经常遇到的是鞍点, 比较少遇到局部极小值, 那就太好了. 科幻小说《三体 III: 死神永生》中有这样一个情节: 东罗马帝国的国王君士坦丁十一世为对抗土耳其人, 找来了具有神秘力量的做狄奥伦娜. 狄奥伦娜可以于万军丛中取上将首级, 但大家不相信她有这么厉害, 想要狄奥伦娜先展示下她的力量. 于是狄奥伦娜拿出了一个圣杯, 大家看到圣杯大吃一惊, 因为这个圣杯本来是放在圣索菲亚大教堂地下室的一个石棺里面, 而且石棺是密封的, 没有人可以打开. 狄奥伦娜不仅取得了圣杯, 还自称在石棺中放了一串葡萄. 于是君士坦丁十一世带人撬开了石棺, 发现圣杯真的被拿走了, 而是棺中真的有一串新鲜的葡萄, 为什么迪奥伦娜可以做到这些事呢? 是因为狄奥伦娜可以进入四维的空间. 从三维的空间来看这个石棺是封闭的, 没有任何路可以进去, 但从高维的空间来看, 这个石棺并不是封闭的, 是有路可以进去的. 误差表面会不会也一样呢.

如图 1.5a 所示的一维空间中的误差表面, 有一个局部极小值. 但是在二维空间 (如图 1.5b 所示), 这个点就可能只是一个鞍点. 常常会有人画类似图 1.5c 这样的图来告诉我们深度学习

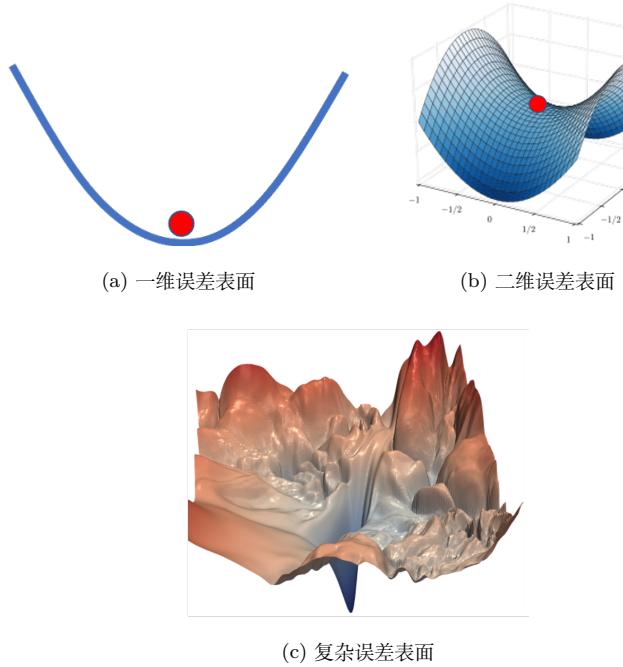


图 1.5 错误表面

的训练是非常复杂的。如果我们移动某两个参数，误差表面的变化非常的复杂，有非常多的局部极小值。低维度空间中的局部极小值点，在更高维的空间中，实际是鞍点。同样地，如果在二维的空间中没有路可以走，会不会在更高维的空间中，其实有路可以走？更高的维度难以可视化，但我们在训练一个网络的时候，参数数量动辄达百万千万级，所以误差表面其实有非常高的维度——我们参数有多少，就代表误差表面的维度是多少。既然维度这么高，会不会其实就有非常多的路可以走呢？既然有非常多的路可以走，会不会其实局部极小值就很少呢？而经验上，我们如果自己做一些实验，会发现实际情况也支持这个假说。图 1.6 是训练某不同神经网络的结果，每个点对应一个神经网络。纵轴代表训练网络时，损失收敛到临界点，损失没法下降时的损失。我们常常会遇到两种情况：损失仍然很高，却遇到了临界点而不再下降；或者损失降得很低，才遇到临界点。图 1.6 中，横轴代表最小值比例（minimum ratio），最小值比例定义为

$$\text{最小值比例} = \frac{\text{正特征值数目}}{\text{负特征值数目}}. \quad (1.12)$$

实际上，我们几乎找不到所有特征值都为正的临界点。在图 1.6 所示的例子中，最小值比例最大也不过处于  $0.5 \sim 0.6$  的范围，代表只有约一半的特征值为正，另一半的特征值为负，代表在所有的维度里面有约一半的路可以让损失上升，还有约一半的路可以让损失下降。虽然在

这个图上，越靠近右侧代表临界点“看起来越像”局部极小值，但是这些点都不是真正的局部极小值。所以从经验上看起来，局部极小值并没有那么常见。多数的时候，我们训练到一个梯度很小的地方，参数不再更新，此时，我们往往只是遇到了鞍点。

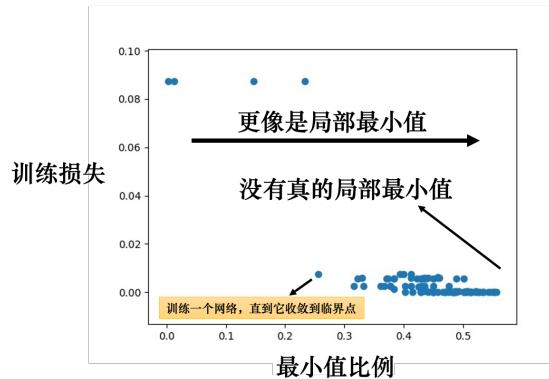


图 1.6 训练不同神经网络的结果

## 1.2 批量和动量

### 1.2.1 批量大小对梯度下降法的影响

实际上在计算梯度的时候，并不是对所有数据的损失  $L$  计算梯度，而是把所有的数据分成一个一个的批量（batch），如图 1.7 所示。每个批量的大小是  $B$ ，即带有  $B$  笔数据。每次在更新参数的时候，会去取出  $B$  笔数据用来计算出损失和梯度更新参数。遍历所有批量的过程称为一个回合（epoch）。事实上，在把数据分为批量的时候，我们还会进行随机打乱（shuffle）。随机打乱有很多不同的做法，一个常见的做法是：在每一个回合开始之前会分一次批量，每个回合的批量的数据都不一样。

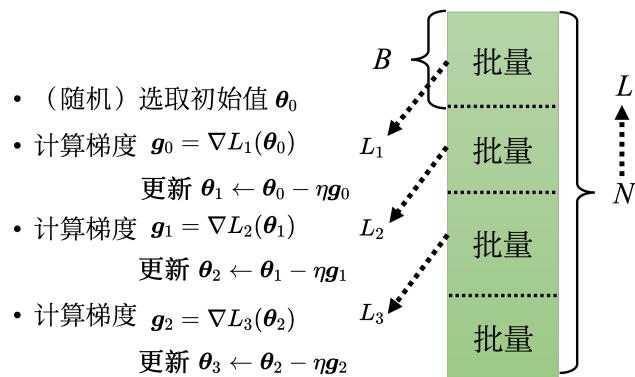


图 1.7 使用批量优化

假设现在我们有 20 笔训练数据，先看下两个最极端的情况。图 1.8 (a) 的情况是没有用批量，批量大小（batch size）为训练数据的大小，这种情况使用全批量（full batch）的数据来更新数据，即批量梯度下降法（Batch Gradient Descent, BGD）。在这种情况下，因为没有用批量，模型必须把 20 笔训练数据都看完，才能够计算损失和梯度，参数才能够更新一次。图 1.8 (b) 的情况就是批量大小等于 1，即随机梯度下降法（Stochastic Gradient Descent, SGD）。随机梯度下降法也称为增量梯度下降法。如果批量大小等于 1，代表只需要拿一笔数据出来算损失，就可以更新一次参数。如果总共有 20 笔数据，在每一个回合里面，参数会更新 20 次。因为现在是只看一笔数据，就更新一次参数，用一笔数据算出来的损失是比较有噪声的，其更新的方向如图 1.8 所示，是曲曲折折的。批量梯度下降没有用批量的方式，要把所有的数据都看过一遍，才能够更新一次参数，因此其每次迭代的计算量大。但批量梯度下降相比随机梯度下降，其每次更新是比较稳定、准确的。

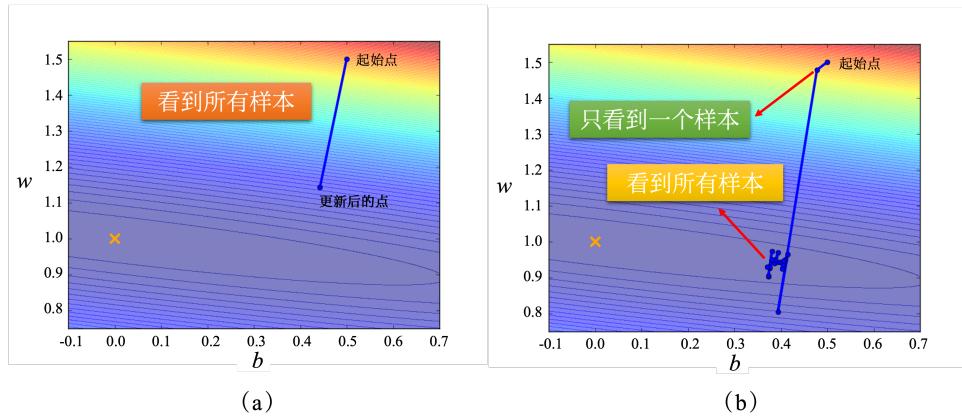


图 1.8 批量梯度下降法与随机梯度下降法

随机梯度下降的梯度上引入了随机噪声，因此在非凸优化问题中，其相比批量梯度下降更容易逃离局部最小值。

实际上考虑并行运算，批量梯度下降并不一定时间比较长。事实上比较大的批量大小，要算损失，再进而算梯度，所需要的时间不一定比小的批量大小要花的时间长。使用 Tesla V100 GPU 在 MNIST 数据集得到的实验结果了如图 1.9 所示。图 1.9 中横坐标表示批量大小，纵坐标表示给定批量大小的批量，计算梯度并更新参数所耗费的时间。批量大小从 1 到 1000，需要耗费的时间几乎是一样的，因为在实际上 GPU 可以做并行运算，这 1000 笔数据是并行处理的，所以 1000 笔数据所花的时间并不是一笔数据的 1000 倍。当然 GPU 并行计算的能力还是存在极限的，当批量大小很大的时候，时间还是会增加的。当批量大小非常大的时候，GPU 在“跑”完一个批量，计算出梯度所花费的时间还是会随着批量大小的增加而逐渐增长。当批量大小增加到 10000，甚至增加到 60000 的时候，GPU 计算梯度并更新参数所耗费的时间确实随着批量大小的增加而逐渐增长。

MNIST 中的“NIST”是指国家标准和技术研究所 (National Institute of Standards and Technology)，其最初收集了这些数据。MNIST 中“M”是指修改的 (Modified)，数据经过预处理以方便机器学习算法使用。MNIST 数据集收集了数万张手写数字 (09) 的  $28 \times 28$  像素的灰度图像及其标签。一般大家第一个会尝试的机器学习的任务，往往就是用 MNIST 做手写数字识别，这个简单的分类问题是深度学习研究中的“Hello World”。

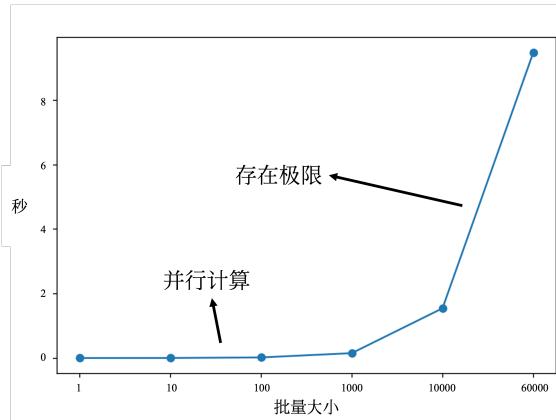


图 1.9 批量大小与计算时间的关系

但是因为有并行计算的能力，因此实际上当批量大小小的时候，要“跑”完一个回合，花的时间是比大的。假设训练数据只有 60000 笔，批量大小设 1，要 60000 个更新才能“跑”完一个回合；如果批量大小等于 1000，60 个更新才能“跑”完一个回合，计算梯度的时间差不多。但 60000 次更新跟 60 次更新比起来，其时间的差距量就非常大了。图 1.10(a) 是用一个批量计算梯度并更新一次参数所需的时间。假设批量大小为 1，“跑”完一个回合，要更新 60000 次参数，其时间是非常大的。但假设批量大小是 1000，更新 60 次参数就会“跑”完一个回合。图 1.10(b) 是“跑”完一个完整的回合需要花的时间。如果批量大小为 1000 或 60000，其时间比批量大小设 1 还要短。图 1.10(a) 和图 1.10(b) 的趋势正好是相反的。因此实际上，在有考虑并行计算的时候，大的批量大小反而是较有效率的，一个回合大的批量花的时间反而是比较少的。

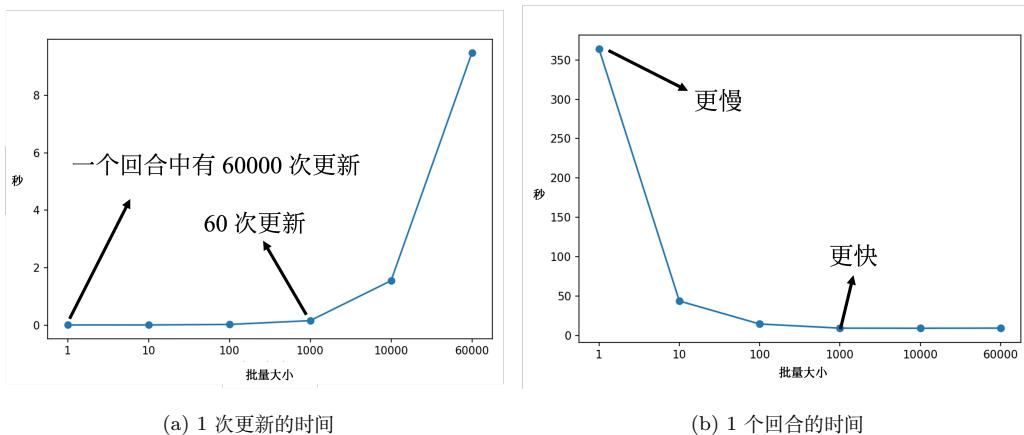


图 1.10 并行计算中批量大小与计算时间的关系

大的批量更新比较稳定，小的批量的梯度的方向是比较有噪声的（noisy）。但实际上有噪声的梯度反而可以帮助训练，如果拿不同的批量来训练模型来做图像识别问题，实验结果如图 1.11 所示，横轴是批量大小，纵轴是正确率。图 1.11(a) 是 MNIST 数据集上的结果，图 1.11(b) 是 CIFAR-10 数据集上的结果。批量大小越大，验证集准确率越差。但这不是过拟合，因为批量大小越大，训练准确率也是越低。因为用的是同一个模型，所以这不是模型偏见的问题。但大的批量大小往往在训练的时候，结果比较差。这个是优化的问题，大的批量大小优化可能会有问题，小的批量大小优化的结果反而是比较好的。

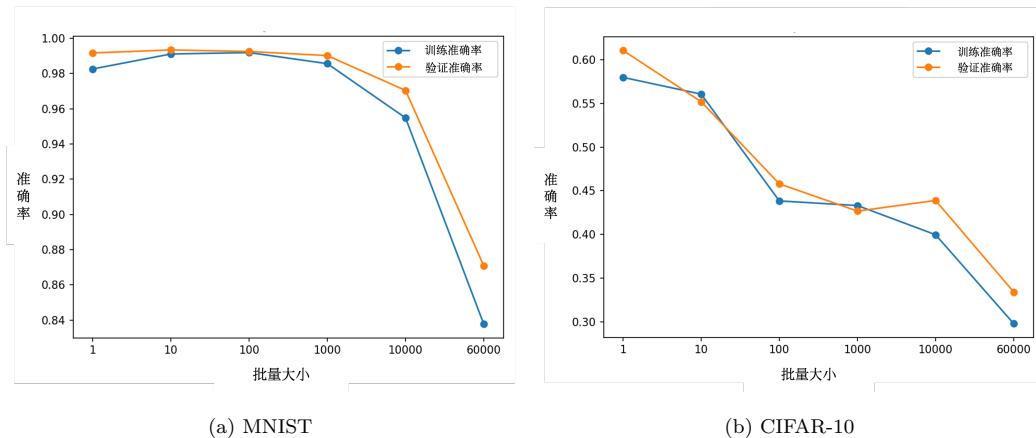


图 1.11 不同的批量来训练模型来做图像识别问题的实验结果

一个可能的解释如图 1.12 所示，批量梯度下降在更新参数的时候，沿着一个损失函数来更新参数，走到一个局部最小值或鞍点显然就停下来了。梯度是零，如果不看 Hessian 矩阵，梯度下降就无法再更新参数了。但小批量梯度下降法（mini-batch gradient descent）每次是挑一个批量计算损失，所以每一次更新参数的时候所使用的损失函数是有差异的。选到第一个批量的时候，用  $L_1$  计算梯度；选到第二个批量的时候，用  $L_2$  计算梯度。假设用  $L_1$  算梯度的时候，梯度是零，就会卡住。但  $L_2$  的函数跟  $L_1$  又不一样， $L_2$  不一定会卡住，可以换下一个批量的损失  $L_2$  计算梯度，模型还是可以训练，还是有办法让损失变小，所以这种有噪声的更新方式反而对训练其实是有帮助的。

其实小的批量也对测试有帮助。假设有一些方法（比如调大的批量的学习率）可以把大的批量跟小的批量训练得一样好。实验结果发现小的批量在测试的时候会是比较好的<sup>[1]</sup>。在论文 “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima” 中，作者在不同数据集上训练了六个网络（包括全连接网络、不同的卷积神经网络），在很多

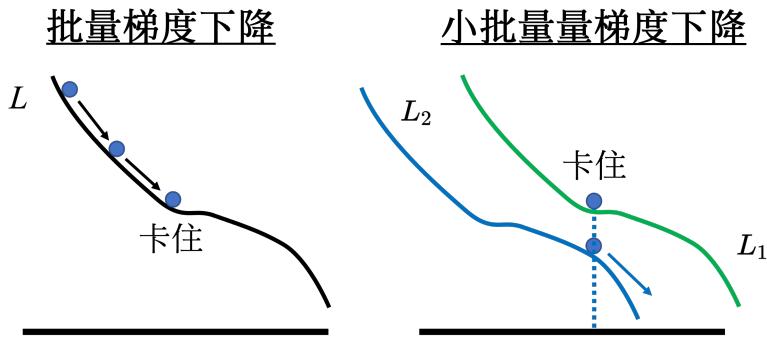


图 1.12 小批量梯度下降更好的原因

不同的情况都观察到一样的结果。在小的批量，一个批量里面有 256 笔样本。在大的批量中，批量大小等于数据集样本数乘 0.1。比如数据集有 60000 笔数据，则一个批量里面有 6000 笔数据。作者想办法，在大的批量跟小的批量，都训练到差不多的训练准确率 (accuracy)。但是就算是在训练的时候结果差不多，测试的时候，大的批量比小的批量差，代表过拟合。在这篇文章里面也给出了一个解释，如图 1.13 所示，训练损失上面有多个局部最小值，这些局部最小值的损失都很低，其损失可能都趋近于 0。但是局部最小值有好最小值跟坏最小值之分，如果局部最小值在一个“峡谷”里面，它是坏的最小值；如果局部最小值在一个平原上，它是好的最小值。训练的损失跟测试的损失函数是不一样的，这有两种可能。一种可能是本来训练跟测试的分布就不一样，另一种可能是因为训练跟测试都是从采样的数据算出来的，也许训练跟测试采样到的数据不一样，所以它们计算出的损失是有一点差距。对在一个“盆地”里面的最小值，其在训练跟测试上面的结果不会差太多，只差了一点点。但是对右边这个在“峡谷”里面的最小值来说，一差就可以天差地远，它在这个训练集上算出来的损失很低，但是因为训练跟测试之间的不一样，所以测试的时候，误差表面 (error surface) 一变，计算出的损失就变得很大。大的批量大小会让我们倾向于走到“峡谷”里面，而小的批量大小倾向于让我们走到“盆地”里面。小的批量有很多的损失，其更新方向比较随机，其每次更新的方向都不太一样。即使“峡谷”非常窄，它也可以跳出去，之后如果有一个非常宽的“盆地”，它才会停下来。

大的批量跟小的批量的对比结果如表 1.1 所示。在有并行计算的情况下，小的批量跟大的批量运算的时间并没有太大的差距。除非大的批量非常大，才会显示出差距。但是一个回合需要的时间，小的批量比较长，大的批量反而是比较快的，所以从一个回合需要的时间来看，大的批量是较有优势的。而小的批量更新的方向比较有噪声的，大的批量更新的方向比较稳

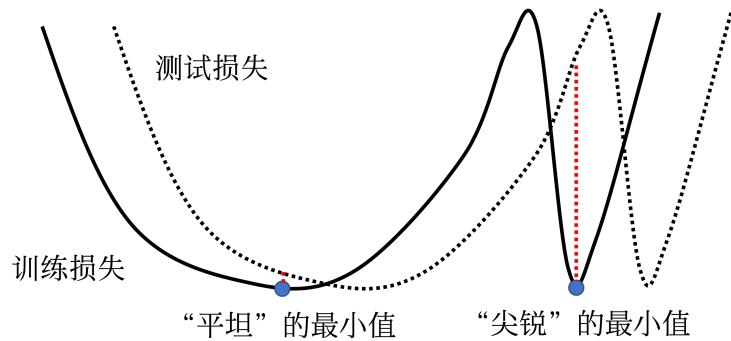


图 1.13 小批量优化容易跳出局部最小值的原因

定。但是有噪声的更新方向反而在优化的时候有优势，而且在测试的时候也会有优势。所以大的批量跟小的批量各有优缺点，批量大小是需要去调整的超参数。

其实用大的批量大小来做训练，用并行计算的能力来增加训练的效率，并且训练出的结果很好是可以做到的<sup>[2-3]</sup>。比如 76 分钟训练 BERT<sup>[4]</sup>，15 分钟训练 ResNet<sup>[5]</sup>，一分钟训练 ImageNet<sup>[6]</sup> 等等。这些论文批量大小很大，比如论文 “Large Batch Optimization for Deep Learning: Training BERT in 76 minutes” 中批量大小为三万。批量大小很大可以算得很快，这些论文有一些特别的方法来解决批量大小可能会带来的劣势。

表 1.1 小批量梯度下降与批量梯度下降的比较

评价标准	小批量梯度下降	批量梯度下降
一次更新的速度（没有并行计算）	相同	相同
一次更新的速度（有并行计算）	相同	相同（批量大小不是很大）
一个回合的时间	更慢	更快
梯度	有噪声	稳定
优化	更好	更坏
泛化	更好	更坏

### 1.2.2 动量法

动量法 (momentum method) 是另外一个可以对抗鞍点或局部最小值的方法。如图 1.14 所示，假设误差表面就是真正的斜坡，参数是一个球，把球从斜坡上滚下来，如果使用梯度下降，球走到局部最小值或鞍点就停住了。但是在物理的世界里，一个球如果从高处滚下来，

就算滚到鞍点或鞍点，因为惯性的关系它还是会继续往前走。如果球的动量足够大，其甚至翻过小坡继续往前走。因此在物理的世界里面，一个球从高处滚下来的时候，它并不一定会被鞍点或局部最小值卡住，如果将其应用到梯度下降中，这就是动量。

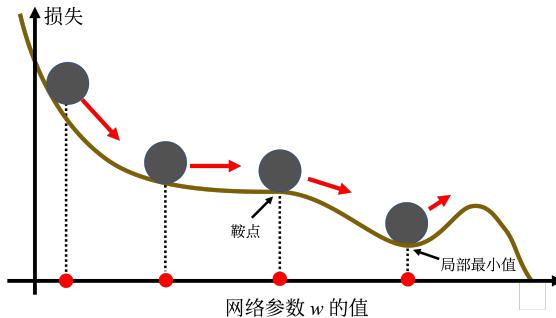


图 1.14 物理世界中的惯性

一般的梯度下降 (vanilla gradient descent) 如图 1.15 所示。初始参数为  $\theta_0$ ，计算一下梯度，计算完梯度后，往梯度的反方向去更新参数  $\theta_1 = \theta_0 - \eta g_0$ 。有了新的参数  $\theta_1$  后，再计算一次梯度，再往梯度的反方向，再更新一次参数，到了新的位置以后再计算一次梯度，再往梯度的反方向去更新参数。

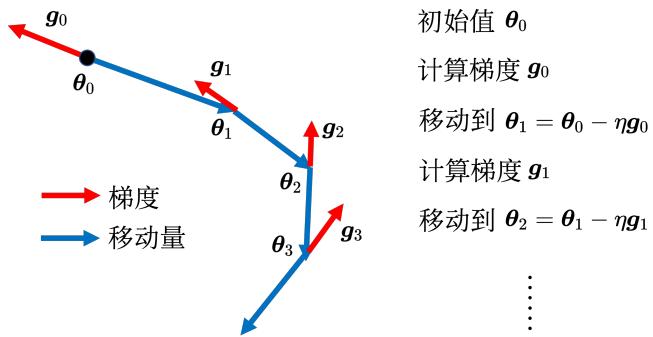


图 1.15 一般梯度下降

引入动量后，每次在移动参数的时候，不是只往梯度的反方向来移动参数，而是根据梯度的反方向加上前一步移动的方向决定移动方向。图 1.16 中红色虚线方向是梯度的反方向，蓝色虚线方向是前一次更新的方向，蓝色实线的方向是下一步要移动的方向。把前一步指示的方向跟梯度指示的方向相加就是下一步的移动方向。如图 1.16 所示，初始的参数值为  $\theta_0 = 0$ ，前一步的参数的更新量为  $m_0 = 0$ 。接下来在  $\theta_0$  的地方，计算梯度的方向  $g_0$ 。下一步的方向是梯度的方向加上前一步的方向，不过因为前一步正好是 0，所以更新的方向跟原来的梯度下

降是相同的。但从第二步开始就不太一样了。从第二步开始，计算  $\mathbf{g}_1$ ，接下来更新的方向为  $\mathbf{m}_2 = \lambda\mathbf{m}_1 - \eta\mathbf{g}_1$ ，参数更新为  $\theta_2$ ，接下来就反复进行同样的过程。

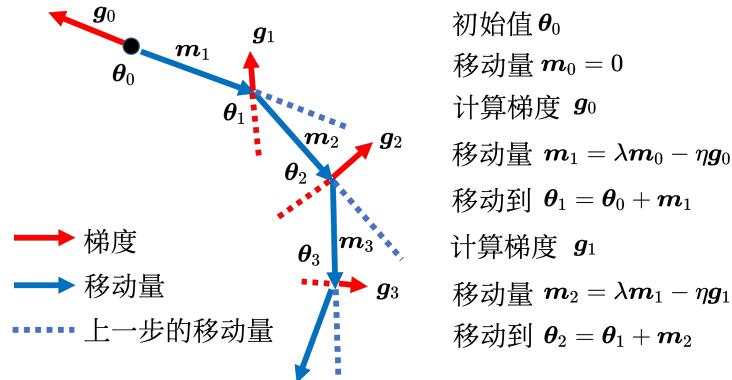


图 1.16 动量法

每一步的移动都用  $\mathbf{m}$  来表示。 $\mathbf{m}$  其实可以写成之前所有计算的梯度的加权和，如式 (1.13) 所示。其中  $\eta$  是学习率， $\lambda$  是前一个方向的权重参数，也是需要调的。引入动量后，可以从两个角度来理解动量法。一个角度是动量是梯度的负反方向加上前一次移动的方向。另外一个角度是当加上动量的时候，更新的方向不是只考虑现在的梯度，而是考虑过去所有梯度的总和。

$$\mathbf{m}_0 = 0$$

$$\begin{aligned} \mathbf{m}_1 &= -\eta\mathbf{g}_0 \\ \mathbf{m}_2 &= -\lambda\eta\mathbf{g}_0 - \eta\mathbf{g}_1 \\ &\vdots \end{aligned} \tag{1.13}$$

动量的简单例子如图 1.17 所示。红色表示负梯度方向，蓝色虚线表示前一步的方向，蓝色实线表示真实的移动量。一开始没有前一次更新的方向，完全按照梯度给指示往右移动参数。负梯度方向跟前一步移动的方向加起来，得到往右走的方向。一般梯度下降走到一个局部最小值或鞍点时，就被困住了。但有动量还是有办法继续走下去，因为动量不是只看梯度，还看前一步的方向。即使梯度方向往左走，但如果前一步的影响力比梯度要大，球还是有可能继续往右走，甚至翻过一个小丘，也许可以走到更好的局部最小值，这就是动量有可能带来的好处。

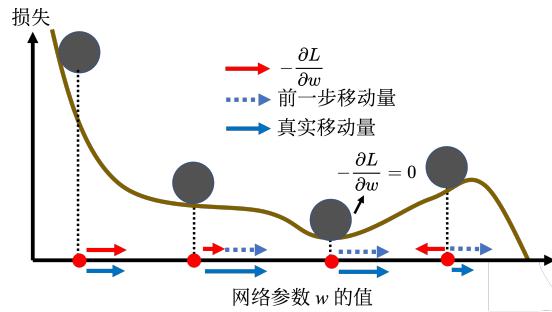


图 1.17 动量的好处

## 参考文献

- [1] KESKAR N S, MUDIGERE D, NOCEDAL J, et al. On large-batch training for deep learning: Generalization gap and sharp minima[J]. arXiv preprint arXiv:1609.04836, 2016.
- [2] GUPTA V, SERRANO S A, DECOSTE D. Stochastic weight averaging in parallel: Large-batch training that generalizes well[J]. arXiv preprint arXiv:2001.02312, 2020.
- [3] YOU Y, GITMAN I, GINSBURG B. Large batch training of convolutional networks[J]. arXiv preprint arXiv:1708.03888, 2017.
- [4] YOU Y, LI J, REDDI S, et al. Large batch optimization for deep learning: Training bert in 76 minutes[J]. arXiv preprint arXiv:1904.00962, 2019.
- [5] AKIBA T, SUZUKI S, FUKUDA K. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes[J]. arXiv preprint arXiv:1711.04325, 2017.
- [6] GOYAL P, DOLLÁR P, GIRSHICK R, et al. Accurate, large minibatch sgd: Training imagenet in 1 hour[J]. arXiv preprint arXiv:1706.02677, 2017.

### 1.3 自适应学习率

临界点其实不一定是在训练一个网络的时候会遇到的最大的障碍。图 1.18 中的横坐标代表参数更新的次数，竖坐标表示损失。一般在训练一个网络的时候，损失原来很大，随着参数不断的更新，损失会越来越小，最后就卡住了，损失不再下降。当我们走到临界点的时候，意味着梯度非常小，但损失不再下降的时候，梯度并没有真的变得很小，图 1.19 给出了示例。图 1.19 中横轴是迭代次数，竖轴是梯度的范数（norm），即梯度这个向量的长度。随着迭代次数增多，虽然损失不再下降，但是梯度的范数并没有真的变得很小。

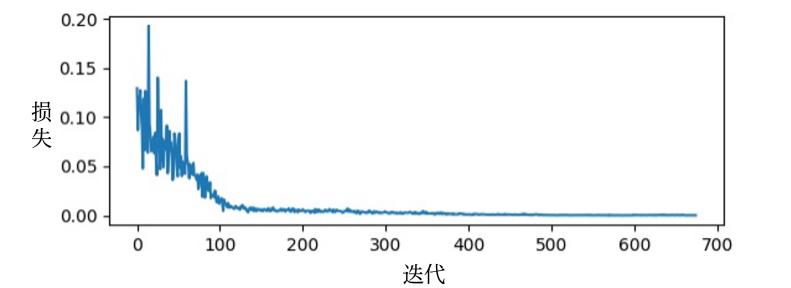


图 1.18 训练网络时损失变化

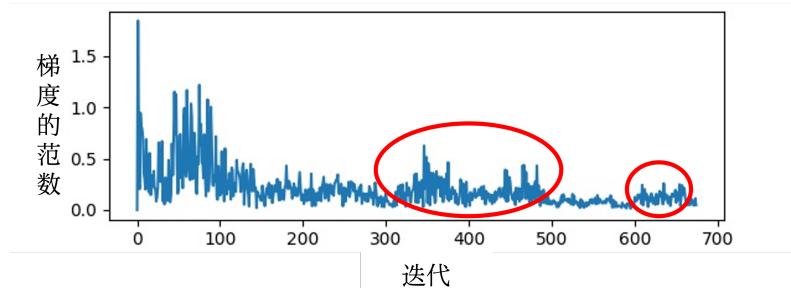


图 1.19 训练网络时梯度范数变化

图 1.20 是误差表面，梯度在山谷的两个谷壁间，不断地来回“震荡”，这个时候损失不会再下降，它不是真的卡到了临界点，卡到了鞍点或局部最小值。但它的梯度仍然很大，只是损失不一定再减小了。所以训练一个网络，训练到后来发现损失不再下降的时候，有时候不是卡在局部最小值或鞍点，只是单纯的损失无法再下降。

如图 1.21 所示，我们现在训练一个网络，训练到现在参数在临界点附近，再根据特征值的正负号判断该临界点是鞍点还是局部最小值。实际上在训练的时候，要走到鞍点或局部最小值，是一件困难的事情。要训练出图 1.21 的结果，要训练到参数很接近临界点，一般的梯

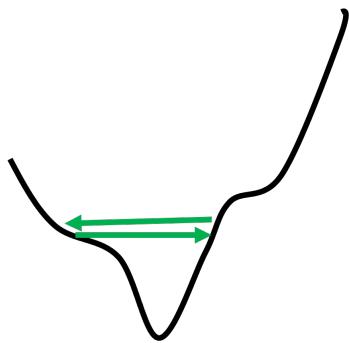


图 1.20 梯度来回“震荡”

度下降，其实是做不到。用一般的梯度下降训练，往往会在梯度还很大的时候，损失就已经降了下去，这个是需要特别方法训练的。要走到一个临界点其实是比较困难的，多数时候训练在还没有走到临界点的时候就已经停止了。

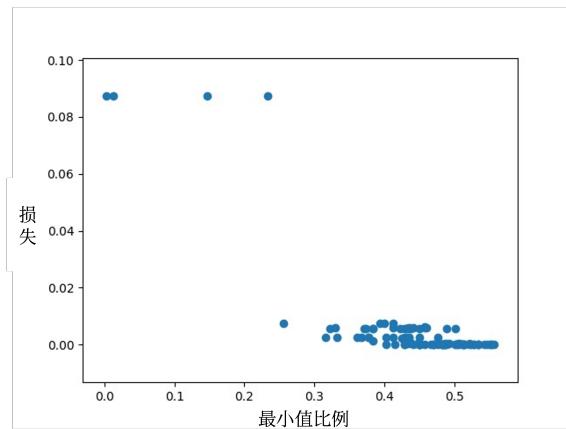


图 1.21 训练不同神经网络的结果

举个例子，我们有两个参数  $w$  和  $b$ ，这两个参数值不一样的时候，损失值也不一样，得到了图 1.22 所示的误差表面，该误差表面的最低点在叉号处。事实上，该误差表面是凸的形状。凸的误差表面的等高线是椭圆形的，椭圆的长轴非常长，短轴相比之下比较短，其在横轴的方向梯度非常小，坡度的变化非常小，非常平坦；其在纵轴的方向梯度变化非常大，误差表面的坡度非常陡峭。现在我们要从黑点（初始点）来做梯度下降。

学习率  $\eta = 10^{-2}$  的结果如图 1.23a 所示。参数在峡谷的两端，参数在山壁的两端不断第“震荡”，损失降不下去，但是梯度仍然是很大的。我们可以试着把学习率设小一点，学习率决定了更新参数的时候的步伐，学习率设太大，步伐太大就无法慢慢地滑到山谷里面。调学习率从  $10^{-2}$  调到  $10^{-7}$  的结果如图 1.23b 所示，参数不再“震荡”了。参数会滑到山谷底后左

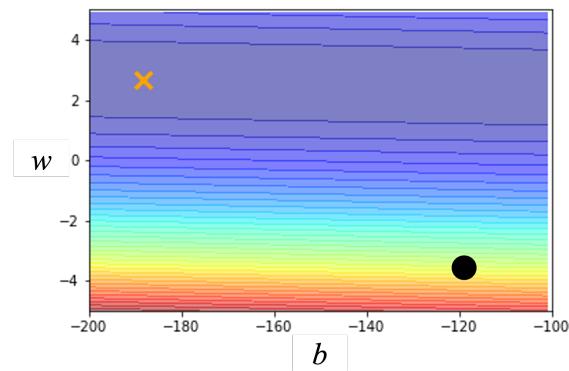


图 1.22 凸误差表面

转，但是这个训练永远走不到终点，因为学习率已经太小了。AB 段的坡度很陡，梯度的值很大，还能够前进一点。左拐以后，BC 段的坡度已经非常平坦了，这种小的学习率无法再让训练前进。事实上在 BC 段有 10 万个点（10 万次更新），但都无法靠近局部最小值，所以显然就算是一个凸的误差表面，梯度下降也很难训练。

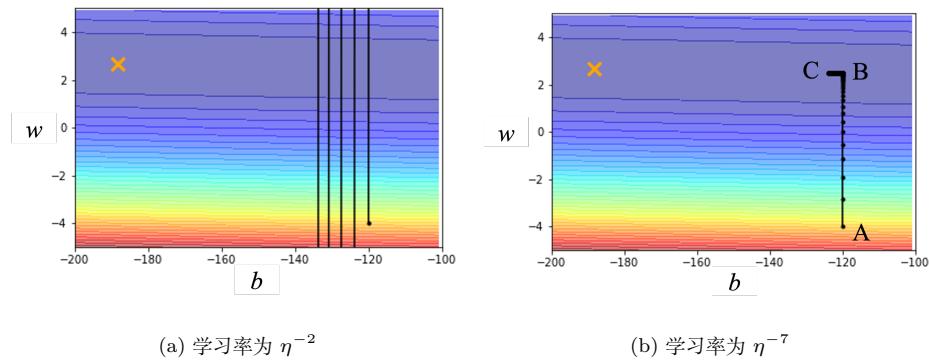


图 1.23 不同的学习率对训练的影响

最原始的梯度下降连简单的误差表面都做不好，因此需要更好的梯度下降的版本。在梯度下降里面，所有的参数都是设同样的学习率，这显然是不够的，应该要为每一个参数定制化学习率，即引入自适应学习率（adaptive learning rate）的方法，给每一个参数不同的学习率。如图 1.24 所示，如果在某一个方向上，梯度的值很小，非常平坦，我们会希望学习率调大一点；如果在某一个方向上非常陡峭，坡度很大，我们会希望学习率可以设得小一点。

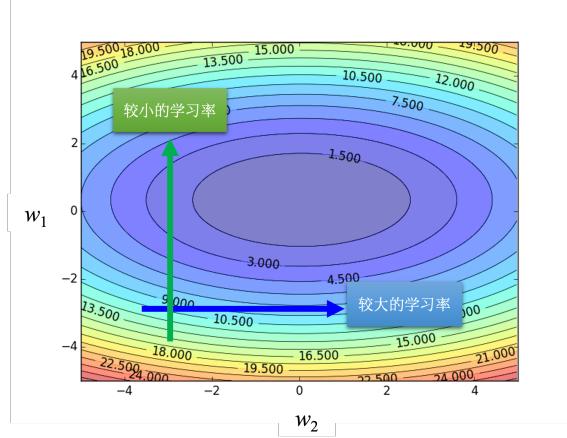


图 1.24 不同参数需要不同的学习率

### 1.3.1 AdaGrad

AdaGrad (Adaptive Gradient) 是典型的自适应学习率方法，其能够根据梯度大小自动调整学习率。AdaGrad 可以做到梯度比较大的时候，学习率就减小，梯度比较小的时候，学习率就放大。

梯度下降更新某个参数  $\theta_t^i$  的过程为

$$\theta_{t+1}^i \leftarrow \theta_t^i - \eta g_t^i \quad (1.14)$$

$\theta_t^i$  在第  $t$  个迭代的值减掉在第  $t$  个迭代参数  $i$  算出来的梯度

$$g_t^i = \frac{\partial L}{\partial \theta^i} \Big|_{\theta=\theta_t} \quad (1.15)$$

$g_t^i$  代表在第  $t$  个迭代，即  $\theta = \theta_t$  时，参数  $\theta^i$  对损失  $L$  的微分，学习率是固定的。

现在要有一个随着参数定制化的学习率，即把原来学习率  $\eta$  变成  $\frac{\eta}{\sigma_t^i}$

$$\theta_{t+1}^i \leftarrow \theta_t^i - \frac{\eta}{\sigma_t^i} g_t^i \quad (1.16)$$

$\sigma_t^i$  的上标为  $i$ ，这代表参数  $\sigma$  与  $i$  相关，不同的参数的  $\sigma$  不同。 $\sigma_t^i$  的下标为  $t$ ，这代表参数  $\sigma$  与迭代相关，不同的迭代也会有不同的  $\sigma$ 。学习率从  $\eta$  改成  $\frac{\eta}{\sigma_t^i}$  的时候，学习率就变得参数相关 (parameter dependent)。

参数相关的一个常见的类型是算梯度的均方根 (root mean square)。参数的更新过程为

$$\theta_1^i \leftarrow \theta_0^i - \frac{\eta}{\sigma_0^i} g_0^i \quad (1.17)$$

其中  $\theta_0^i$  是初始化参数。而  $\sigma_0^i$  的计算过程为

$$\sigma_0^i = \sqrt{(\mathbf{g}_0^i)^2} = |\mathbf{g}_0^i| \quad (1.18)$$

其中  $\mathbf{g}_0^i$  是梯度。将  $\sigma_0^i$  的值代入更新的公式可知  $\frac{\mathbf{g}_0^i}{\sigma_0^i}$  的值是 +1 或 -1。第一次在更新参数，从  $\theta_0^i$  更新到  $\theta_1^i$  的时候，要么是加上  $\eta$ ，要么是减掉  $\eta$ ，跟梯度的大小无关，这个是第一步的情况。

第二次更新参数过程为

$$\theta_2^i \leftarrow \theta_1^i - \frac{\eta}{\sigma_1^i} \mathbf{g}_1^i \quad (1.19)$$

其中  $\sigma_1^i$  是过去所有计算出来的梯度的平方的平均再开根号，即均方根，如式 (1.20) 所示。

$$\sigma_1^i = \sqrt{\frac{1}{2} [(\mathbf{g}_0^i)^2 + (\mathbf{g}_1^i)^2]} \quad (1.20)$$

同样的操作反复继续下去，如式 (1.21) 所示。

$$\begin{aligned} \theta_3^i &\leftarrow \theta_2^i - \frac{\eta}{\sigma_2^i} \mathbf{g}_2^i & \sigma_2^i = \sqrt{\frac{1}{3} [(\mathbf{g}_0^i)^2 + (\mathbf{g}_1^i)^2 + (\mathbf{g}_2^i)^2]} \\ &\vdots \end{aligned} \quad (1.21)$$

第  $t+1$  次更新参数的时候，即

$$\theta_{t+1}^i \leftarrow \theta_t^i - \frac{\eta}{\sigma_t^i} \mathbf{g}_t^i \quad \sigma_t^i = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (\mathbf{g}_t^i)^2} \quad (1.22)$$

$\frac{\eta}{\sigma_t^i}$  当作是新的学习率来更新参数。

图 1.25 中有两个参数： $\theta^1$  和  $\theta^2$ 。 $\theta^1$  坡度小， $\theta^2$  坡度大。因为  $\theta^1$  坡度小，根据式 (1.22)， $\theta_1^i$  这个参数上面算出来的梯度值都比较小，因为梯度算出来的值比较小，所以算出来的  $\sigma_t^i$  就小， $\sigma_t^i$  小学习率就大。反过来， $\theta^1$  坡度大，所以计算出的梯度都比较大， $\sigma_t^i$  就比较大，在更新的时候，步伐（参数更新的量）就比较小。因此有了  $\sigma_t^i$  这一项以后，就可以随着梯度的不同，每一个参数的梯度的不同，来自动调整学习率的大小。

### 1.3.2 RMSProp

同一个参数需要的学习率，也会随着时间而改变。在图 1.26 中的误差表面中，如果考虑横轴方向，绿色箭头处坡度比较陡峭，需要较小的学习率，但是走到红色箭头处，坡度变得平坦了起来，需要较大的学习率。因此同一个参数的同个方向，学习率也是需要动态调整的，于是就有了一个新的方法——RMSprop。

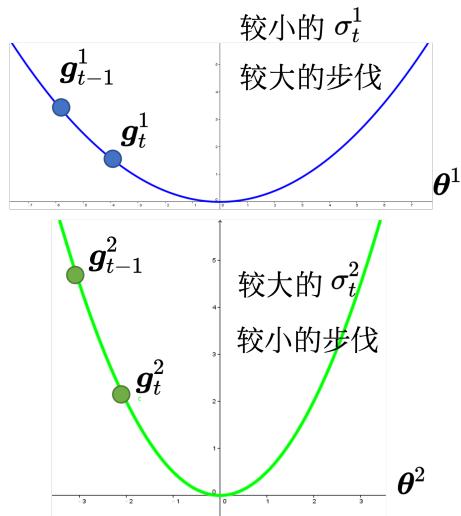


图 1.25 自动调整学习率示例

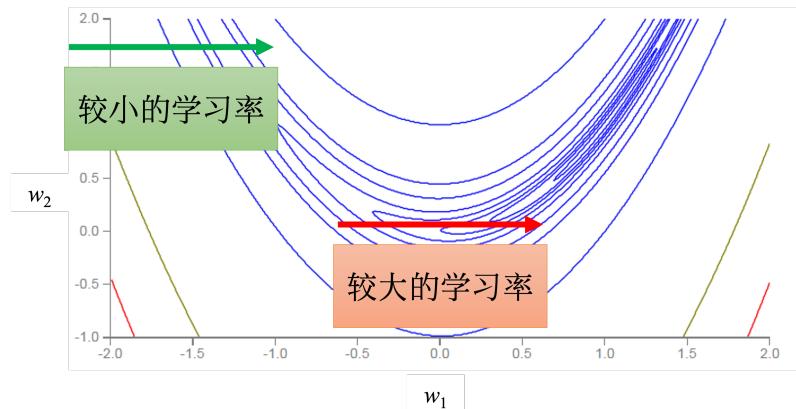


图 1.26 AdaGrad 的问题

RMSprop 没有论文，Geoff Hinton 在 Coursera 上开过深度学习的课程，他在他的课程里面讲了 RMSprop，如果要引用，需要引用对应视频的链接。

RMSprop 第一步跟 Apagrad 的方法是相同的，即

$$\sigma_1^i = \sqrt{\frac{1}{2} [(g_0^i)^2 + (g_1^i)^2]} \quad (1.23)$$

第二步更新过程为

$$\theta_2^i \leftarrow \theta_1^i - \frac{\eta}{\sigma_1^i} g_1^i \quad \sigma_1^i = \sqrt{\alpha (\sigma_0^i)^2 + (1 - \alpha) (g_1^i)^2} \quad (1.24)$$

其中  $0 < \alpha < 1$ ，其是一个可以调整的超参数。计算  $\theta_1^i$  的方法跟 AdaGrad 算均方根不一样，在算均方根的时候，每一个梯度都有同等的重要性，但在 RMSprop 里面，可以自己调整现在

的这个梯度的重要性。如果  $\alpha$  设很小趋近于 0，代表  $\mathbf{g}_1^i$  相较于之前算出来的梯度而言，比较重要；如果  $\alpha$  设很大趋近于 1，代表  $\mathbf{g}_1^i$  比较不重要，之前算出来的梯度比较重要。

同样的过程就反复继续下去，如式 (1.25) 所示。

$$\begin{aligned}\theta_3^i &\leftarrow \theta_2^i - \frac{\eta}{\sigma_2^i} \mathbf{g}_2^i \quad \sigma_2^i = \sqrt{\alpha (\sigma_1^i)^2 + (1 - \alpha) (\mathbf{g}_2^i)^2} \\ &\vdots \\ \theta_{t+1}^i &\leftarrow \theta_t^i - \frac{\eta}{\sigma_t^i} \mathbf{g}_t^i \quad \sigma_t^i = \sqrt{\alpha (\sigma_{t-1}^i)^2 + (1 - \alpha) (\mathbf{g}_t^i)^2}\end{aligned}\tag{1.25}$$

RMSProp 通过  $\alpha$  可以决定， $\mathbf{g}_t^i$  相较于之前存在  $\sigma_{t-1}^i$  里面的  $\mathbf{g}_1^i, \mathbf{g}_2^i, \dots, \mathbf{g}_{t-1}^i$  的重要性有多大。如果使用 RMSprop，就可以动态调整  $\sigma_t^i$  这一项。图 1.27 中黑线是误差表面，球就从 A 走到 B，AB 段的路很平坦， $\mathbf{g}$  很小，更新参数的时候，我们会走比较大的步伐。走动 BC 段后梯度变大了，AdaGrad 反应比较慢，而 RMSprop 会把  $\alpha$  设小一点，让新的、刚看到的梯度的影响比较大，很快地让  $\sigma_t^i$  的值变大，很快地让步伐变小，RMSprop 可以很快地“踩刹车”。如果走到 CD 段，CD 段是平坦的地方，可以调整  $\alpha$ ，让其比较看重最近算出来的梯度，梯度一变小， $\sigma_t^i$  的值就变小了，走的步伐就变大了。

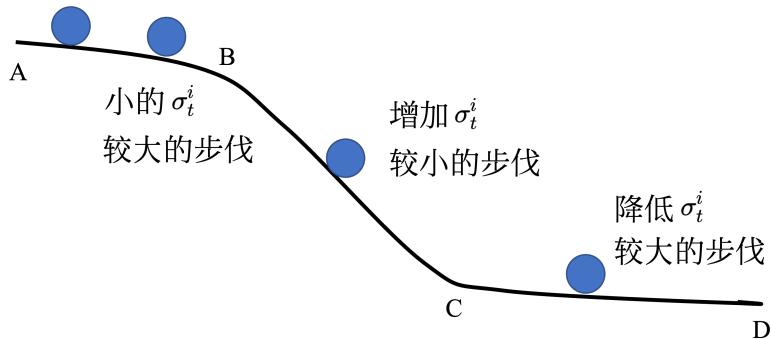


图 1.27 RMSprop 示例

### 1.3.3 Adam

最常用的优化的策略或者优化器 (optimizer) 是 Adam (Adaptive moment estimation)<sup>[1]</sup>。Adam 可以看作 RMSprop 加上动量，其使用动量作为参数更新方向，并且能够自适应调整学习率。PyTorch 里面已经写好了 Adam 优化器，这个优化器里面有一些超参数需要人为决定，但是往往用 PyTorch 预设的参数就足够好了。

## 1.4 学习率调度

如图 1.23 所示的简单的误差表面，我们都训练不起来，加上自适应学习率以后，使用 AdaGrad 方法优化的结果如图 1.28 所示。一开始优化的时候很顺利，在左转的时候，有 AdaGrad 以后，可以再继续走下去，走到非常接近终点的位置。走到 BC 段时，因为横轴方向的梯度很小，所以学习率会自动变大，步伐就可以变大，从而不断前进。接下来的问题走到图 1.28 中红圈的地方，快走到终点的时候突然“爆炸”了。 $\sigma_t^i$  是把过去所有的梯度拿来作平均。在 AB 段梯度很大，但在 BC 段，纵轴的方向梯度很小，因此纵轴方向累积了很小的  $\sigma_t^i$ ，累积到一定程度以后，步伐就变很大，但有办法修正回来。因为步伐很大，其会走到梯度比较大的地方。走到梯度比较大的地方后， $\sigma_t^i$  会慢慢变大，更新的步伐大小会慢慢变小，从而回到原来的路线。

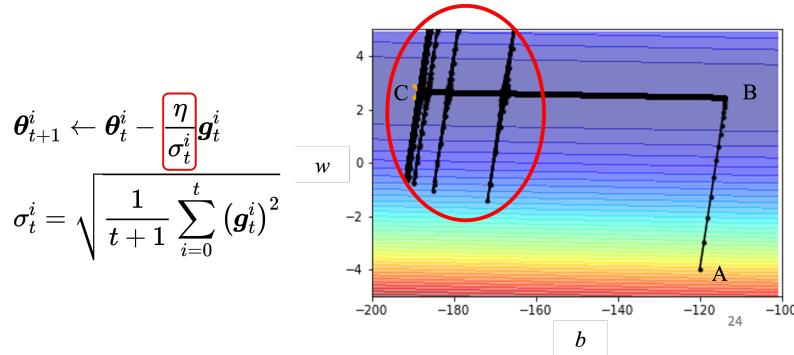


图 1.28 AdaGrad 优化的问题

通过学习率调度 (learning rate scheduling) 可以解决这个问题。之前的学习率调整方法中  $\eta$  是一个固定的值，而在学习率调度中  $\eta$  跟时间有关，如式 (1.26) 所示。学习率调度中最常见的策略是学习率衰减 (learning rate decay)，也称为学习率退火 (learning rate annealing)。随着参数的不断更新，让  $\eta$  越来越小，如图 1.29 所示。图 1.23b 的情况，如果加上学习率下降，可以很平顺地走到终点，如图 1.30 所示。在图 1.23b 红圈的地方，虽然步伐很大，但  $\eta$  变得非常小，步伐乘上  $\eta$  就变小了，就可以慢慢地走到终点。

$$\theta_{t+1}^i \leftarrow \theta_t^i - \frac{\eta_t}{\sigma_t^i} g_t^i \quad (1.26)$$

除了学习率下降以外，还有另外一个经典的学习率调度的方式——预热 (warmup)。预热的方法是让学习率先变大后变小，至于变到多大、变大的速度、变小的速度是超参数。残差

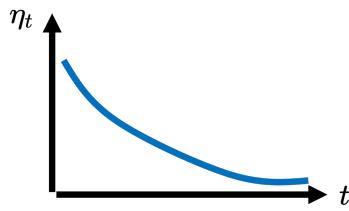


图 1.29 学习率衰减

网络<sup>[2]</sup>里面有预热的，在残差网络里面，学习率先设置成 0.01，再设置成 0.1，并且其论文还特别说明，一开始用 0.1 反而训练不好。除了残差网络，BERT 和 Transformer 的训练也都使用了预热。

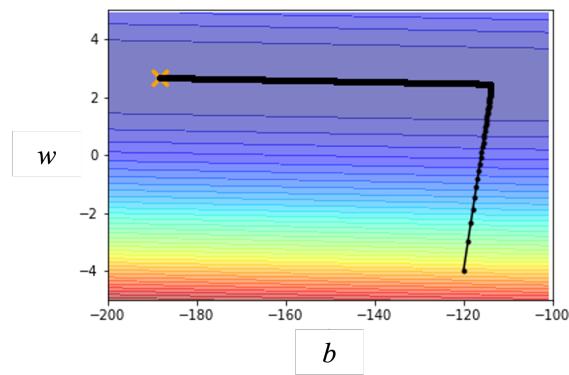


图 1.30 学习率衰减的优化效果

Q：为什么需要预热？

A：当我们使用 Adam、RMSprop 或 AdaGrad 时，需要计算  $\sigma$ 。而  $\sigma$  是一个统计的结果。从  $\sigma$  可知某一个方向的陡峭程度。统计的结果需要足够多的数据才精准，一开始统计结果  $\sigma$  是不精准的。一开始学习率比较小是用来探索收集一些有关误差表面的情报，先收集有关  $\sigma$  的统计数据，等  $\sigma$  统计得比较精准以后，再让学习率慢慢爬升。如果读者想要学更多有关预热的东西可参考 Adam 的进阶版——RAdam<sup>[3]</sup>。

## 1.5 优化总结

所以我们从最原始的梯度下降，进化到这一个版本，如式 (1.27) 所示。

$$\theta_{t+1}^i \leftarrow \theta_t^i - \frac{\eta_t}{\sigma_t^i} m_t^i \quad (1.27)$$

其中  $m_t^i$  是动量。

这个版本里面有动量，其不是顺着某个时刻算出的梯度方向来更新参数，而是把过去所有算出梯度的方向做一个加权总和当作更新的方向。接下来的步伐大小为  $\frac{m_t^i}{\sigma_t^i}$ 。最后通过  $\eta_t$  来实现学习率调度。这个是目前优化的完整的版本，这种优化器除了 Adam 以外，还有各种变形。但其实各种变形是使用不同的方式来计算  $m_t^i$  或  $\sigma_t^i$ ，或者是使用不同的学习率调度的方式。

Q：动量  $m_t^i$  考虑了过去所有的梯度，均方根  $\sigma_t^i$  考虑了过去所有的梯度，一个放在分子，一个放在分母，并且它们都考虑过去所有的梯度，不就是正好抵消了吗？

A： $m_t^i$  和  $\sigma_t^i$  在使用过去所有梯度的方式是不一样的，动量是直接把所有的梯度都加起来，所以它有考虑方向，它有考虑梯度的正负。但是均方根不考虑梯度的方向，只考虑梯度的大小，计算  $\sigma_t^i$  的时候，都要把梯度取一个平方项，把平方的结果加起来，所以只考虑梯度的大小，不考虑它的方向，所以动量跟  $\sigma_t^i$  计算出来的结果并不会互相抵消。

## 参考文献

- [1] KINGMA D P, BA J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.
- [2] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]// Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
- [3] LIU L, JIANG H, HE P, et al. On the variance of the adaptive learning rate and beyond [J]. arXiv preprint arXiv:1908.03265, 2019.

## 第 2 章 卷积神经网络

我们从卷积神经网络（Convolutional Neural Network，CNN）开始，探讨网络的架构设计。卷积神经网络是一种非常典型的网络架构，常用于图像分类等任务。通过卷积神经网络，我们可以知道网络架构如何设计，以及为什么合理的网络架构可以优化网络的表现。

所谓图像分类，就是给机器一张图像，由机器去判断这张图像里面有什么样的东西——是猫还是狗、是飞机还是汽车。怎么把图像当做模型的输入呢？对于机器，图像可以描述为三维张量（张量可以想成维度大于 2 的矩阵）。一张图像是一个三维的张量，其中一维代表图像的宽，另外一维代表图像的高，还有一维代表图像的通道（channel）的数目。

Q：什么是通道？

A：彩色图像的每个像素都可以描述为红色（red）、绿色（green）、蓝色（blue）的组合，这 3 种颜色就称为图像的 3 个色彩通道。这种颜色描述方式称为 RGB 色彩模型，常用于在屏幕上显示颜色。

网络的输入往往是向量，因此，将代表图像的三维张量“丢”到网络里之前，需要先将它“拉直”，如图 3.3 所示。在这个例子里面，张量有  $100 \times 100 \times 3$  个数字，所以一张图像是由  $100 \times 100 \times 3$  个数字所组成的，把这些数字排成一排就是一个巨大的向量。这个向量可以作为网络的输入，而这个向量里面每一维里面存的数值是某一个像素在某一个通道下的颜色强度。

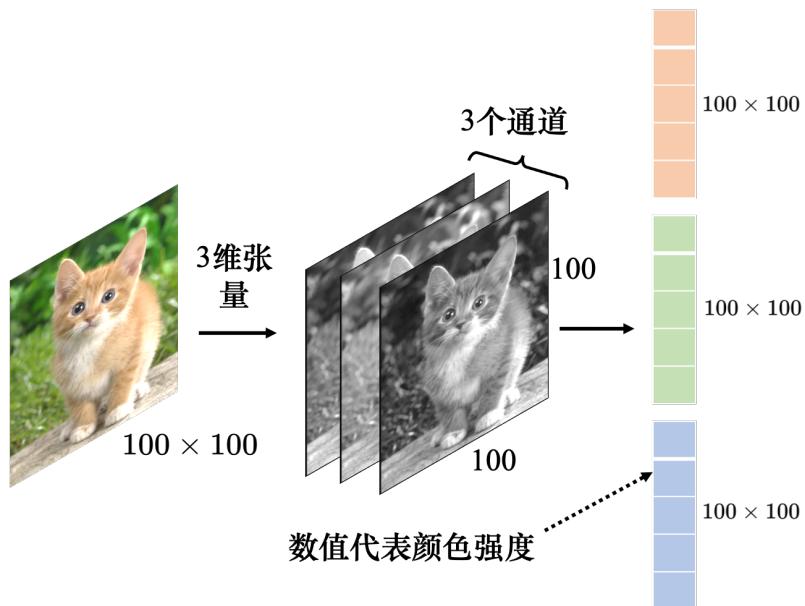


图 2.1 把图像作为模型输入

图像有大有小，而且不是所有图像尺寸都是一样的。常见的处理方式是把所有图像先调整成相同尺寸，再“丢”到图像的识别系统里面。以下的讨论中，默认模型输入的图像尺寸固定为 100 像素  $\times$  100 像素。

如图 3.4 所示，如果把向量当做全连接网络（fully connected network）的输入，输入的特征向量（feature vector）的长度就是  $100 \times 100 \times 3$ 。这是一个非常长的向量。由于每个神经元跟输入的向量中的每个数值都需要一个权重，所以当输入的向量长度是  $100 \times 100 \times 3$ ，且第 1 层有 1000 个神经元时，第 1 层的权重就需要  $1000 \times 100 \times 100 \times 3 = 3 \times 10^7$  个权重，这是一个非常巨大的数目。更多的参数为模型带来了更好的弹性和更强的能力，但也增加了过拟合（overfitting）的风险。模型的弹性越大，就越容易过拟合。为了避免过拟合，在做图像识别的时候，考虑到图像本身的特性，并不一定需要全连接，即不需要每个神经元跟输入的每个维度都有一个权重。接下来就是针对图像识别这个任务，对图像本身特性进行一些观察。

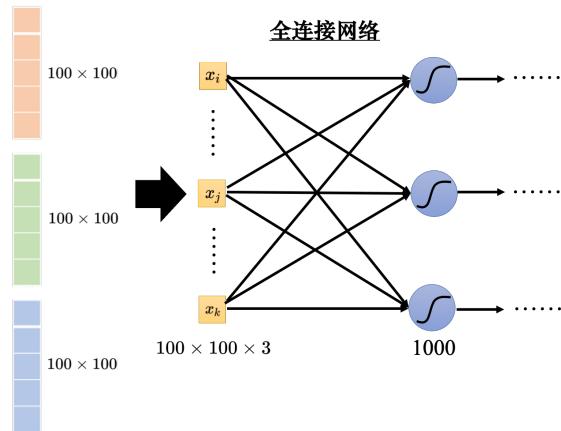


图 2.2 全连接网络

模型的输出应该是什么呢？模型的目标是分类，因此可将不同的分类结果表示成不同的独热向量  $y'$ 。在这个独热向量里面，类别对应的值为 1，其余类别对应的值为 0。例如，我们规定向量中的某些维度代表狗、猫、树等分类结果，那么若分类结果为猫，则猫所对应的维度的数值就是 1，其他东西所对应的维度的数值就是 0，如图 2.3 所示。独热向量  $y'$  的长度决定了模型可以识别出多少不同种类的东西。如果向量的长度是 5，代表模型可以识别出 5 种不同的东西。现在比较强的图像识别系统往往可以识别出 1000 种以上，甚至上万种不同的东西。如果希望图像识别系统可以识别上万种目标，标签就会是维度上万的独热向量。模型的输出通过 softmax 以后，输出是  $\hat{y}$ 。我们希望  $y'$  和  $\hat{y}$  的交叉熵（cross entropy）越小越好。

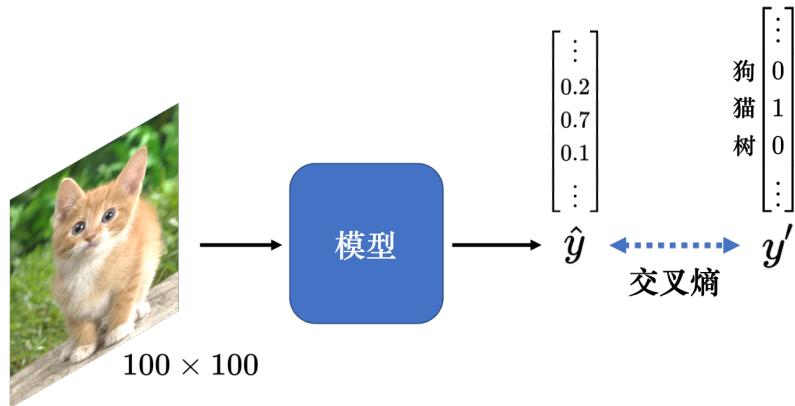


图 2.3 图像分类

## 2.1 观察 1：检测模式不需要整张图像

假设我们的任务是让网络识别出图像的动物。对一个图像识别的类神经网络里面的神经元而言，它要做的就是检测图像里面有没有出现一些特别重要的模式（pattern），这些模式是代表了某种物件的。举例来说，如果现在有三个神经元分别看到鸟嘴、眼睛、鸟爪 3 个模式，这就代表类神经网络看到了一只鸟，如图 3.5 所示。

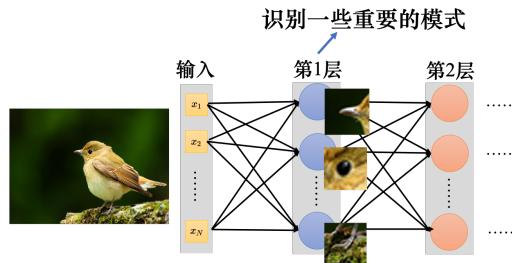


图 2.4 使用神经网络来检测模式

人在判断一个物件的时候，往往也是抓最重要的特征。看到这些特征以后，就会直觉地看到了某种物件。对于机器，也许这是一个有效的判断图像中物件的方法。但假设用神经元来判断某种模式是否出现，也许并不需要每个神经元都去看一张完整的图像。因为并不需要看整张完整的图像才能判断重要的模式（比如鸟嘴、眼睛、鸟爪）是否出现，如图 3.6 所示，要知道图像有没有一个鸟嘴，只要看非常小的范围。这些神经元不需要把整张图像当作输入，只需要把图像的一小部分当作输入，就足以让它们检测某些特别关键的模式是否出现，这是第 1 个观察。

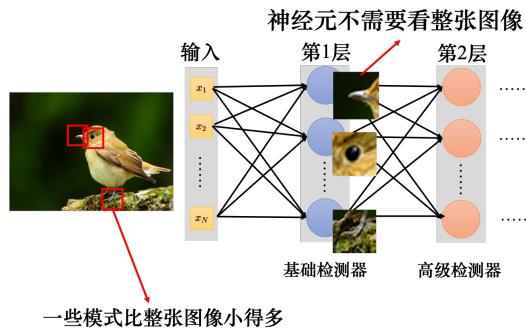


图 2.5 检测模式不需要整张图像

## 2.2 简化 1：感受野

根据观察 1 可以做第 1 个简化，卷积神经网络会设定一个区域叫做感受野（receptive field），每个神经元都只关心自己的感受野里面发生的事情，感受野是由我们自己决定的。举例来说，如图 3.7 所示，蓝色的神经元的守备范围就是红色正方体框的感受野。这个感受野里面有  $3 \times 3 \times 3$  个数值。对蓝色的神经元，它只需要关心这个小范围，不需要在意整张图像里面有什么东西，只在意它自己的感受野里面发生的事情就好。这个神经元会把  $3 \times 3 \times 3$  的数值“拉直”变成一个长度是  $3 \times 3 \times 3 = 27$  维的向量，再把这 27 维的向量作为神经元的输入，这个神经元会给 27 维的向量的每个维度一个权重，所以这个神经元有  $3 \times 3 \times 3 = 27$  个权重，再加上偏置（bias）得到输出。这个输出再送给下一层的神经元当作输入。

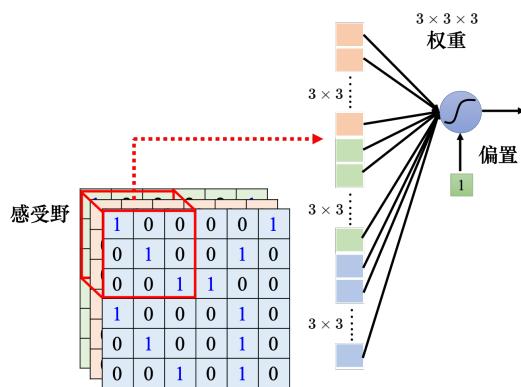


图 2.6 感受野

如图 3.8 所示，蓝色的神经元看左上角这个范围，这是它的感受野。黄色的神经元看右下角  $3 \times 3 \times 3$  的范围。图 3.8 中的一个正方形代表  $3 \times 3 \times 3$  的范围，右下角的正方形是黄色神经元的感受野。感受野彼此之间也可以是重叠的，比如绿色的神经元的感受野跟蓝色的、黄

色的神经元都有一些重叠的空间。我们没有办法检测所有的模式，所以同一个范围可以有多个不同的神经元，即多个神经元可以去守备同一个感受野。接下来我们讨论下如何设计感受野。

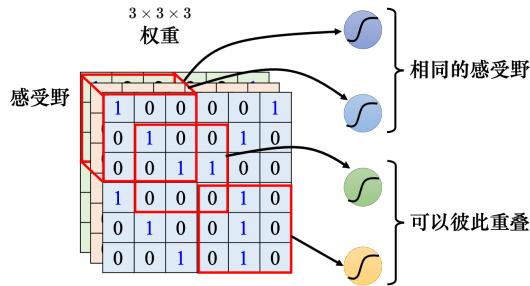


图 2.7 感受野彼此重叠

举例来说，感受野可以有大有小，因为模式有的比较小，有的比较大。有的模式也许在  $3 \times 3$  的范围内就可以被检测出来，有的模式也许要  $11 \times 11$  的范围才能被检测出来。此外，感受野可以只考虑某些通道。目前感受野是 RGB 三个通道都考虑，但也许有些模式只在红色或蓝色的通道会出现，即有的神经元可以只考虑一个通道。之后在讲到网络压缩的时候，会讲到这种网络的架构。感受野不仅可以是正方形的，例如刚才举的例子里面  $3 \times 3$ 、 $11 \times 11$ ，也可以是长方形的，完全可以根据对问题的理解来设计感受野。虽然感受野可以任意设计，但下面要跟大家讲一下最经典的感受野安排方式。

Q: 感受野一定要相连吗？

A: 感受野的范围不一定要相连，理论上可以有一个神经元的感受野就是图像的左上角跟右上角。但是就要想想为什么要这么做，会不会有什么模式也要看一个图像的左上角跟右下角才能够找到。如果没有，这种感受野就没什么用。要检测一个模式，这个模式就出现在整个图像里面的某一个位置，而不是分成好几部分，出现在图像里面的不同位置。所以通常的感受野都是相连的领地，但如果要设计很奇怪的感受野去解决很特别的问题，完全是可以的，这都是自己决定的。

一般在做图像识别的时候，可能不会觉得有些模式只出现在某一个通道里面，所以会看全部的通道。既然会看全部的通道，那么在描述一个感受野的时候，只要讲它的高跟宽，不用讲它的深度，因为它的深度就等于通道数，而高跟宽合起来叫做核大小。举例来说，如图 2.8 所示，核大小就是  $3 \times 3$ 。在图像识别里面，一般核大小不会设太大， $3 \times 3$  的核大小就足够了， $7 \times 7$ 、 $9 \times 9$  算是蛮大的核大小。如果核大小都是  $3 \times 3$ ，意味着我们认为在做图像识别的时候，重要的模式都只在  $3 \times 3$  这么小的范围内就可以被检测出来了。但有些模式也许很大，

也许  $3 \times 3$  的范围没办法检测出来，后面我们会再回答这个问题。常见的感受野设定方式就是核大小为  $3 \times 3$ 。

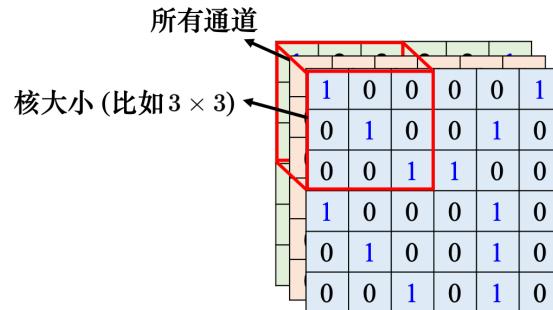


图 2.8 卷积核

一般同一个感受野会有一组神经元去守备这个范围，比如 64 个或者是 128 个神经元去守备一个感受野的范围。目前为止，讲的都是一个感受野，接下来介绍下各个不同感受野之间的关系。我们把左上角的感受野往右移一个步幅，就制造出一个新的守备范围，即新的感受野。移动的量叫做步幅 (stride)。图 2.9 中的这个例子里面，步幅就等于 2。步幅是一个超参数 (hyperparameter)，需要人为调整。因为希望感受野跟感受野之间是有重叠的，所以步幅往往不会设太大，一般设为 1 或 2。

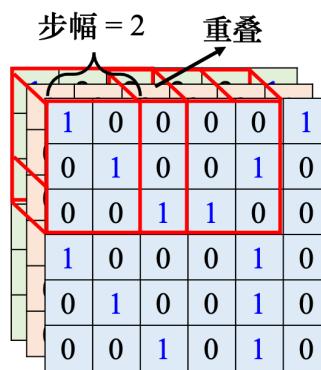


图 2.9 步幅

Q: 为什么希望感受野之间是有重叠的呢？

A: 因为假设感受野完全没有重叠，如果有一个模式正好出现在两个感受野的交界上面，就没有任何神经元去检测它，这个模式可能会丢失，所以希望感受野彼此之间有高度的重叠。如今步幅 = 2，感受野就会重叠。

接下来需要考虑一个问题：感受野超出了图像的范围，怎么办呢？如果不在超过图像的范围“摆”感受野，就没有神经元去检测出现在边界的模式，这样就会漏掉图像边界的地方，所以一般边界的地方也会考虑的。如图 2.10 所示，超出范围就做填充（padding），填充就是补值，一般使用零填充（zero padding），超出范围就补 0，如果感受野有一部分超出图像的范围之外，就当做那个里面的值都是 0。其实也有别的补值的方法，比如补整张图像里面所有值的平均值或者把边界的这些数字拿出来补没有值的地方。

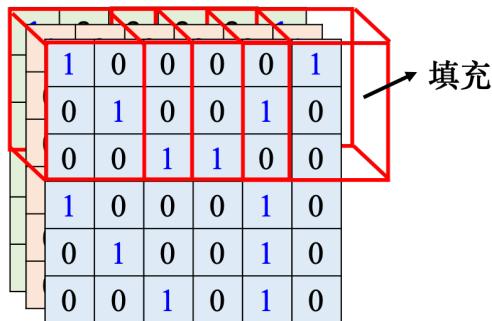


图 2.10 填充

除了水平方向的移动，也会有垂直方向上的移动，垂直方向步幅也是设 2，如图 2.11 所示。我们就按照这种方式扫过整张图像，所以整张图像里面每一寸土地都是有被某一个感受野覆盖的。也就是图像里面每个位置都有一群神经元在检测那个地方，有没有出现某些模式。这个是第 1 个简化。

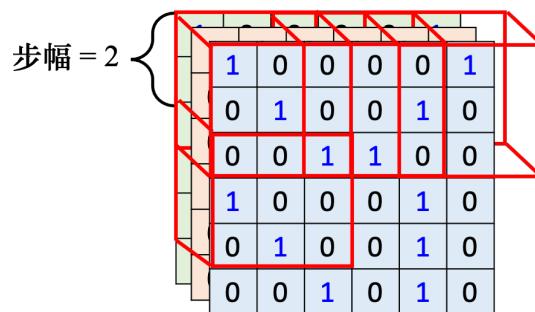


图 2.11 垂直移动

### 2.3 观察 2：同样的模式可能会出现在图像的不同区域

第 2 个观察是同样的模式，可能会出现在图像的不同区域。比如说模式鸟嘴，它可能出现在图像的左上角，也可能出现在图像的中间，同样的模式出现在图像的不同的位置也不是

太大的问题。如图 3.10 所示，因为出现在左上角的鸟嘴，它一定落在某一个感受野里面。因为感受野是盖满整个图像的，所以图像里面所有地方都在某个神经元的守备范围内。假设在某个感受野里面，有一个神经元的工作就是检测鸟嘴，鸟嘴就会被检测出来。所以就算鸟嘴出现在中间也没有关系。假设其中有一个神经元可以检测鸟嘴，鸟嘴出现在图像的中间也会被检测出来。但这些检测鸟嘴的神经元做的事情是一样的，只是它们守备的范围不一样。既然如此，其实没必要每个守备范围都去放一个检测鸟嘴的神经元。如果不同的守备范围都要有一个检测鸟嘴的神经元，参数量会太多了，因此需要做出相应的简化。

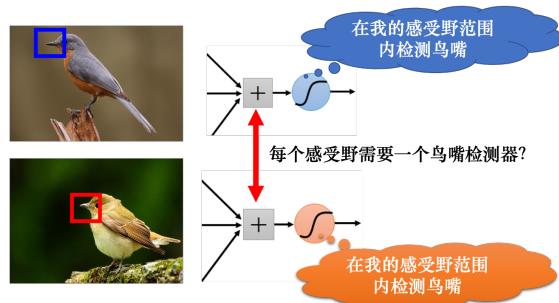


图 2.12 每个感受野都放一个鸟嘴检测器

## 2.4 简化 2：共享参数

在提出简化技巧前，我们先举个类似的例子。这个概念就类似于教务处希望可以推大型的课程一样，假设每个院系都需要深度学习相关的课程，没必要在每个院系都开机器学习的课程，可以开一个比较大型的课程，让所有院系的人都可以修课。如果放在图像处理上，则可以让不同感受野的神经元共享参数，也就是做参数共享 (parameter sharing)，如图 3.11 所示。所谓参数共享就是两个神经元的权重完全是一样的。

如图 3.12 所示，颜色相同，权重完全是一样的，比如上面神经元的第一个权重是  $w_1$ ，下面神经元的第一个权重也是  $w_1$ ，它们是同一个权重，用同一种颜色黄色来表示。上面神经元跟下面神经元守备的感受野是不一样的，但是它们的参数是相同的。虽然两个神经元的参数是一模一样，但它们的输出不会永远都是一样的，因为它们的输入是不一样的，它们照顾的范围是不一样的。上面神经元的输入是  $x_1, x_2, \dots$ ，下面神经元的输入是  $x'_1, x'_2, \dots$ 。上面神经元的输出为

$$\sigma(w_1x_1 + w_2x_2 + \dots + 1) \quad (2.1)$$

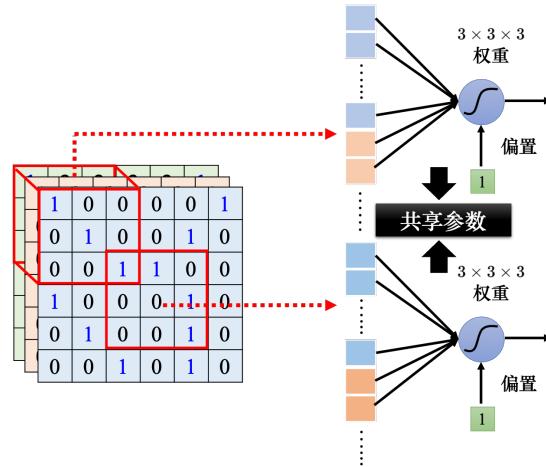


图 2.13 共享参数

下面神经元的输出为

$$\sigma(w_1x'_1 + w_2x'_2 + \dots + 1) \quad (2.2)$$

因为输入不一样的关系，所以就算是两个神经元共用参数，它们的输出也不会是一样的。所以这是第 2 个简化，让一些神经元可以共享参数，共享的方式完全可以自己决定。接下来将介绍图像识别方面，常见的共享方法是如何设定的。

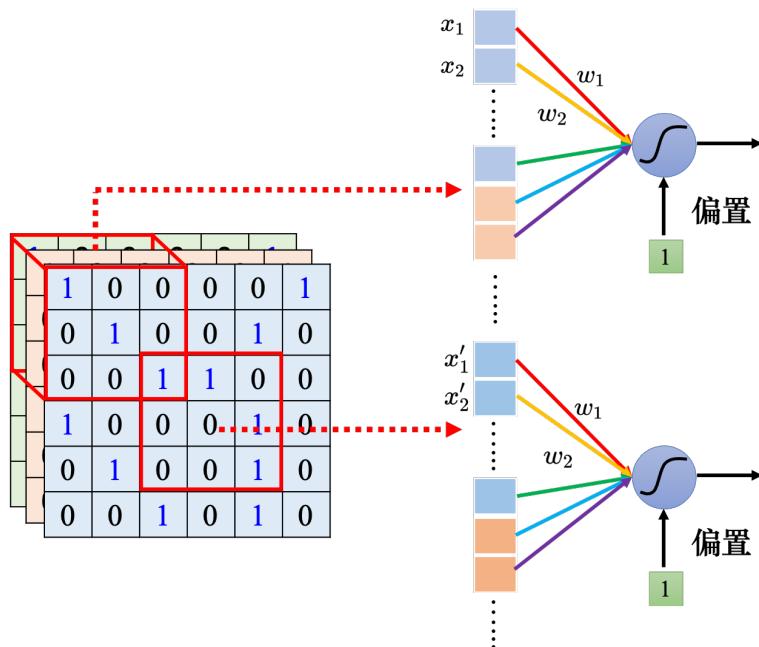


图 2.14 两个神经元共享参数

如图 3.13 所示，每个感受野都有一组神经元在负责守备，比如 64 个神经元，它们彼此

之间可以共享参数。图 3.14 中使用一样的颜色代表这两个神经元共享一样的参数，所以每个感受野都只有一组参数，就是上面感受野的第 1 个神经元会跟下面感受野的第 1 个神经元共用参数，上面感受野的第 2 个神经元跟下面感受野的第 2 个神经元共用参数……所以每个感受野都只有一组参数而已，这些参数称为滤波器（filter）。这是第 2 个简化的方法。

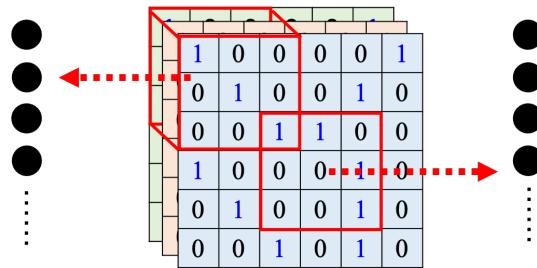


图 2.15 守备感受野的神经元

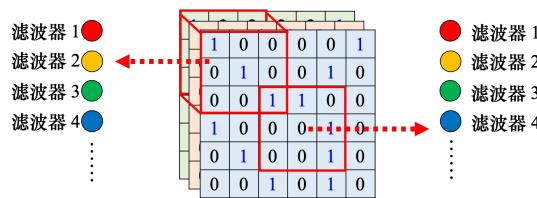


图 2.16 多个神经元共享参数

## 2.5 简化 1 和 2 的总结

目前已经讲了两个简化的办法，我们来总结下。如图 3.15 所示，全连接网络是弹性最大的。全连接网络可以决定它看整张图像还是只看一个范围，如果它只想看一个范围，可以把很多权重设成 0。全连接层（fully connected layer）可以自己决定看整张图像还是一个小范围。但加上感受野的概念以后，只能看一个小范围，网络的弹性是变小的。参数共享又进一步限制了网络的弹性。本来在学习的时候，每个神经元可以各自有不同的参数，它们可以学出相同的参数，也可以有不一样的参数。但是加入参数共享以后，某一些神经元无论如何参数都要一模一样的，这又增加了对神经元的限制。而感受野加上参数共享就是卷积层（convolutional layer），用到卷积层的网络就叫卷积神经网络。卷积神经网络的偏差比较大。但模型偏差大不一定是坏事，因为当模型偏差大，模型的灵活度（flexibility）较低时，比较不容易过拟合。全连接层可以做各式各样的事情，它可以有各式各样的变化，但它可能没有办法在任何特定的任务上做好。而卷积层是专门为图像设计的，感受野、参数共享都是为图像设计的。虽然卷积

神经网络模型偏差很大，但用在图像上不是问题。如果把它用在图像之外的任务，就要仔细想想这些任务有没有图像用的特性。

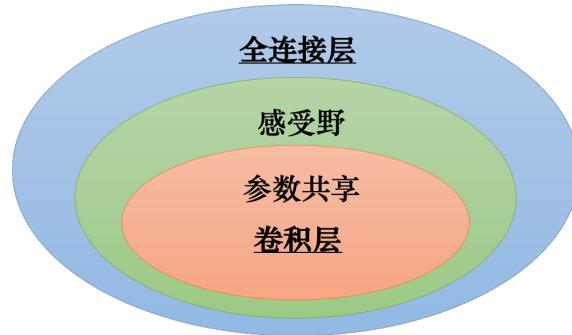


图 2.17 卷积层与全连接层的关系

接下来通过第 2 个版本的故事来说明卷积神经网络。如图 3.16 所示，卷积层里面有很多滤波器，这些滤波器的大小是  $3 \times 3 \times$  通道。如果图像是彩色的，它有 RGB 三个通道。如果是黑白的图像，它的通道就等于 1。一个卷积层里面就是有一排的滤波器，每个滤波器都是一个  $3 \times 3 \times$  通道，其作用是要去图像里面检测某个模式。这些模式要在  $3 \times 3 \times$  通道，这个小的范围内，它才能够被这些滤波器检测出来。举个例子，假设通道为 1，也就是图像是黑白的。滤波器就是一个一个的张量，这些张量里面的数值就是模型里面的参数。这些滤波器里面的数值其实是未知的，它是可以通过学习找出来的。假设这些滤波器里面的数值已经找出来了，如图 3.17 所示。如图 3.18 所示，这是一个  $6 \times 6$  的大小的图像。先把滤波器放在图像的左上角，接着把滤波器里面所有的 9 个值跟左上角这个范围内的 9 个值对应相乘再相加，也就是做内积，结果是 3。接下来设置好步幅，然后把滤波器往右移或往下移，重复几次，可得到模式检测的结果，图 3.18 中的步幅为 1。使用滤波器 1 检测模式时，如果出现图像  $3 \times 3$  范围内对角线都是 1 这种模式的时候，输出的数值会最大。输出里面左上角和左下角的值最大，所以左上角和左下角有出现对角线都是 1 的模式，这是第 1 个滤波器。

如图 3.19 所示，接下来把每个滤波器都做重复的过程。比如说有第 2 个滤波器，它用来检测图像  $3 \times 3$  范围内中间一列都为 1 的模式。把第 2 个滤波器先从左上角开始扫起，得到一个数值，往右移一个步幅，再得到一个数值再往右移一个步幅，再得到一个数值。重复同样的操作，直到把整张图像都扫完，就得到另外一组数值。每个滤波器都会给我们一组数字，红色的滤波器给我们一组数字，蓝色的滤波器给我们另外一组数字。如果有 64 个滤波器，就可以得到 64 组的数字。这组数字称为特征映射 (feature map)。当一张图像通过一个卷积层里

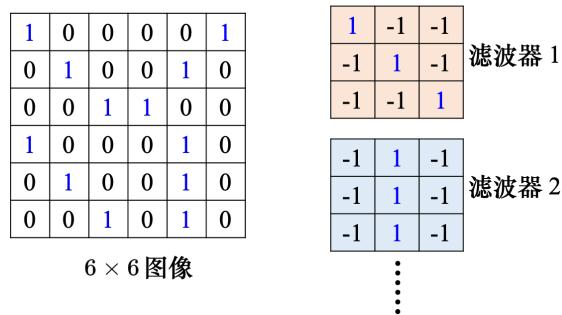
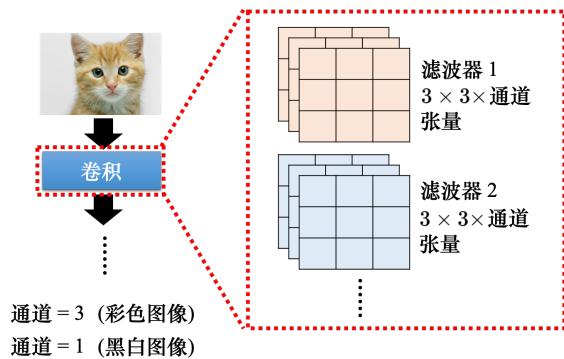


图 2.19 滤波器示例

面一堆滤波器的时候，就会产生一个特征映射。假设卷积层里面有 64 个滤波器，产生的特征映射就有 64 组数字。在上述例子中每一组是  $4 \times 4$ ，即第 1 个滤波器产生  $4 \times 4$  个数字，第 2 个滤波器也产生  $4 \times 4$  个数字，第 3 个也产生  $4 \times 4$  个数字，64 个滤波器都产生  $4 \times 4$  个数字。特征映射可以看成是另外一张新的图像，只是这个图像的通道不是 RGB 3 个通道，有 64 个通道，每个通道就对应到一个滤波器。本来一张图像有 3 个通道，通过一个卷积变成一张新的有 64 个通道图像。

卷积层是可以叠很多层的，如图 3.20 所示，第 2 层的卷积里面也有一堆的滤波器，每个滤波器的大小设成  $3 \times 3$ 。其高度必须设为 64，因为滤波器的高度就是它要处理的图像的通道。如果输入的图像是黑白的，通道是 1，滤波器的高度就是 1。如果输入的图像是彩色的，通道为 3，滤波器的高度就是 3。对于第 2 个卷积层，它的输入也是一张图像，这个图像的通道是 64。这个 64 是前一个卷积层的滤波器数目，前一个卷积层的滤波器数目是 64，输出以后就是 64 个通道。所以如果第 2 层想要把这个图像当做输入，滤波器的高度必须是 64。所以第 2 层也有一组滤波器，只是这组滤波器的高度是 64。

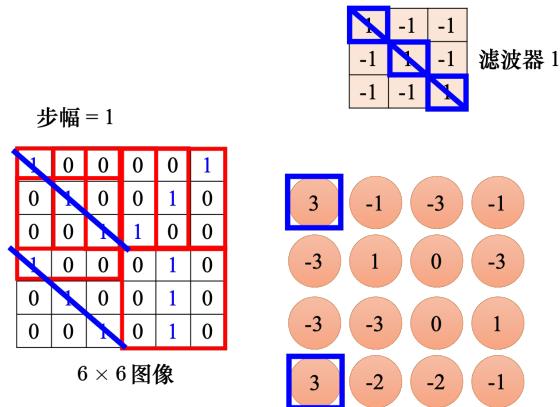


图 2.20 使用滤波器检测模式

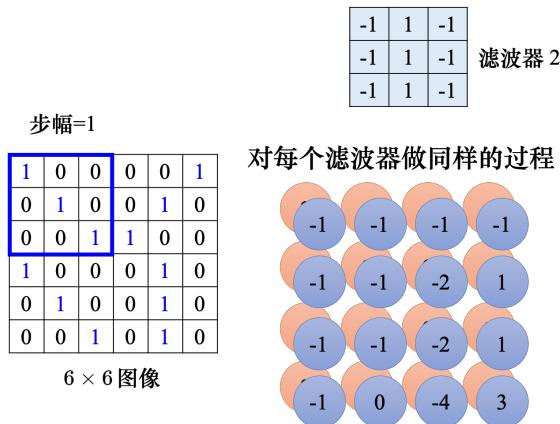


图 2.21 使用多个滤波器检测模式

Q：如果滤波器的大小一直设  $3 \times 3$ ，会不会让网络没有办法看比较大范围的模式呢？

A：不会。如图 3.21 所示，如果在第 2 层卷积层滤波器的大小一样设  $3 \times 3$ ，当我们看第 1 个卷积层输出的特征映射的  $3 \times 3$  的范围的时候，在原来的图像上是考虑了一个  $5 \times 5$  的范围。虽然滤波器只有  $3 \times 3$ ，但它在图像上考虑的范围是比较大的是  $5 \times 5$ 。因此网络叠得越深，同样是  $3 \times 3$  的大小的滤波器，它看的范围就会越来越大。所以网络够深，不用怕检测不到比较大的模式。

刚才讲了两个版本的故事，这两个版本的故事是一模一样的。第 1 个版本的故事里面说到了有一些神经元，这些神经元会共用参数，这些共用的参数就是第 2 个版本的故事里面的滤波器。如图 3.22 所示，这组参数有  $3 \times 3 \times 3$  个，即滤波器里面有  $3 \times 3 \times 3$  个数字，这边特别还用颜色把这些数字圈起来，权重就是这些数字。为了简化，这边去掉了偏置。神经元是

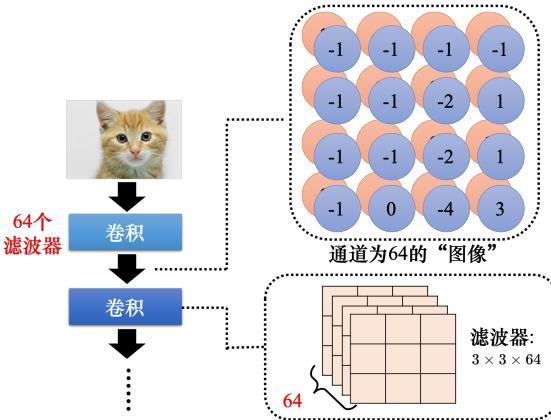


图 2.22 对图像进行卷积

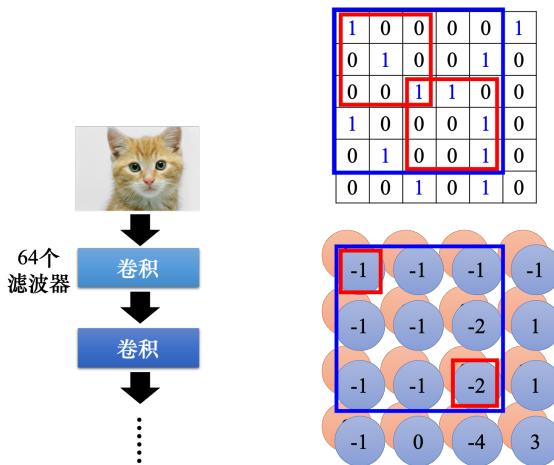


图 2.23 网络越深，可以检测的模式越大

有偏置的，滤波器也是有偏置的。在一般的实践上，卷积神经网络的滤波器都是有偏置的。

如图 3.23 所示，在第 1 个版本的故事里面，不同的神经元可以共享权重，去守备不同的范围。而共享权重其实就是用滤波器扫过一张图像，这个过程就是卷积。这就是卷积层名字的由来。把滤波器扫过图像就相当于不同的感受野神经元可以共用参数，这组共用的参数就叫做一个滤波器。

## 2.6 观察 3：下采样不影响模式检测

第 3 个观察是下采样不影响模式检测。把一张比较大的图像做下采样 (subsampling)，把图像偶数的列都拿掉，奇数的行都拿掉，图像变成为原来的  $1/4$ ，但是不会影响里面是什么东西。举例来说，如图 2.26 所示，把一张大的鸟的图像缩小，这张小的图像还是一只鸟。

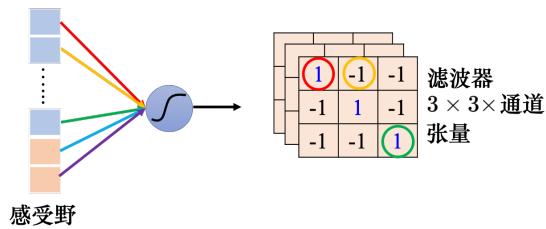


图 2.24 共享参数示例

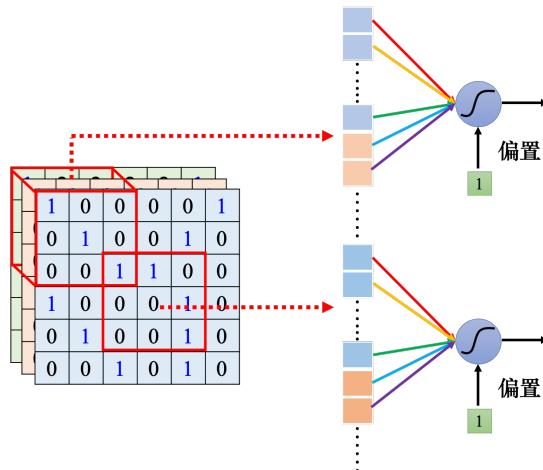


图 2.25 从不同的角度理解参数共享

## 2.7 简化 3：汇聚

根据第 3 个观察，汇聚被用到了图像识别中。汇聚没有参数，所以它不是一个层，它里面没有权重，它没有要学习的东西，汇聚比较像 Sigmoid、ReLU 等激活函数 (activation function)。因为它里面是没有要学习的参数的，它就是一个操作符 (operator)，其行为都是固定好的，不需要根据数据学任何东西。每个滤波器都产生一组数字，要做汇聚的时候，把这些数字分组，可以  $2 \times 2$  个一组， $3 \times 3$ 、 $4 \times 4$  也可以，这个是我们自己决定的，图 2.27 中的例子是  $2 \times 2$  个一组。汇聚有很多不同的版本，以最大汇聚 (max pooling) 为例。最大汇聚在每一组里面选



图 2.26 下采样示意

一个代表，选的代表就是最大的一个，如图 2.28 所示。除了最大汇聚，还有平均汇聚（mean pooling），平均汇聚是取每一组的平均值。

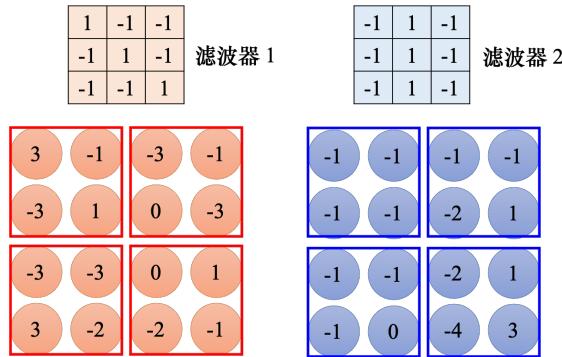


图 2.27 最大汇聚示例

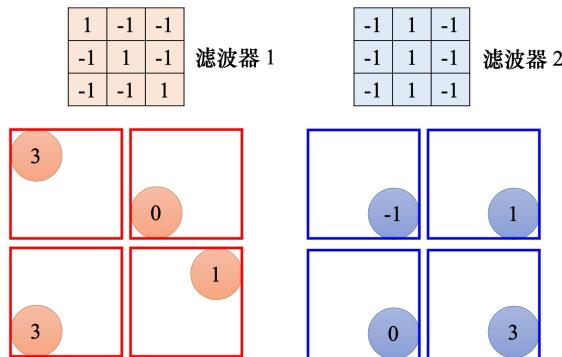


图 2.28 最大汇聚结果

做完卷积以后，往往后面还会搭配汇聚。汇聚就是把图像变小。做完卷积以后会得到一张图像，这张图像里面有很多的通道。做完汇聚以后，这张图像的通道不变。如图 3.27 所示，在刚才的例子里面，本来  $4 \times 4$  的图像，如果把这个输出的数值  $2 \times 2$  个一组， $4 \times 4$  的图像就会变成  $2 \times 2$  的图像，这就是汇聚做的事情。一般在实践上，往往就是卷积跟汇聚交替使用，可能做几次卷积，做一次汇聚。比如两次卷积，一次汇聚。不过汇聚对于模型的性能 (performance) 可能会带来一点伤害。假设要检测的是非常微细的东西，随便做下采样，性能可能会稍微差一点。所以近年来图像的网络的设计往往也开始把汇聚丢掉，它会做这种全卷积的神经网络，整个网络里面都是卷积，完全都不用汇聚。汇聚最主要的作用是减少运算量，通过下采样把图像变小，从而减少运算量。随着近年来运算能力越来越强，如果运算资源足够支撑不做汇聚，很多网络的架构的设计往往就不做汇聚，而是使用全卷积，卷积从头到尾，看看做不做得起来，看看能不能做得更好。

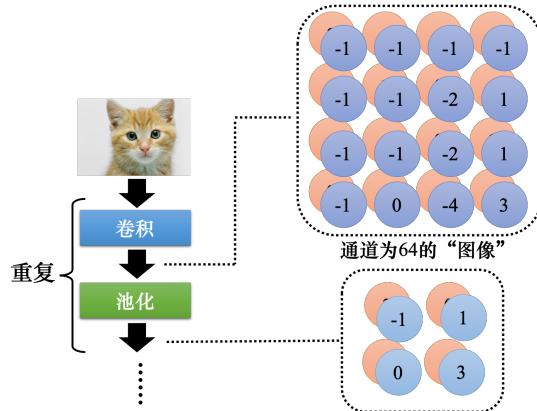


图 2.29 重复使用卷积和池化

一般架构就是卷积加汇聚，汇聚是可有可无的，很多人可能会选择不用汇聚。如图 3.28 所示，如果做完几次卷积和汇聚以后，把汇聚的输出做扁平化 (flatten)，再把这个向量丢进全连接层里面，最终还要过个 softmax 来得到图像识别的结果。这就是一个经典的图像识别的网络，里面有卷积、汇聚和扁平化，最后再通过几个全连接层或 softmax 来得到图像识别的结果。

扁平化就是把图像里面本来排成矩阵样子的东西“拉直”，即把所有的数值“拉直”变成一个向量。

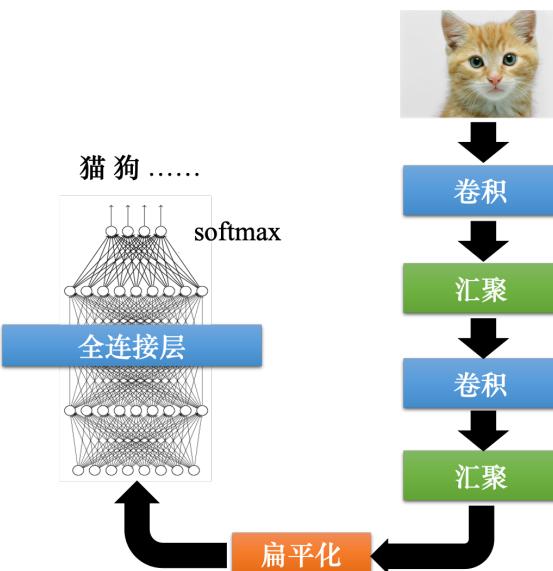


图 2.30 经典的图像识别的网络

## 2.8 卷积神经网络的应用：下围棋

除了图像识别以外，卷积神经网络另外一个最常见的应用是用来下围棋，以 AlphaGo 为例。下围棋其实是一个分类的问题，网络的输入是棋盘上黑子跟白子的位置，输出就是下一步应该要落子的位置。网络的输入是一个向量，棋盘上有  $19 \times 19$  个位置，可以把一个棋盘表示成一个  $19 \times 19$  维的向量。在这个向量里面，如果某个位置有一个黑子，这个位置就填 1，如果有白子，就填 -1，如果没有子，就填 0。不一定要黑子是 1，白子是 -1，没有子就是 0，这只是一个可能的表示方式。通过把棋盘表示成向量，网络就可以知道棋盘上的盘势。把这个向量输到一个网络里面，下围棋就可以看成一个分类的问题，通过网络去预测下一步应该落子的最佳位置，所以下围棋就是一个有  $19 \times 19$  个类别的分类问题，网络会输出  $19 \times 19$  个类别中的最好类别，据此选择下一步落子的位置。这个问题可以用一个全连接网络来解决，但用卷积神经网络的效果更好。

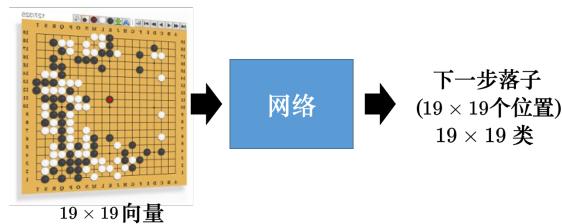


图 2.31 使用卷积神经网络下围棋

Q: 为什么卷积神经网络可以用在下围棋上?

A: 首先一个棋盘可以看作是一个分辨率  $19 \times 19$  的图像。一般图像很大,  $100 \times 100$  的分辨率的图像, 都是很小的图像了。但是棋盘是一个更小的图像, 其分辨率只有  $19 \times 19$ 。这个图像里面每个像素代表棋盘上一个可以落子的位置。一般图像的通道就是 RGB。而在 AlphaGo 的原始论文里面, 每个棋盘的位置, 即每个棋盘上的像素是用 48 个通道来描述, 即棋盘上的每个位置都用 48 个数字来描述那个位置发生的事情<sup>[1]</sup>。48 个数字是围棋高手设计出来的, 包括比如这个位置是不是要被叫吃了, 这个位置旁边有没有颜色不一样的等等。所以当我们用 48 个数字来描述棋盘上的一个位置时, 这个棋盘就是  $19 \times 19$  的分辨率的图像, 其通道就是 48。卷积神经网络其实并不是随便用都会好的, 它是为图像设计的。如果一个问题跟图像没有共通的特性, 就不该用卷积神经网络。既然下围棋可以用卷积神经网络, 这意味着围棋跟图像有共同的特性。图像上的第 1 个观察是, 只需要看小范围就可以知道很多重要的模式。下围棋也是一样的, 图 3.29 中的模式不用看整个棋盘的盘势, 就知道发生了什么事 (白子被黑子围住了)。接下来黑子如果放在被围住的白子下面, 就可以把白子提走。白子如果放在白子下面, 被围住的白子才不会被提走。其实 AlphaGo 的第 1 层的滤波器大小就是  $5 \times 5$ , 所以显然设计这个网络的人觉得棋盘上很多重要的模式, 也许看  $5 \times 5$  的范围就可以知道。此外, 图像上的第 2 个观察是同样的模式可能会出现在不同的位置, 在下围棋里面也是一样的。如图 3.30 所示, 这个叫吃的模式, 它可以出现在棋盘上的任何位置, 它可以出现在左上角, 也可以出现在右下角, 所以从这个观点来看图像跟下围棋有很多共同之处。

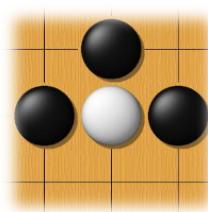


图 2.32 围棋的模式

在做图像的时候都会做汇聚, 一张图像做下采样以后, 并不会影响我们对图像中物件的判断。但汇聚对于下围棋这种精细的任务并不实用, 下围棋时随便拿掉一个列拿掉一个行, 整个棋局就不一样。AlphaGo 在 Nature 上的论文正文里面没有提它用的网络架构, 而是在附

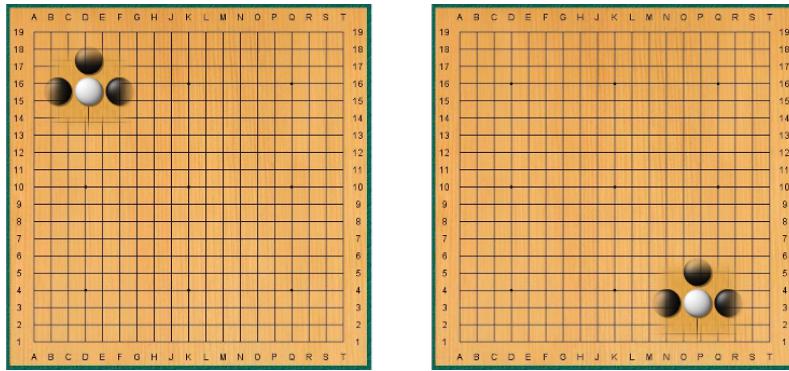


图 2.33 吃的模式

件中介绍了这个细节。AlphaGo 把一个棋盘看作  $19 \times 19 \times 48$  大小的图像。接下来它有做零填充。它的滤波器的大小是  $5 \times 5$ , 然后有  $k = 192$  个滤波器,  $k$  的值是试出来的, 它也试了 128、256, 发现 192 的效果最好。这是第 1 层, 步幅 =1, 使用了 ReLU。在第 2 层到第 12 层都有做零填充。核大小都是  $3 \times 3$ , 一样是  $k$  个滤波器, 也就是每一层都是 192 个滤波器, 步幅一样设 1, 这样叠了很多层以后, 因为是一个分类的问题, 最后加上了一个 softmax, 没有用汇聚, 所以这是一个很好的设计类神经网络的例子。在下围棋的时候不适合用汇聚。所以我们要想清楚, 在用一个网络架构的时候, 这个网络的架构到底代表什么意思, 它适不适合用在这个任务上。卷积神经网络除了下围棋、图像识别以外, 近年来也用在语音上和文字处理上。比如论文 “Convolutional Neural Networks for Speech Recognition”<sup>[2]</sup> 将卷积神经网络应用到语音上, 论文 “UNITN: Training Deep Convolutional Neural Network for Twitter Sentiment Classification”<sup>[3]</sup> 把卷积神经网络应用到文字处理上。如果想把卷积神经网络用在语音和文字处理上, 就要对感受野和参数共享进行重新设计, 其跟图像不同, 要考虑语音跟文字的特性来设计。所以不要以为在图像上的卷积神经网络, 直接套到语音上它也奏效 (work), 可能是不奏效的。要想清楚图像语音有什么样的特性, 要怎么设计合适感受野。

其实卷积神经网络不能处理图像放大缩小或者是旋转的问题, 假设给卷积神经网络看的狗的图像大小都相同, 它可以识别这是一只狗。当把这个图像放大的时候, 它可能就不能识别这张图像是一只狗。卷积神经网络就是这么 “笨”, 对它来说, 这是两张图像。虽然两张图像的形状是一模一样的, 但是如果把它们 “拉直” 成向量, 里面的数值就是不一样的。虽然人眼一看觉得两张图像的形状很像, 但对卷积神经网络来说它们是非常不一样的。所以事实上, 卷积神经网络并不能够处理图像放大缩小或者是旋转的问题。假设图像里面的物件都是比较小的, 当卷积神经网络在某种大小的图像上面学会做图像识别, 我们把物件放大, 它的性能就会

降低不少，卷积神经网络并没有想像的那么强。因此在做图像识别的时候往往都要做数据增强 (data augmentation)。所谓数据增强就是把训练数据每张图像里面截一小块出来放大，让卷积神经网络看过不同大小的模式；把图像旋转，让它看过某一个物件旋转以后长什么样子，卷积神经网络才会做到好的结果。卷积神经网络不能够处理缩放 (scaling) 跟旋转 (rotation) 的问题，但 Special Transformer Layer 网络架构可以处理这个问题。

## 参考文献

- [1] SILVER D, HUANG A, MADDISON C J, et al. Mastering the game of go with deep neural networks and tree search[J]. nature, 2016, 529(7587): 484-489.
- [2] ABDEL-HAMID O, MOHAMED A R, JIANG H, et al. Convolutional neural networks for speech recognition[J]. IEEE/ACM Transactions on audio, speech, and language processing, 2014, 22(10): 1533-1545.
- [3] SEVERYN A, MOSCHITTI A. Unitn: Training deep convolutional neural network for twitter sentiment classification[C]//Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015). 2015: 464-469.

## 第 3 章 自注意力机制

讲完了卷积神经网络 (Convolutional Neural Network, CNN) 以后，我们要讲另外一个常见的网络架构——**自注意力模型 (self-attention model)**。目前为止，不管是在预测 YouTube 观看人数的问题上，还是图像处理上，网络的输入都是一个向量。如图 3.1 所示，输入可以看作是一个向量，如果是回归问题，输出是一个标量，如果是分类问题，输出是一个类别。

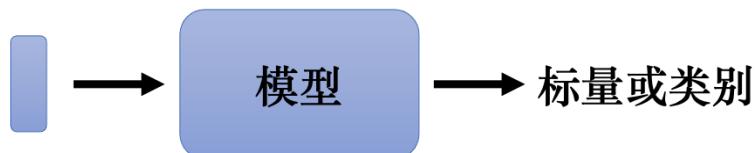


图 3.1 输入是一个向量

### 3.1 输入是向量序列的情况

在图像识别的时候，假设输入的图像大小都是一样的。但如果问题变得复杂，如图 3.2 所示，输入是一组向量，并且输入的向量的数量是会改变的，即每次模型输入的序列长度都不一样，这个时候应该要怎么处理呢？我们通过具体的例子来讲解处理方法。



图 3.2 输入是一组向量

第一个例子是文字处理，假设网络的输入是一个句子，每一个句子的长度都不一样（每个句子里面词汇的数量都不一样）。如果把一个句子里面的每一个词汇都描述成一个向量，用向量来表示，模型的输入就是一个向量序列，而且该向量序列的大小每次都不一样（句子的长度不一样，向量序列的大小就不一样）。

将词汇表示成向量最简单的做法是独热编码 (one-hot encoding)，创建一个很长的向量，该向量的长度跟世界上存在的词汇的数量是一样多的。假设英文是十万个词汇，我们就创建一个十万维的向量，每一个维度对应到一个词汇，如式 (3.1) 所示。但是这种表示方法有一个非常严重的问题，它假设所有的词汇彼此之间都是没有关系的。cat 和 dog 都是动物，它们应该比较像；cat 是动物，apple 是植物，它们应该比较不像。但从独热向量中不能看到这件事

情，其里面没有任何语义的信息。

$$\begin{aligned}
 \text{apple} &= [1, 0, 0, 0, 0, \dots] \\
 \text{bag} &= [0, 1, 0, 0, 0, \dots] \\
 \text{cat} &= [0, 0, 1, 0, 0, \dots] \\
 \text{dog} &= [0, 0, 0, 1, 0, \dots] \\
 \text{elephant} &= [0, 0, 0, 0, 1, \dots]
 \end{aligned} \tag{3.1}$$

除了独热编码，词嵌入（word embedding）也可将词汇表示成向量。词嵌入使用一个向量来表示一个词汇，而这个向量是包含语义信息的。如图 3.3 所示，如果把词嵌入画出来，所有的动物可能聚集成一团，所有的植物可能聚集成一团，所有的动词可能聚集成一团等等。词嵌入会给每一个词汇一个向量，而一个句子就是一组长度不一的向量。

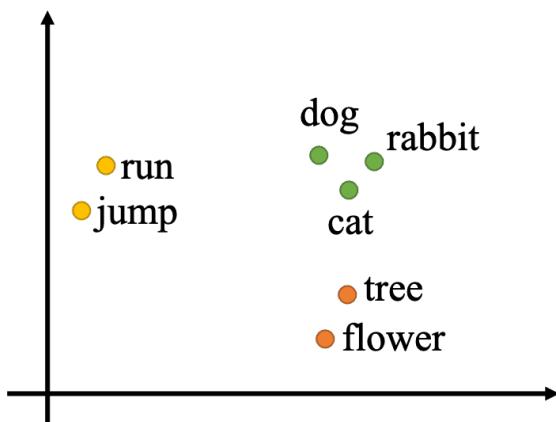


图 3.3 词嵌入

接下来举一些把一个向量的序列当做输入的例子。如图 3.4 所示，一段声音信号其实是一组向量。我们会把一段声音信号取一个范围，这个范围叫做一个窗口（window），把该窗口里面的信息描述成一个向量，这个向量称为一帧（frame）。通常这个窗口的长度就是 25 毫秒。为了要描述一整段的声音信号，我们会把这个窗口往右移一点，通常移动的大小是 10 毫秒。

**Q:** 为什么窗口的长度是 25 毫秒，窗口移动的大小是 10 毫秒？

**A:** 前人帮我们调好了，他们已经把所有的可能都试过了，调一个最好的结果。

总之，一段声音信号就是用一串向量来表示，而因为每一个窗口，他们往右移都是移动 10 毫秒，所以一秒钟的声音信号有 100 个向量，所以一分钟的声音信号就有这个 100 乘以

60，就有 6000 个向量。所以语音其实很复杂的。一小段的声音信号，它里面包含的信息量其实是非常可观的，所以声音信号也是一堆向量。

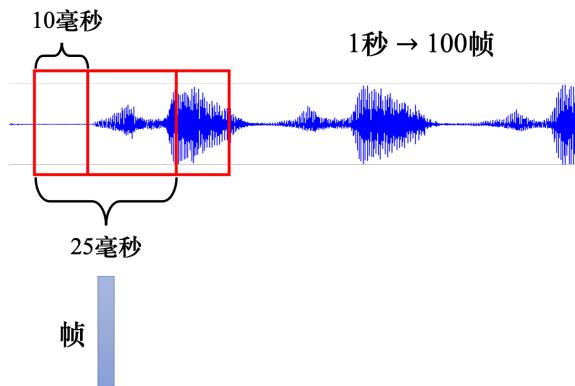


图 3.4 语音处理

一个图 (graph) 也是一堆向量。如图 3.5 所示，社交网络是一个图，在社交网络上面每一个节点就是一个人。每一个节点可以看作是一个向量。每一个人的简介里面的信息（性别、年龄、工作等等）都可以用一个向量来表示。所以一个社交网络可以看做是一堆的向量所组成的。



图 3.5 社交网络<sup>[1]</sup>

药物发现 (drug discovery) 跟图有关，现在像这种的应用非常受到重视。举个例子，如图 3.6 所示，一个分子也可以看作是一个图。如果把一个分子当做是模型的输入，每一个分子可以看作是一个图，分子上面的每一个球就是一个原子，每个原子就是一个向量。每个原子可以用独热向量来表示，比如氢、碳、氧的独热向量表示如式 (3.2) 所示。

$$\mathbf{H} = [1, 0, 0, 0, 0, \dots]$$

$$\mathbf{C} = [0, 1, 0, 0, 0, \dots] \quad (3.2)$$

$$\mathbf{O} = [0, 0, 1, 0, 0, \dots]$$

如果用独热向量来表示每一个原子，一个分子就是一个图，它就是一堆向量。

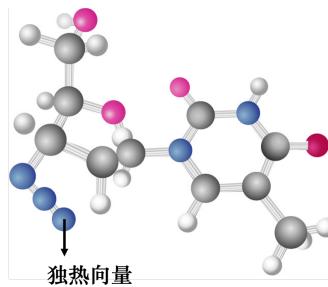


图 3.6 药物发现（需要重绘，版权问题）

### 3.1.1 类型 1：输入与输出数量相同

模型的输入是一堆向量，它可以是文字，可以是语音，可以是图。而输出有三种可能性，第一种可能性是每一个向量都有一个对应的标签。如图 3.7 所示，当模型看到输入是 4 个向量的时候，它就要输出 4 个标签。如果是回归问题，每个标签是一个数值。如果是分类问题，每个标签是一个类别。但是在类型 1 的问题里面，输入跟输出的长度是一样的。模型不需要去烦恼要输出多少的标签，输出多少的标量。反正输入是 4 个向量，输出就是 4 个标量。这是第一种类型。



图 3.7 类型 1：输入与输出数量相同

什么样的应用会用到第一种类型的输出呢？举个例子，如图 3.8 所示，在文字处理上，假设我们要做的是词性标注（Part-Of-Speech tagging, POS tagging）。机器会自动决定每一个词汇的词性，判断该词是名词还是动词还是形容词等等。这个任务并不是很容易，举个例子，现在有一个句子：I saw a saw，这句话的意思是我看到一个锯子，第二个 saw 是名词锯子。所以机器要知道，第一个 saw 是个动词，第二个 saw 是名词，每一个输入的词汇都要

有一个对应的输出的词性。这个任务就是输入跟输出的长度是一样的情况，属于第一个类型的输出。如果是语音，一段声音信号里面有一串向量。每一个向量都要决定它是哪一个音标。这不是真正的语音识别，这是一个语音识别的简化版。如果是社交网络，给定一个社交网络，模型要决定每一个节点有什么样的特性，比如某个人会不会买某个商品，这样我们才知道要不要推荐某个商品给他。以上就是举输入跟输出数量一样的例子，这是第一种可能的输出。

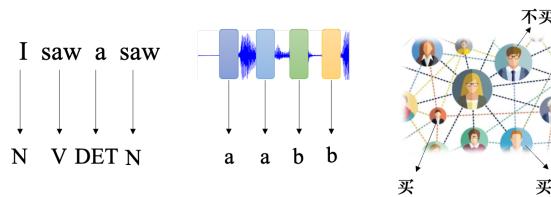


图 3.8 类型 1 应用的例子<sup>[1]</sup>

### 3.1.2 类型 2：输入是一个序列，输出是一个标签

第二种可能的输出如图 3.9 所示，整个序列只需要输出一个标签就好。



图 3.9 类似 2：输入是一个序列，输出是一个标签

举例而言，如图 3.10 所示，输入是文字，比如情感分析 (sentiment analysis)。情感分析就是给机器看一段话，模型要决定说这段话是积极的 (positive) 还是消极的 (negative)。情感分析很有应用价值，假设公司开发的一个产品上线了，我们想要知道网友的评价，但是又不可能一则一则地分析网友的留言。而使用情感分析就可以让机器自动去判别当一则贴文里面提到某个产品的时候，它是积极的还是消极的，这样就可以知道产品在网友心中的评价。给定一整个句子，只需要一个标签（积极的或消极的）。如果是语音，机器听一段声音，再决定是谁讲的这个声音。如果是图，比如给定一个分子，预测该分子的亲水性。

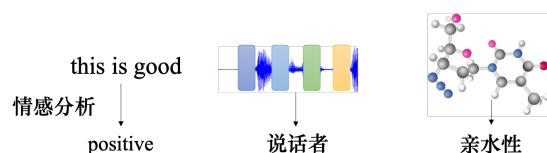


图 3.10 类型 2 的应用例子（亲水性这张图需要重绘，版权问题）

### 3.1.3 类型 3：序列到序列

还有第 3 个可能的输出：我们不知道应该输出多少个标签，机器要自己决定输出多少个标签。如图 3.11 所示，输入是  $N$  个向量，输出可能是  $N'$  个标签。 $N'$  是机器自己决定的。这种任务又叫做序列到序列（Sequence to Sequence, Seq2Seq）的任务。翻译就是序列到序列的任务，因为输入输出是不同的语言，它们的词汇的数量本来就不会一样多。真正的语音识别输入一句话，输出一段文字，其实也是一个序列到序列的任务。



图 3.11 类型 3：序列到序列任务

## 3.2 自注意力的运作原理

我们就先只讲第一个类型：输入跟输出数量一样多的状况，以序列标注(sequence labeling)为例。序列标注要给序列里面的每一个向量一个标签。要怎么解决序列标注的问题呢？直觉的想法就是使用全连接网络(fully-connected network)。如图 3.12 所示，虽然输入是一个序列，但可以不要管它是不是一个序列，各个击破，把每一个向量分别输入到全连接网络里面得到输出。这种做法有非常大的瑕疵，以词性标注为例，给机器一个句子：I saw a saw。对于全连接网络，这个句子中的两个 saw 完全一模一样，它们是同一个词汇。既然全连接网络输入同一个词汇，它没有理由输出不同的东西。但实际上，我们期待第一个 saw 要输出动词，第二个 saw 要输出名词。但全连接网络无法做到这件事，因为这两个 saw 是一模一样的。有没有

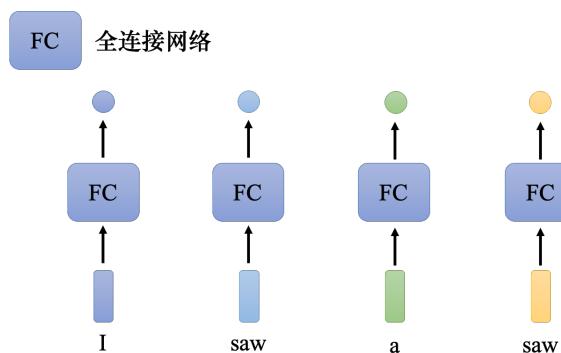


图 3.12 序列标注

有可能让全连接网络考虑更多的信息，比如上下文的信息呢？这是有可能的，如图 3.13 所示，

把每个向量的前后几个向量都“串”起来，一起输入到全连接网络就可以了。

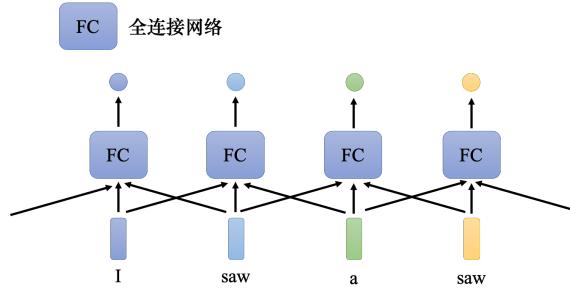


图 3.13 考虑上下文

在语音识别里面，我们不是只看一帧判断这个帧属于哪一个音标，而是看该帧以及其前后 5 个帧（共 11 个帧）来决定它是哪一个音标。所以可以给全连接网络一整个窗口的信息，让它可以考虑一些上下文，即与该向量相邻的其他向量的信息。如图 3.14 所示。但是这种方法还是有极限的，如果有某一个任务不是考虑一个窗口就可以解决的，而是要考虑一整个序列才能够解决，那要怎么办呢？有人可能会想说这个还不容易，把窗口开大一点啊，大到可以把整个序列盖住，就可以了。但是序列的长度是有长有短的，输入给模型的序列的长度，每次可能都不一样。如果要开一个窗口把整个序列盖住，可能要统计一下训练数据，看看训练数据里面最长序列的长度。接着开一个窗口比最长的序列还要长，才可能把整个序列盖住。但是开一个这么大的窗口，意味着全连接网络需要非常多的参数，可能不只运算量很大，还容易过拟合。如果想要更好地考虑整个输入序列的信息，就要用到自注意力模型。

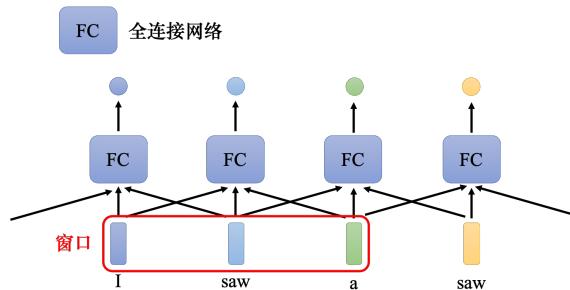


图 3.14 使用窗口来考虑上下文

自注意力模型的运作方式如图 3.15 所示，自注意力模型会“吃”整个序列的数据，输入几个向量，它就输出几个向量。图 3.15 中输入 4 个向量，它就输出 4 个向量。而这 4 个向量都是考虑整个序列以后才得到的，所以输出的向量有一个黑色的框，代表它不是一个普通的向量，它是考虑了整个句子以后才得到的信息。接着再把考虑整个句子的向量丢进全连接网

络，再得到输出。因此全连接网络不是只考虑一个非常小的范围或一个小的窗口，而是考虑整个序列的信息，再来决定现在应该要输出什么样的结果，这就是自注意力模型。

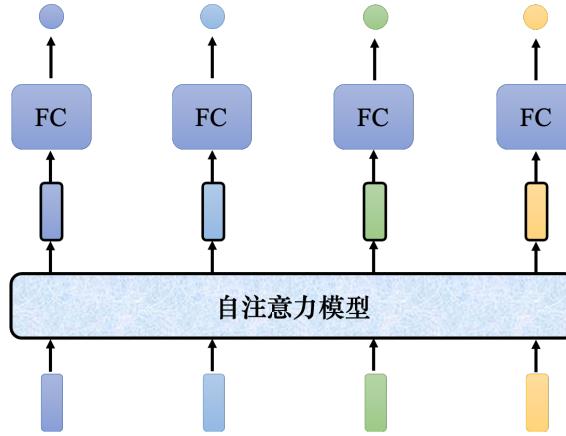


图 3.15 自注意力模型的运作方式

自注意力模型不是只能用一次，可以叠加很多次。如图 3.16 所示，自注意力模型的输出通过全连接网络以后，得到全连接网络的输出。全连接网络的输出再做一次自注意力模型，再重新考虑一次整个输入序列的数据，将得到的数据输入到另一个全连接网络，就可以得到最终的结果。全连接网络和自注意力模型可以交替使用。全连接网络专注于处理某一个位置的信息，自注意力把整个序列信息再处理一次。有关自注意力最知名的相关的论文是“Attention Is All You Need”。在这篇论文里面，谷歌提出了 Transformer 网络架构。Transformer 就是变形金刚，所以提到这个网络的时候，大家会想到变形金刚。而 Transformer 里面最重要的模块是自注意力，它是变形金刚的火种源。有很多更早的论文提出过类似自注意力的架构，只是叫别的名字，比如叫 Self-Matching。“Attention Is All You Need”这篇论文将自注意力模块发扬光大。

自注意力模型的运作过程如图 3.17 所示，其输入是一串的向量，这个向量可能是整个网络的输入，也可能是某个隐藏层（hidden layer）的输出，所以不用  $x$  来表示它，而用  $a$  来表示它，代表它有可能是前面已经做过一些处理，是某个隐藏层的输出。输入一组向量  $a$ ，自注意力要输出一组向量  $b$ ，每个  $b$  都是考虑了所有的  $a$  以后才生成出来的。 $b^1, b^2, b^3, b^4$  是考虑整个输入的序列  $a^1, a^2, a^3, a^4$  才产生出来的。

接下来介绍下向量  $b^1$  产生的过程，了解产生向量  $b^1$  的过程后，剩下向量  $b^2, b^3, b^4$  产生的过程以此类推。怎么产生向量  $b^1$  呢？如图 3.18 所示，第一个步骤是根据  $a^1$  找出输入序列里面跟  $a^1$  相关的其他向量。自注意力的目的是考虑整个序列，但是又不希望把整个序列所

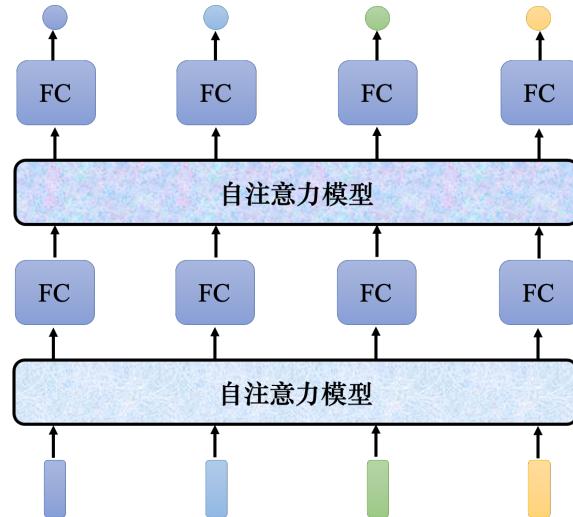


图 3.16 自注意力模型与全连接网络的叠加使用

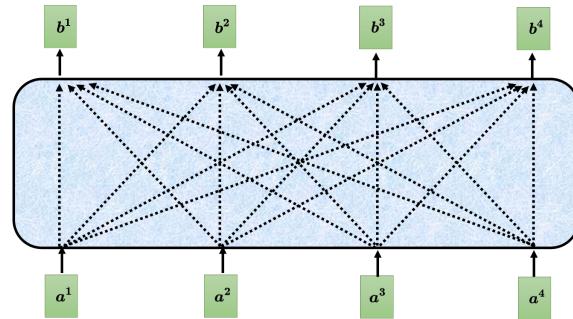
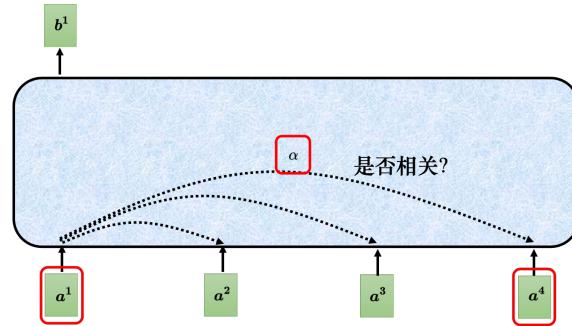


图 3.17 自注意力模型的运作方式

有的信息包在一个窗口里面。所以有一个特别的机制，这个机制是根据向量  $a^1$  找出整个很长的序列里面哪些部分是重要的，哪些部分跟判断  $a^1$  是哪一个标签是有关系的。每一个向量跟  $a^1$  的关联的程度可以用数值  $\alpha$  来表示。自注意力的模块如何自动决定两个向量之间的关联性呢？给它两个向量  $a^1$  跟  $a^4$ ，它怎么计算出一个数值  $\alpha$  呢？我们需要一个计算注意力的模块。

计算注意力的模块使用两个向量作为输入，直接输出数值  $\alpha$ ， $\alpha$  可以当做两个向量的关联的程度。怎么计算  $\alpha$ ？比较常见的做法是用点积（dot product）。如图 3.19a 所示，把输入的两个向量分别乘上两个不同的矩阵，左边这个向量乘上矩阵  $W^q$ ，右边这个向量乘上矩阵  $W^k$ ，得到两个向量  $q$  跟  $k$ ，再把  $q$  跟  $k$  做点积，把它们做逐元素（element-wise）的相乘，再全部加起来以后就得到一个标量（scalar） $\alpha$ ，这是一种计算  $\alpha$  的方式。

其实还有其他的计算方式，如图 3.19b 所示，有另外一个叫做相加（additive）的计算方式，其计算方法就是把两个向量通过  $W^q$ 、 $W^k$  得到  $q$  和  $k$ ，但不是把它们做点积，而是把  $q$

图 3.18 向量  $b^1$  产生的过程

和  $k$  “串”起来“丢”到一个  $\tanh$  函数，再乘上矩阵  $W$  得到  $\alpha$ 。总之，有非常多不同的方法可以计算注意力，可以计算关联程度  $\alpha$ 。但是在接下来的内容里面，我们都只用点积这个方法，这也是目前最常用的方法，也是用在 Transformer 里面的方法。

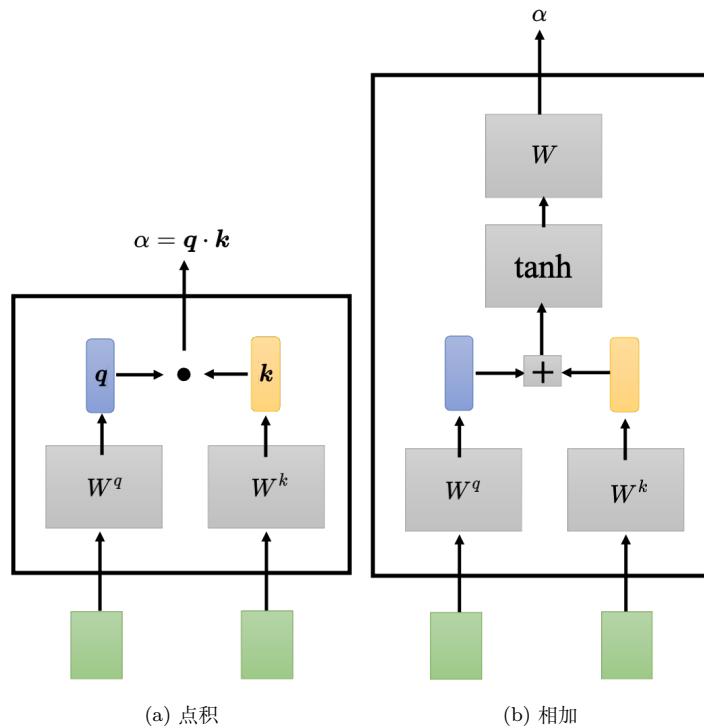


图 3.19 计算向量关联程度的方法

接下来怎么把它套用在自注意力里面呢？分别计算  $a^1$  与  $a^2$ 、 $a^3$ 、 $a^4$  之间的关联性  $\alpha$ 。如图 3.20 所示，把  $a^1$  乘上  $W^q$  得到  $q^1$ 。 $q$  称为查询 (query)，它就像是我们使用搜索引擎查找相关文章所使用的关键字，所以称之为查询。

接下来要去把  $a^2$ 、 $a^3$ 、 $a^4$  乘上  $W^k$  得到向量  $k$ ，向量  $k$  称为键 (key)。把查询  $q^1$  跟键

$k^2$  算内积 (inner-product) 就得到  $\alpha_{1,2}$ 。 $\alpha_{1,2}$  代表查询是  $q^1$  提供的，键是  $k^2$  提供的时候， $q^1$  跟  $k^2$  之间的关联性。关联性  $\alpha$  也被称为注意力的分数。计算  $q^1$  与  $k^2$  的内积也就是计算  $a^1$  与  $a^2$  的注意力的分数。计算出  $a^1$  与  $a^2$  的关联性以后，接下来还需要计算  $a^1$  与  $a^3$ 、 $a^4$  的关联性。把  $a^3$  乘上  $W^k$  得到键  $k^3$ ， $a^4$  乘上  $W^k$  得到键  $k^4$ ，再把键  $k^3$  跟查询  $q^1$  做内积，得到  $a^1$  与  $a^3$  之间的关联性，即  $a^1$  跟  $a^3$  的注意力分数。把  $k^4$  跟  $q^1$  做点积，得到  $\alpha_{1,4}$ ，即  $a^1$  跟  $a^4$  之间的关联性。

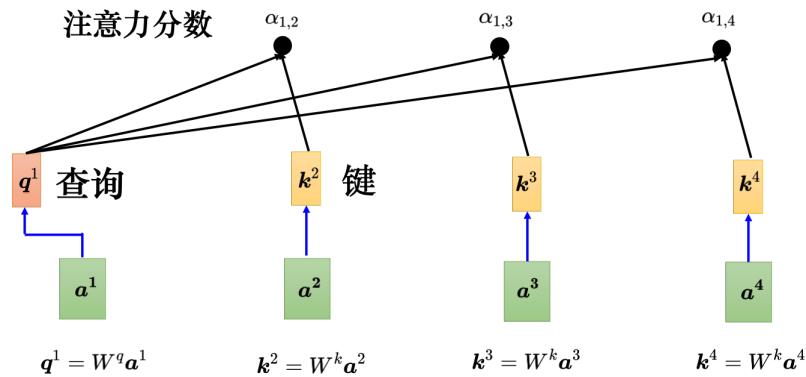


图 3.20 自注意力机制中使用点乘

一般在实践的时候，如图 3.21 所示， $a^1$  也会跟自己算关联性，把  $a^1$  乘上  $W^k$  得到  $k^1$ 。用  $q^1$  跟  $k^1$  去计算  $a^1$  与自己的关联性。计算出  $a^1$  跟每一个向量的关联性以后，接下来会对所有的关联性做一个 softmax 操作，如式 (3.3) 所示，把  $\alpha$  全部取  $e$  的指数，再把指数的值全部加起来做归一化 (normalize) 得到  $\alpha'$ 。这里的 softmax 操作跟分类的 softmax 操作是一模一样的。

$$\alpha'_{1,i} = \exp(\alpha_{1,i}) / \sum_j \exp(\alpha_{1,j}) \quad (3.3)$$

所以本来有一组  $\alpha$ ，通过 softmax 就得到一组  $\alpha'$ 。

Q: 为什么要用 softmax?

A: 这边不一定要用 softmax，可以用别的激活函数，比如 ReLU。有人尝试使用 ReLU，结果发现还比 softmax 好一点。所以不一定要用 softmax，softmax 只是最常见的，我们可以尝试其他激活函数，看能不能试出比 softmax 更好的结果。

得到  $\alpha'$  以后，接下来根据  $\alpha'$  去抽取出序列里面重要的信息。如图 3.22 所示，根据  $\alpha$  可知哪些向量跟  $a^1$  是最有关系的，接下来我们要根据关联性，即注意力的分数来抽取重要的信息。把向量  $a^1$  到  $a^4$  乘上  $W^v$  得到新的向量： $v^1$ 、 $v^2$ 、 $v^3$  和  $v^4$ ，接下来把每一个向量都去

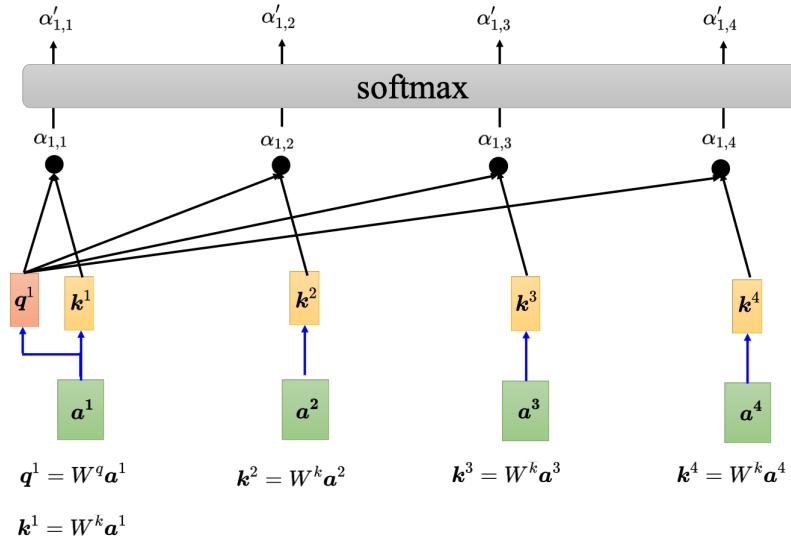
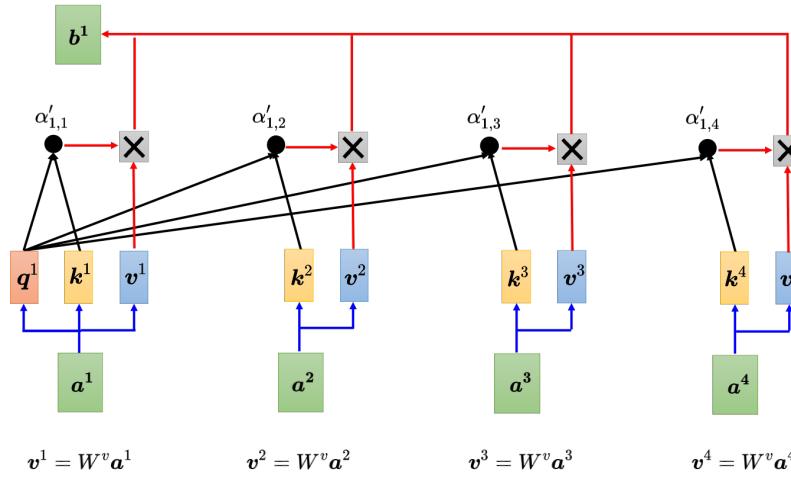


图 3.21 添加 softmax

乘上注意力的分数  $\alpha'$ , 再把它们加起来, 如式 (3.4) 所示。

$$\mathbf{b}^1 = \sum_i \alpha'_{1,i} \mathbf{v}^i \quad (3.4)$$

如果  $a^1$  跟  $a^2$  的关联性很强, 即  $\alpha'_{1,2}$  的值很大。在做加权和 (weighted sum) 以后, 得到的  $\mathbf{b}^1$  的值就可能会比较接近  $\mathbf{v}^2$ , 所以谁的注意力的分数最大, 谁的  $\mathbf{v}$  就会主导 (dominant) 抽出来的结果。这边我们讲述了如何从一整个序列得到  $\mathbf{b}^1$ 。同理, 可以计算出  $\mathbf{b}^2$  到  $\mathbf{b}^4$ 。

图 3.22 根据  $\alpha'$  抽取序列中重要的信息

刚才讲的是自注意力运作的过程, 接下来从矩阵乘法的角度再重新讲一次自注意力的运作过程, 如图 3.23 所示。现在已经知道  $a^1$  到  $a^4$ , 每一个  $a$  都要分别产生  $q$ 、 $k$  和  $v$ ,  $a^1$  要

产生  $q^1, k^1, v^1, a^2$  要产生  $q^2, k^2$  和  $v^2$ , 以此类推。如果要用矩阵运算表示这个操作, 每一个  $a^i$  都乘上一个矩阵  $W^q$  得到  $q^i$ , 这些不同的  $a$  可以合起来当作一个矩阵。什么意思呢?  $a^1$  乘上  $W^q$  得到  $q^1$ ,  $a^2$  也乘上  $W^q$  得到  $q^2$ , 以此类推。把  $a^1$  到  $a^4$  拼起来可以看作是一个矩阵  $I$ , 矩阵  $I$  有四列, 它的列就是自注意力的输入:  $a^1$  到  $a^4$ 。把矩阵  $I$  乘上矩阵  $W^q$  得到  $Q$ 。 $W^q$  是网络的参数,  $Q$  的四个列就是  $q^1$  到  $q^4$ 。

产生  $k$  和  $v$  的操作跟  $q$  是一模一样的,  $a$  乘上  $W^k$  就会得到键  $k$ 。把  $I$  乘上矩阵  $W^k$ , 就得到矩阵  $K$ 。 $K$  的 4 个列就是 4 个键:  $k^1$  到  $k^4$ 。 $I$  乘上矩阵  $W^v$  会得到矩阵  $V$ 。矩阵  $V$  的 4 个列就是 4 个向量  $v^1$  到  $v^4$ 。因此把输入的向量序列分别乘上三个不同的矩阵可得到  $q$ ,  $k$  和  $v$ 。

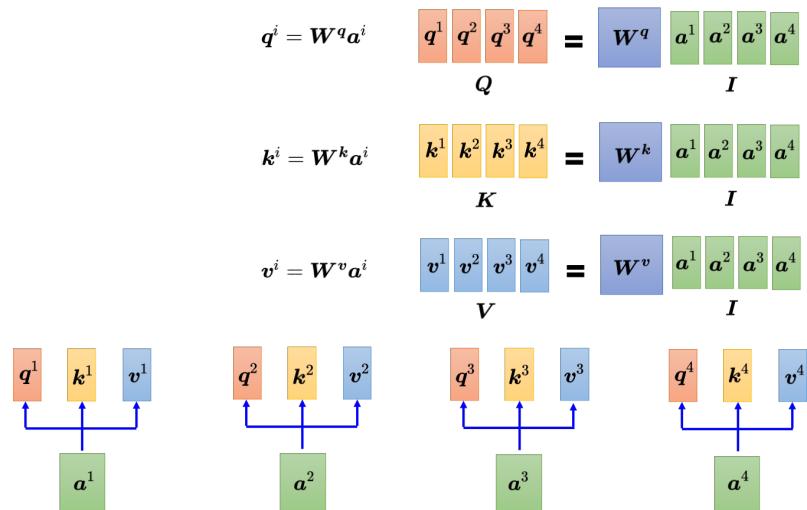


图 3.23 从矩阵乘法的角度理解自注意力的运作过程

如图 3.24 所示, 下一步是每一个  $q$  都会去跟每一个  $k$  去计算内积, 去得到注意力的分数, 先计算  $q^1$  的注意力分数。

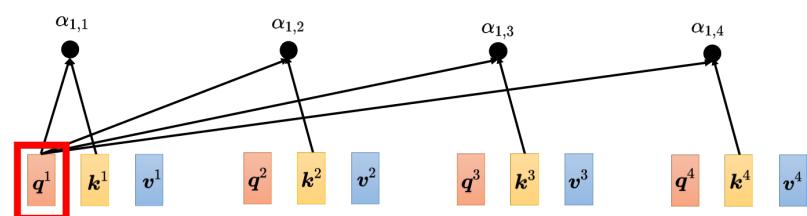


图 3.24 使用  $q^1$  计算注意力分数

如图 3.25 所示, 如果从矩阵操作的角度来看注意力计算这个操作, 把  $q^1$  跟  $k^1$  做内积, 得到  $\alpha_{1,1}$ 。 $q^1$  乘上  $(k^1)^T$ , 也就是  $q^1$  跟  $k^1$  做内积。同理,  $\alpha_{1,2}$  是  $q^1$  跟  $k^2$  做内积,  $\alpha_{1,3}$  是

$q^1$  跟  $k^3$  做内积,  $\alpha_{1,4}$  就是  $q^1$  跟  $k^4$  做内积。这四个步骤的操作, 其实可以把它拼起来, 看作是矩阵跟向量相乘。 $q^1$  乘  $k^1$ ,  $q^1$  乘  $k^2$ ,  $q^1$  乘  $k^3$ ,  $q^1$  乘  $k^4$  这四个动作, 可以看作是把  $(k^1)^T$  到  $(k^4)^T$  拼起来当作是一个矩阵的四行, 把这个矩阵乘上  $q^1$  可得到注意力分数的矩阵, 矩阵的每一行都是注意力的分数, 即  $\alpha_{1,1}$  到  $\alpha_{1,4}$ 。

$$\begin{array}{ll} \alpha_{1,1} = (\mathbf{k}^1)^T \quad \mathbf{q}^1 & \alpha_{1,2} = (\mathbf{k}^2)^T \quad \mathbf{q}^1 \\ \alpha_{1,3} = (\mathbf{k}^3)^T \quad \mathbf{q}^1 & \alpha_{1,4} = (\mathbf{k}^4)^T \quad \mathbf{q}^1 \end{array} \quad \begin{array}{c} \alpha_{1,1} \\ \alpha_{1,2} \\ \alpha_{1,3} \\ \alpha_{1,4} \end{array} = \begin{array}{c} (\mathbf{k}^1)^T \\ (\mathbf{k}^2)^T \\ (\mathbf{k}^3)^T \\ (\mathbf{k}^4)^T \end{array} \quad \mathbf{q}^1$$

图 3.25 从矩阵操作角度来理解注意力分数计算的过程

如图 3.26 所示, 不只是  $q^1$  要对  $k^1$  到  $k^4$  计算注意力,  $q^2$  也要对  $k^1$  到  $k^4$  计算注意力。我们把  $q^2$  也乘上  $k^1$  到  $k^4$ , 得到  $\alpha_{2,1}$  到  $\alpha_{2,4}$ 。现在的操作是一模一样的, 把  $q^3$  乘  $k^1$  到  $k^4$ , 把  $q^4$  乘上  $k^1$  到  $k^4$  可以得到注意力的分数。

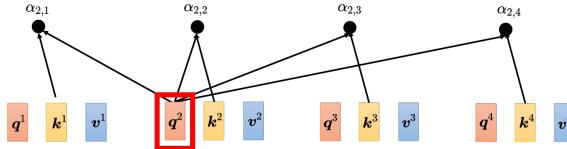


图 3.26 使用  $q^2$  计算注意力分数

如图 3.27 所示, 通过两个矩阵的相乘就得到注意力的分数。一个矩阵的行就是  $k$ , 即  $k^1$  到  $k^4$ 。另外一个矩阵的列就是  $q$ , 即  $q^1$  到  $q^4$ 。把  $k$  所形成的矩阵  $K^T$  乘上  $q$  所形成的矩阵  $Q$  就得到这些注意力的分数。假设  $K$  的列是  $k^1$  到  $k^4$ , 在这边相乘的时候, 要对矩阵  $K$  做一下转置得到  $K^T$ ,  $K^T$  乘上  $Q$  就得到矩阵  $A$ ,  $A$  里面存的就是  $Q$  跟  $K$  之间的注意力的分数。对注意力的分数做一下归一化 (normalization), 比如使用 softmax, 对  $A$  的每一列做 softmax, 让每一列里面的值相加是 1。softmax 不是唯一的选项, 完全可以选择其他的操作, 比如 ReLU 之类的, 得到的结果也不会比较差。由于把  $\alpha$  做 softmax 操作以后, 它得到的值有异于  $\alpha$  的原始值, 所以用  $A'$  来表示通过 softmax 以后的结果。

如图 3.28 所示, 计算出  $A'$  以后, 需要把  $v^1$  到  $v^4$  乘上对应的  $\alpha$  再相加得到  $b$ 。如果把  $v^1$  到  $v^4$  当成是矩阵  $V$  的 4 个列拼起来, 则把  $A'$  的第一个列乘上  $V$  就得到  $b^1$ , 把  $A'$  的第二个列乘上  $V$  得到  $b^2$ , 以此类推。所以等于把矩阵  $A'$  乘上矩阵  $V$  得到矩阵  $O$ 。矩阵  $O$  里面的每一个列就是自注意力的输出  $b^1$  到  $b^4$ 。所以整个自注意力的操作过程可分为以下

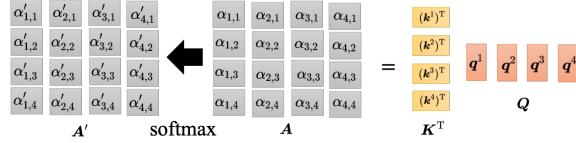


图 3.27 注意力分数的计算过程

步骤，先产生了  $q$ 、 $k$  和  $v$ ，再根据  $q$  去找出相关的位置，然后对  $v$  做加权和。这一串操作就是一连串矩阵的乘法。

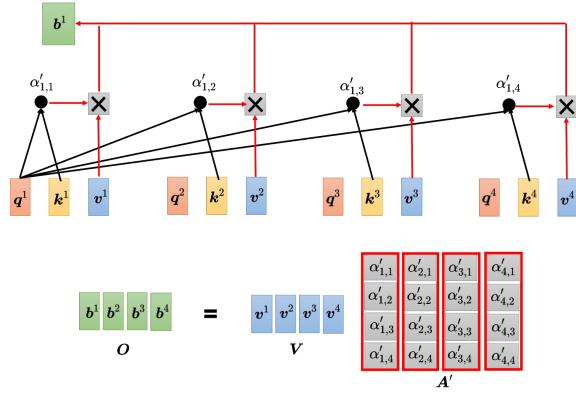


图 3.28 自注意力输出的计算过程

如图 3.29 所示，自注意力的输入是一组的向量，将这排向量拼起来可得到矩阵  $I$ 。输入  $I$  分别乘上三个矩阵： $W^q$ 、 $W^k$  跟  $W^v$ ，得到三个矩阵  $Q$ 、 $K$  和  $V$ 。接下来  $Q$  乘上  $K^T$  得到矩阵  $A$ 。把矩阵  $A$  做一些处理可得到  $A'$ ， $A'$  称为注意力矩阵（attention matrix）。把  $A'$  再乘上  $V$  就得到自注意力层的输出  $O$ 。自注意力的操作较为复杂，但自注意力层里面唯一需要学的参数就只有  $W^q$ 、 $W^k$  跟  $W^v$ 。只有  $W^q$ 、 $W^k$ 、 $W^v$  是未知的，需要通过训练数据把它学习出来的。其他的操作都没有未知的参数，都是人为设定好的，都不需要通过训练数据学习。

### 3.3 多头注意力

自注意力有一个进阶的版本——**多头自注意力**（multi-head self-attention）。多头自注意力的使用是非常广泛的，有一些任务，比如翻译、语音识别，用比较多的头可以得到比较好的结果。至于需要用多少的头，这个又是另外一个超参数（hyperparameter），也是需要调的。为什么需要比较多的头呢？在使用自注意力计算相关性的时候，就是用  $q$  去找相关的  $k$ 。但是相关有很多种不同的形式，所以也许可以有多个  $q$ ，不同的  $q$  负责不同种类的相关

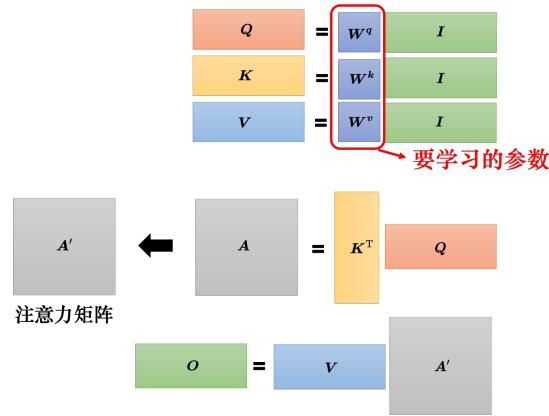


图 3.29 从矩阵乘法的角度来理解注意力

性，这就是多头注意力。如图 3.30 所示，先把  $a$  乘上一个矩阵得到  $q$ ，接下来再把  $q$  乘上另外两个矩阵，分别得到  $q^1$ 、 $q^2$ 。用两个上标， $q^{i,1}$  跟  $q^{i,2}$  代表有两个头， $i$  代表的是位置，1 跟 2 代表是这个位置的第几个  $q$ ，这个问题里面有两种不同的相关性，所以需要产生两种不同的头来找两种不同的相关性。既然  $q$  有两个， $k$  也要有两个， $v$  也要有两个。怎么从  $q$  得到  $q^1$ 、 $q^2$ ，怎么从  $k$  得到  $k^1$ 、 $k^2$ ，怎么从  $v$  得到  $v^1$ 、 $v^2$ ? 其实就是把  $q$ 、 $k$ 、 $v$  分别乘上两个矩阵，得到不同的头。对另外一个位置也做一样的事情，另外一个位置在输入  $a$  以后，它也会得到两个  $q$ 、两个  $k$ 、两个  $v$ 。接下来怎么做自注意力呢，跟之前讲的操作是一模一样的，只是现在 1 那一类的一起做，2 那一类的一起做。也就是  $q^1$  在算这个注意力的分数的时候，就不要管  $k^2$  了，它就只管  $k^1$  就好。 $q^{i,1}$  分别与  $k^{i,1}$ 、 $k^{j,1}$  算注意力，在做加权和的时候也不要管  $v^2$  了，看  $v^{i,1}$  跟  $v^{j,1}$  就好，把注意力的分数乘  $v^{i,1}$  和  $v^{j,1}$ ，再相加得到  $b^{i,1}$ ，这边只用了其中一个头。

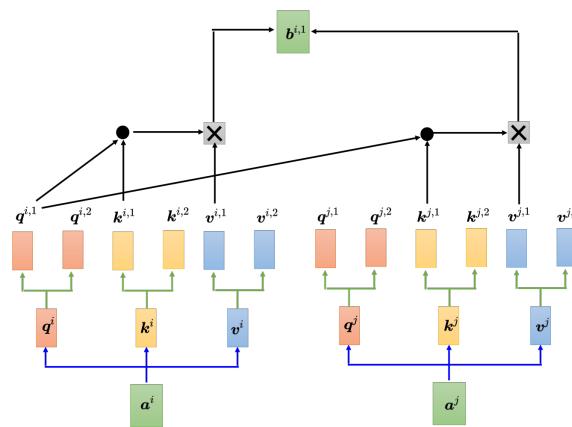


图 3.30 多头自注意力的计算过程

如图 3.31 所示，我们可以使用另外一个头做相同的事情。 $q^2$  只对  $k^2$  做注意力，在做加权和的时候，只对  $v^2$  做加权和得到  $b^{i,2}$ 。如果有多个头，如 8 个头、16 个头，操作也是一样的。

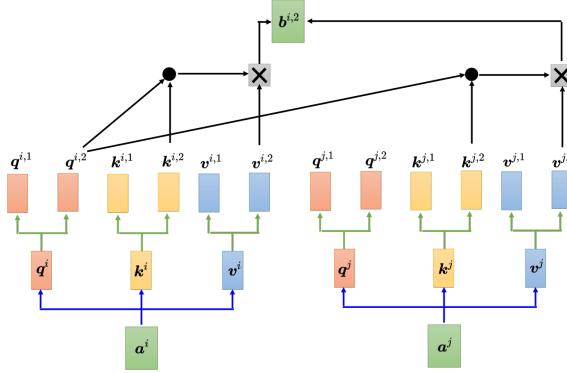


图 3.31 多头自注意力另一个头的计算过程

如图 3.32 所示，得到  $b^{i,1}$  跟  $b^{i,2}$ ，可能会把  $b^{i,1}$  跟  $b^{i,2}$  接起来，再通过一个 transform，即再乘上一个矩阵然后得到  $b^i$ ，再送到下一层去，这就是自注意力的变形——多头自注意力。

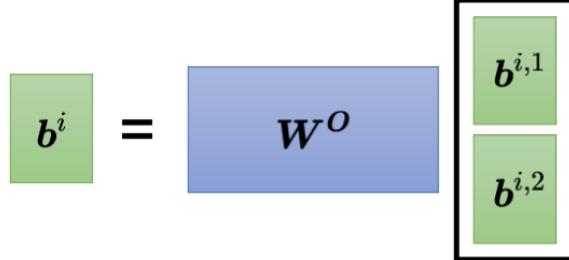


图 3.32 从矩阵乘法的角度来理解多头自注意力

### 3.4 位置编码

讲到目前为止，自注意力层少了一个也许很重要的信息，即位置的信息。对一个自注意力层而言，每一个输入是出现在序列的最前面还是最后面，它是完全没有这个信息的。有人可能会问：输入不是有位置 1、2、3、4 吗？但 1、2、3、4 是作图的时候，为了帮助大家理解所标上的一个编号。对自注意力而言，位置 1、位置 2、位置 3 跟位置 4 没有任何差别，这四个位置的操作是一模一样的。对它来说， $q^1$  跟  $q^4$  的距离并没有特别远，1 跟 4 的距离并没有特别远，2 跟 3 的距离也没有特别近，对它来说就是天涯若比邻，所有的位置之间的距离都是一样的。

的，没有谁在整个序列的最前面，也没有谁在整个序列的最后面。但是这可能会有一个问题：位置的信息被忽略了，而有时候位置的信息很重要。举个例子，在做词性标注的时候，我们知道动词比较不容易出现在句首，如果某一个词汇它是放在句首的，它是动词的可能性就比较低，位置的信息往往也是有用的。可是到目前为止，自注意力的操作里面没有位置的信息。因此做自注意力的时候，如果我们觉得位置的信息很重要，需要考虑位置信息时，就要用到**位置编码 (positional encoding)**。如图 3.33 所示，位置编码为每一个位置设定一个向量，即位置向量 (positional vector)。位置向量用  $e^i$  来表示，上标  $i$  代表位置，不同的位置就有不同的向量，不同的位置都有一个专属的  $e$ ，把  $e$  加到  $a^i$  上面就结束了。这相当于告诉自注意力位置的信息，如果看到  $a^i$  被加上  $e^i$ ，它就知道现在出现的位置应该是在  $i$  这个位置。

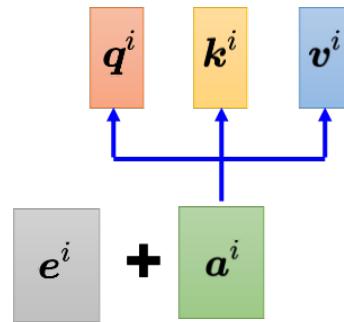


图 3.33 位置编码

最早的 Transformer 论文 “Attention Is All You Need” 用的  $e^i$  如图 3.34 所示。图上面每一列就代表一个  $e$ ，第一个位置就是  $e^1$ ，第二个位置就是  $e^2$ ，第三个位置就是  $e^3$ ，以此类推。每一个位置的  $a$  都有一个专属的  $e$ 。模型在处理输入的时候，它可以知道现在的输入的位置的信息，这个位置向量是人为设定的。人为设定的向量有很多问题，假设在定这个向量的时候只定到 128，但是序列的长度是 129，怎么办呢？在最早的 “Attention Is All You Need” 论文中，其位置向量是通过正弦函数和余弦函数所产生的，避免了人为设定向量固定长度的尴尬。

Q: 为什么要通过正弦函数和余弦函数产生向量，有其他选择吗？为什么一定要这样产生手工的位置向量呢？

A: 不一定要通过正、余弦函数来产生向量，我们可以提出新的方法。此外，不一定要这样产生手工的向量，位置编码仍然是一个尚待研究的问题，甚至位置编码是可以根据数据学出来的。有关位置编码，可以参考论文“Learning to Encode Position for Transformer with Continuous Dynamical Model”，该论文比较了不同的位置编码方法并提出了新的位置编码。

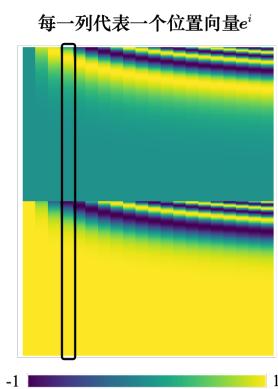


图 3.34 Transformer 中的自注意力

如图 3.35a 所示，最早的位置编码是用正弦函数所产生的，图 3.35a 中每一行代表一个位置向量。如图 3.35b 所示，位置编码还可以使用循环神经网络生成。总之，位置编码可通过各种不同的方法来产生。目前还不知道哪一种方法最好，这是一个尚待研究的问题。所以不用纠结为什么正弦函数最好，我们永远可以提出新的做法。

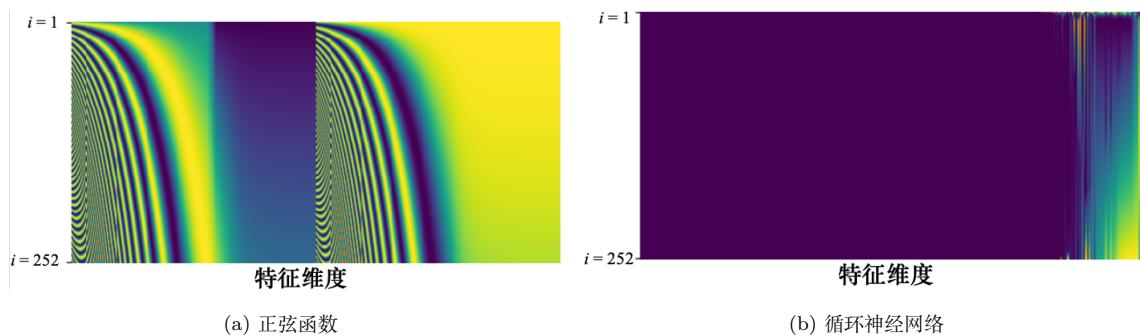


图 3.35 产生位置编码的各种方法<sup>[2]</sup>

### 3.5 截断自注意力

自注意力的应用很广泛，在自然语言处理领域，除了 Transformer，还有 BERT 也用到了自注意力，所以自注意力在自然语言处理上面的应用是大家都较为熟悉的，但自注意力不是只能用在自然语言处理相关的应用上，它还可以用在很多其他的问题上。比如在做语音的时候，也可以用自注意力。不过将自注意力用于语音处理时，可以对自注意力做一些小小的改动。举个例子，如果要把一段声音信号表示成一组向量，这排向量可能会非常长。在做语音识别的时候，把声音信号表示成一组向量，而每一个向量只代表了 10 毫秒的长度而已。所以如果是 1 秒钟的声音信号，它就有 100 个向量了，5 秒钟的声音信号就有 500 个向量，随便讲一句话都是上千个向量了。所以一段声音信号，通过向量序列描述它的时候，这个向量序列的长度是非常大的。非常大的长度会造成什么问题呢？在计算注意力矩阵的时候，其复杂度（complexity）是长度的平方。假设该矩阵的长度为  $L$ ，计算注意力矩阵  $A'$  需要做  $L \times L$  次的内积，如果  $L$  的值很大，计算量就很可观，并且需要很大内存（memory）才能够把该矩阵存下来。所以如果在做语音识别的时候，我们讲一句话，这一句话所产生的这个注意力矩阵可能会太大，大到我们根本就不容易处理，不容易训练。所以在做语音的时候有一招叫做**截断自注意力（truncated self-attention）**。截断自注意力做的事情就是，在做自注意力的时候不要看一整句话，就只看一个小的范围就好，这个范围是人设定的。在做语音识别的时候，如果要辨识某个位置有什么样的音标，这个位置有什么样的内容，并不需要看整句话，只要看这句话以及它前后一定范围之内的信息，就可以判断。在做自注意力的时候，也许没有必要让自注意力考虑一整个句子，只需要考虑一个小范围就好，这样就可以加快运算的速度。这就是截断自注意力。

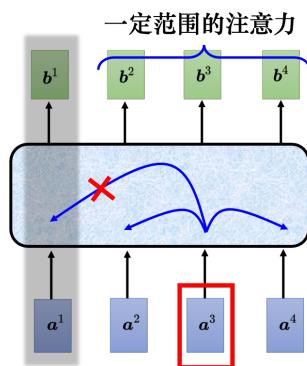


图 3.36 截断自注意力

### 3.6 自注意力与卷积神经网络对比

自注意力还可以被用在图像上。到目前为止，在提到自注意力的时候，自注意力适用的范围是输入为一组向量的时候。一张图像可以看作是一个向量序列，如图 3.37 所示，一张分辨率为  $5 \times 10$  的图像（图 3.37a）可以表示为一个大小为  $5 \times 10 \times 3$  的张量（图 3.37b），3 代表 RGB 这 3 个通道（channel），每一个位置的像素可看作是一个三维的向量，整张图像是  $5 \times 10$  个向量。所以可以换一个角度来看图像，图像其实也是一个向量序列，它既然也是一个向量序列，完全可以用自注意力来处理一张图像。自注意力在图像上的应用，读者可以参考“Self-Attention Generative Adversarial Networks”和“End-to-End Object Detection with Transformers”这两篇论文。

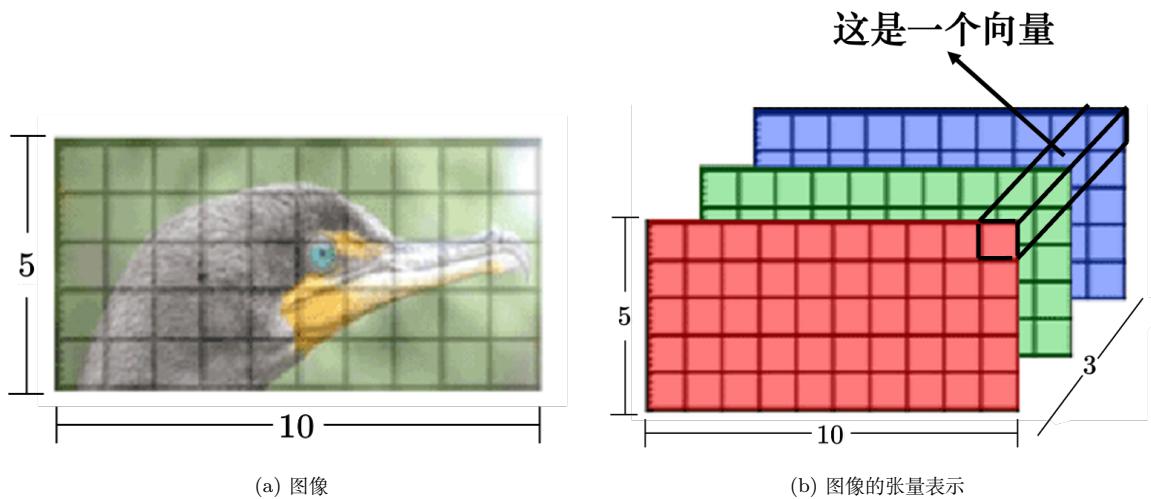


图 3.37 使用自注意力处理图像<sup>[3]</sup>

自注意力跟卷积神经网络之间有什么样的差异或者关联？如图 3.38a 所示，如果用自注意力来处理一张图像，假设红色框内的“1”是要考虑的像素，它会产生查询，其他像素产生键。在做内积的时候，考虑的不是一个小的范围，而是整张图像的信息。如图 3.38b 所示，在做卷积神经网络的时候，卷积神经网络会“画”出一个感受野（receptive field），每一个滤波器（filter），每一个神经元，只考虑感受野范围里面的信息。所以如果我们比较卷积神经网络跟自注意力会发现，卷积神经网络可以看作是一种简化版的自注意力，因为在做卷积神经网络的时候，只考虑感受野里面的信息。而在做自注意力的时候，会考虑整张图像的信息。在卷积神经网络里面，我们要划定感受野。每一个神经元只考虑感受野里面的信息，而感受野的大小是人决定的。而用自注意力去找出相关的像素，就好像是感受野是自动被学出来的，网

络自己决定感受野的形状。网络决定说以这个像素为中心，哪些像素是真正需要考虑的，哪些像素是相关的，所以感受野的范围不再是人工划定，而是让机器自己学出来。关于自注意力跟卷积神经网络的关系，读者可以读论文“On the Relationship between Self-attention and Convolutional Layers”，这篇论文里面会用数学的方式严谨地告诉我们，卷积神经网络就是自注意力的特例。

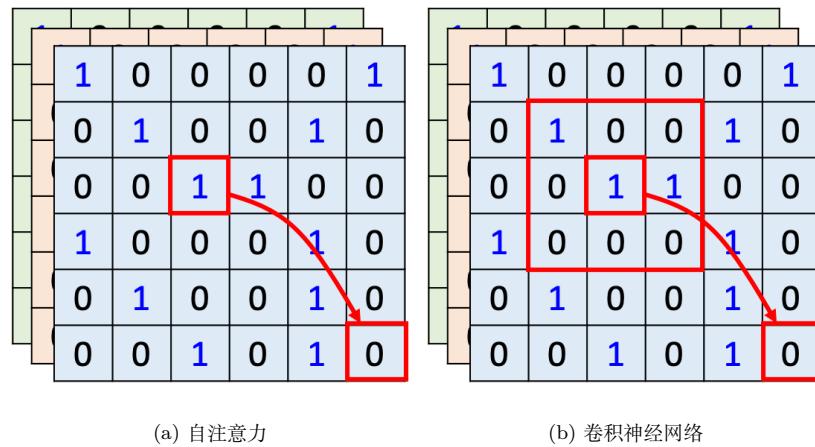


图 3.38 自注意力和卷积神经网络的区别

自注意力只要设定合适的参数，就可以做到跟卷积神经网络一模一样的事情。卷积神经网络的函数集（function set）与自注意力的函数集的关系如图 3.39 所示。所以自注意力是更灵活的卷积神经网络，而卷积神经网络是受限制的自注意力。自注意力只要通过某些设计、某些限制就会变成卷积神经网络。



图 3.39 卷积神经网络的函数集与自注意力的函数集的关系

既然卷积神经网络是自注意力的一个子集，说明自注意力更灵活。更灵活的模型需要更多的数据。如果数据不够，就有可能过拟合。而比较有限制的模型，它适合在数据少的时候使用，它可能比较不会过拟合。如果限制设的好，也会有不错的结果。谷歌的论文“An Image

is Worth 16x16 Words: Transformers for Image Recognition at Scale” 把自注意力应用在图像上面，把一张图像拆成  $16 \times 16$  个图像块 (patch)，它把每一个图像块想像成是一个字 (word)。因为一般自注意力比较常用在自然语言处理上面，所以我们可以想像每一个图像块就是一个字。如图 3.40 所示，横轴是训练的图像的量，对谷歌来说用的所谓的数据量比较少，也是我们没有办法用的数据量。这边有 1000 万张图，是数据量比较小的设置 (setting)，数据量比较大的设置呢，有 3 亿张图像。在这个实验里面，自注意力是浅蓝色的这一条线，卷积神经网络是深灰色的这条线。随着数据量越来越多，自注意力的结果越来越好。最终在数据量最多的时候，自注意力可以超过卷积神经网络，但在数据量少的时候，卷积神经网络是可以比自注意力得到更好的结果的。自注意力的弹性比较大，所以需要比较多的训练数据，训练数据少的时候就会过拟合。而卷积神经网络的弹性比较小，在训练数据少的时候结果比较好。但训练数据多的时候，它没有办法从更大量的训练数据得到好处。这就是自注意力跟卷积神经网络的比较。

Q: 自注意力跟卷积神经网络应该选哪一个?

A: 事实上可以都用，比如 conformer 里面同时用到了自注意力和卷积神经网络。

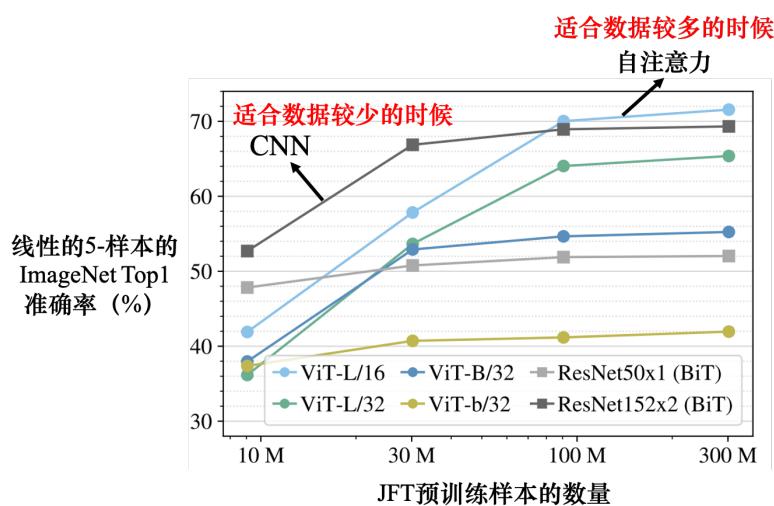


图 3.40 自注意力与卷积神经网络对比<sup>[4]</sup>

### 3.7 自注意力与循环神经网络对比

我们来比较一下自注意力跟循环神经网络 (Recurrent Neural Network, RNN)。目前，循环神经网络的角色很大一部分都可以用自注意力来取代了。但循环神经网络跟自注意力一样，

都是要处理输入是一个序列的状况。如图 3.41a 所示，在循环神经网络里面有一个输入序列、一个隐状态的向量、一个循环神经网络的块（block）。循环神经网络的块“吃”记忆的向量，输出一个东西。这个东西会输入全连接网络来进行预测。

循环神经网络中的隐状态存储了历史信息，可以看作一种记忆（Memory）。

接下来当第二个向量作为输入的时候，前一个时间点“吐”出来的东西也会作为输入丢进循环神经网络里面产生新的向量，再拿去给全连接网络。第三个向量进来的时候，第三个向量跟前一个时间点的输出，一起丢进循环神经网络再产生新的输出。在第四个向量输入的时候，把第四个向量跟前一个时间点产生出来的输出再一起做处理，得到新的输出再通过全连接网络的层，这就是循环神经网络。它跟自注意力做的事情也非常像，它们的输入都是一个向量序列。如图 3.41b 所示，自注意力输出是一个向量序列，该序列中的每一个向量都考虑了整个输入序列，再输入到全连接网络去做处理。循环神经网络也会输出一组向量，这排向量也会给全连接网络做进一步的处理。

自注意力跟循环神经网络有一个显而易见的不同，自注意力的每一个向量都考虑了整个输入的序列，而循环神经网络的每一个向量只考虑了左边已经输入的向量，它没有考虑右边的向量。但循环神经网络也可以是双向的，所以如果用双向循环神经网络（Bidirectional Recurrent Neural Network, Bi-RNN），那么每一个隐状态的输出也可以看作是考虑了整个输入的序列。但是假设把循环神经网络的输出跟自注意力的输出拿来做对比，就算使用双向循环神经网络还是有一些差别的。如图 3.41a 所示，对于循环神经网络，如果最右边黄色的向量要考虑最左边的输入，它就必须把最左边的输入存在记忆里面，才能不“忘掉”，一路带到最右边，才能够在最后一个时间点被考虑。但对自注意力没有这个问题，它只要输出一个查询，输出一个键，只要它们匹配（match）得起来，天涯若比邻。自注意力可以轻易地从整个序列上非常远的向量抽取信息，所以这是循环神经网络跟自注意力一个不一样的地方。还有另外一个更主要的不同是，循环神经网络在处理输入、输出均为一组序列的时候，是没有办法并行化的。比如计算第二个输出的向量，不仅需要第二个输入的向量，还需要前一个时间点的输出向量。当输入是一组向量，输出是另一组向量的时候，循环神经网络无法并行处理所有的输出，但自注意力可以。自注意力输入一组向量，再输出的时候，每一个向量是并行产生的，并不需要等谁先运算完才能把其他向量运算出来。由于自注意力输出向量序列时，每一个向量是同时产生的。所以在运算速度上，自注意力会比循环神经网络更有效率。很多的应用已经把循

环神经网络的架构逐渐改成自注意力的架构了。如果想要更进一步了解循环神经网络跟自注意力的关系，可以阅读论文“Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention”。

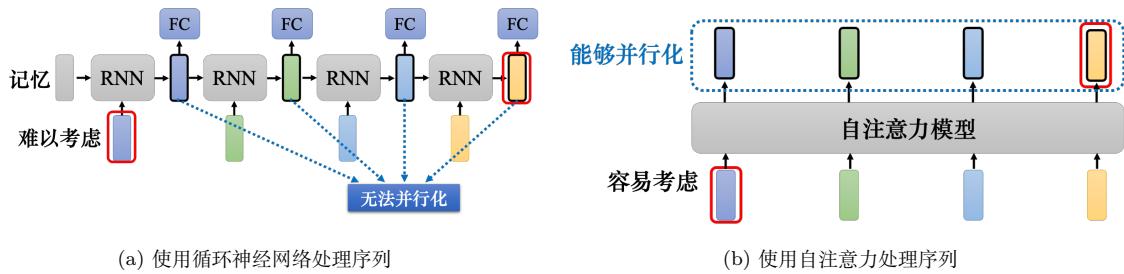


图 3.41 自注意力与循环神经网络对比

自注意力也可以被用在图上面，图也可以看作是一堆向量，如果是一堆向量，就可以用自注意力来处理。但把自注意力用在图上面，会有些地方不一样。图中的每一个节点（node）可以表示成一个向量。但我们不只有节点的信息，还有边（edge）的信息。如果节点之间是有相连的，这些节点也就是有关联的。之前在做自注意力的时候，所谓的关联性是网络自己找出来的。但是现在既然有了图的信息，关联性就不需要机器自动找出来，图上面的边已经暗示了节点跟节点之间的关联性。所以当把自注意力用在图上面的时候，我们可以在计算注意力矩阵的时候，只计算有边相连的节点就好。举个例子，如图 3.42 所示，在这个图上，节点 1 只和节点 5、6、8 相连，因此只需要计算节点 1 和节点 5、节点 6、节点 8 之间的注意力分数；节点 2 之和节点 3 相连，因此只需要计算节点 2 和节点 3 之间的注意力的分数，以此类推。如果两个节点之间没有相连，其实就暗示我们这两个节点之间没有关系。既然没有关系，就不需要再去计算它的注意力分数，直接把它设为 0 就好了。因为图往往是人为根据某些领域知识（domain knowledge）建出来的，所以如果领域知识告诉我们这两个向量之间没有关联，就没有必要再用机器去学习这件事情。当把自注意力按照这种限制用在图上面的时候，其实就是一种图神经网络（Graph Neural Network，GNN）。

自注意力有非常多的变形，论文“Long Range Arena: A Benchmark for Efficient Transformers”里面比较了各种不同的自注意力的变形。因为自注意力最大的问题是其运算量非常大，所以如何减少自注意力的运算量是未来可以研究的重点方向。自注意力最早是用在 Transformer 上面，所以很多人讲 Transformer 的时候，其实指的是自注意力。有人说广义的 Transformer 指的就是自注意力，所以来各种的自注意力的变形都叫做是 xxformer，比

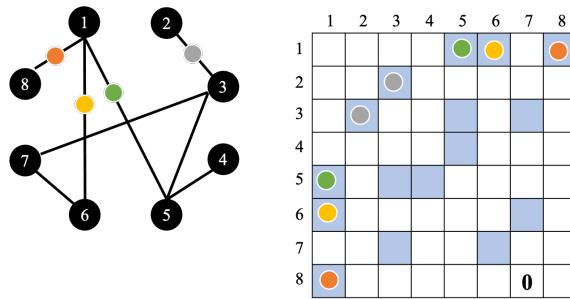


图 3.42 自注意力在图上的应用

如 Linformer、Performer、Reformer 等等。这些新的 xxformer 往往比原来的 Transformer 性能差一点，但是速度会比较快。到底什么样的自注意力才能够真的又快又好？这仍然是一个尚待研究的问题。如果想要对自注意力进一步研究的话，可以参考论文“Efficient Transformers: A Survey”，该论文介绍了各种自注意力的变形。

## 参考文献

- [1] Shreyansh nanawati 的文章 “Social Network Analytics” [Z].
- [2] LIU X, YU H F, DHILLON I S, et al. Learning to encode position for transformer with continuous dynamical model[C]//International Conference on Machine Learning(ICML). 2020: 6327–6335.
- [3] SINGH B P. Imaging applications of charge coupled devices (ccds) for cherenkov telescope [R]. 2015.
- [4] DOSOVITSKIY A, BEYER L, KOLESNIKOV A, et al. An image is worth 16x16 words: Transformers for image recognition at scale[C]//International Conference on Learning Representations(ICLR). 2021.

## 第 4 章 Transformer

### 4.1 序列到序列模型

Transformer 是一个序列到序列（Sequence-to-Sequence，Seq2Seq）的模型，我们先介绍下序列到序列模型。序列到序列模型输入和输出都是一个序列，输入与输出序列长度之间的关系有两种情况。第一种情况下，输入跟输出的长度一样；第二种情况下，机器决定输出的长度。

### 4.2 序列到序列模型的应用

#### 4.2.1 语音识别、机器翻译与语音翻译

序列到序列模型的常见应用如图 4.1 所示。

- 语音识别：输入是声音信号，输出是语音识别的结果，即输入的这段声音信号所对应的文字。我们用圆圈来代表文字，比如每个圆圈代表中文里面的一个方块字。输入跟输出的长度有一些关系，但没有绝对的关系，输入的声音信号的长度是  $T$ ，并无法根据  $T$  得到输出的长度  $N$ 。其实可以由机器自己决定输出的长度，由机器去听这段声音信号的内容，决定输出的语音识别结果。
- 机器翻译：机器输入一个语言的句子，输出另外一个语言的句子。输入句子的长度是  $N$ ，输出句子的长度是  $N'$ 。输入“机器学习”四个字，输出是两个英语的词汇：“machine learning”，但是并不是所有中文跟英语的关系都是输出就是输入的二分之一。 $N$  跟  $N'$  之间的关系由机器决定。
- 语音翻译：我们对机器说一句话，比如“machine learning”，机器直接把听到的英语的声音信号翻译成中文。

Q: 为什么要做语音翻译，把语音识别系统跟机器翻译系统接起来就直接是语音翻译。

A: 世界上很多语言是没有文字的，无法做语音识别。因此需要对这些语言做语音翻译，直接把它翻译成文字。

以闽南语的语音识别为例，闽南语的文字不是很普及，一般人不一定能看懂。因此我们想做语音的翻译，对机器讲一句闽南语，它直接输出的是同样意思的中文的句子，这样一般人就可以看懂。我们可以训练一个神经网络，该神经网络输入某一种语言的声音信号，输出是另

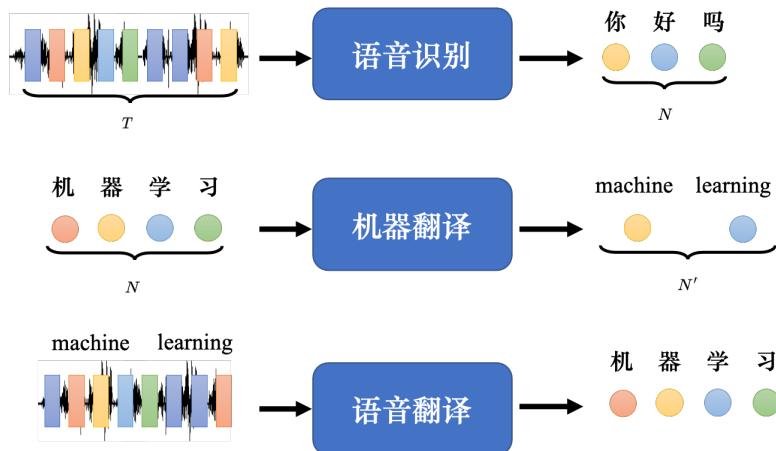


图 4.1 序列到序列的常见应用

外一种语言的文字，需要学到闽南语的声音信号跟中文文字的对应关系。YouTube 上面有很多的乡土剧，乡土剧是闽南语语音、中文字幕，所以只要下载它的闽南语语音和中文字幕，这样就有闽南语声音信号跟中文之间的对应关系，就可以训练一个模型来做闽南语的语音识别：输入闽南语，输出中文。李宏毅实验室下载了 1500 个小时的乡土剧的数据，并用其来训练一个语音识别系统。这会有一些问题，比如乡土剧有很多噪声、音乐，乡土剧的字幕不一定跟声音能对应起来。可以忽略这些问题，直接训练一个模型，输入是声音信号，输出直接是中文的文字，这样训练能够做一个闽南语语音识别系统。

#### 4.2.2 语音合成

输入文字、输出声音信号就是语音合成（Text-To-Speech, TTS）。现在还没有真的做端到端（end-to-end）的模型，以闽南语的语音合成为例，其使用的模型还是分成两阶，首先模型会先把中文的文字转成闽南语的拼音，再把闽南语的拼音转成声音信号。从闽南语的拼音转成声音信号这一段是通过序列到序列模型 echotron 实现的

#### 4.2.3 聊天机器人

除了语音以外，文本也很广泛的使用了序列到序列模型。比如用序列到序列模型可用来训练一个聊天机器人。聊天机器人就是我们对它说一句话，它要给出一个回应。因为聊天机器人的输入输出都是文字，文字是一个向量序列，所以可用序列到序列的模型来做一个聊天机器人。我们可以收集大量人的对话（比如电视剧、电影的台词等等），如图 5.3 所示，假设在对话里面有出现，一个人说：“Hi”，另外一个人说：“Hello! How are you today?”。我们可以

教机器，看到输入是“Hi”，输出就要跟“Hello! How are you today?”越接近越好。



图 4.2 聊天机器人的例子

#### 4.2.4 问答任务

序列到序列模型在自然语言处理 (Natural Language Processing, NLP) 的领域的应用很广泛，而很多自然语言处理的任务都可以想成是问答 (Question Answering, QA) 的任务，比如下面是一些例子。

- 翻译。机器读的文章是一个英语句子，问题是这个句子的德文翻译是什么？输出的答案就是德文。
- 自动做摘要：给机器读一篇长的文章，让它把长的文章的重点找出来，即给机器一段文字，问题是这段文字的摘要是什么。
- 情感分析 (sentiment analysis)：机器要自动判断一个句子是正面的还是负面的。如果把情感分析看成是问答的问题，问题是给定句子是正面还是负面的，希望机器给出答案。

问答就是给机器读一段文字，问机器一个问题，希望它可以给出一个正确的答案。

因此各式各样的自然语言处理的问题往往都可以看作是问答的问题，而问答的问题可以用序列到序列模型来解。序列到序列模型的输入是一篇文章和一个问题，输出就是问题的答案。问题加文章合起来是一段很长的文字，答案是一段文字。只要是输入一个序列，输出是一个序列，序列到序列模型就可以解。虽然各种自然语言处理的问题都能用序列到序列模型来解，但是对多数自然语言处理的任务或对多数的语音相关的任务而言，往往为这些任务定制化模型会得到更好的结果。序列到序列模型就像瑞士刀，瑞士刀可以解决各式各样的问题，砍

瑞士刀可以用瑞士刀，切菜也可以用瑞士刀，但是它不一定是最好用的。因此针对各种不同的任务定制的模型往往比只用序列到序列模型的模型更好。谷歌 Pixel 4 手机用于语音识别的模型不是序列到序列模型，而是 RNN-Transducer 模型，这种模型是为了语音的某些特性所设计的，表现更好。

#### 4.2.5 句法分析

很多问题都可以用序列到序列模型来解，以句法分析（syntactic parsing）为例，如图 5.4 所示，给机器一段文字：比如“deep learning is very powerful”，机器要产生一个句法的分析树，即句法树（syntactic tree）。通过句法树告诉我们 deep 加 learning 合起来是一个名词短语，very 加 powerful 合起来是一个形容词短语，形容词短语加 is 以后会变成一个动词短语，动词短语加名词短语合起来是一个句子。

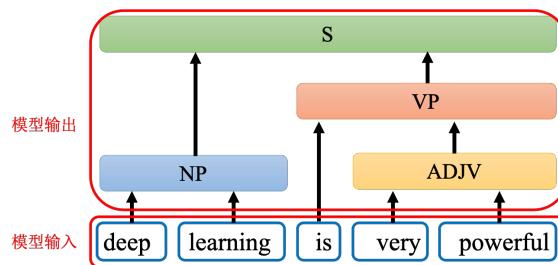


图 4.3 句法分析示例

在句法分析的任务中，输入是一段文字，输出是一个树状的结构，而一个树状的结构可以看成一个序列，该序列代表了这个树的结构，如图 5.5 所示。把树的结构转成一个序列以后，我们就可以用序列到序列模型来做句法分析，具体可参考论文“Grammar as a Foreign Language”<sup>[1]</sup>。这篇论文放在 arXiv 上面的时间是 14 年的年底，当时序列到序列模型还不流行，其主要只有被用在翻译上。因此这篇论文的标题才会取“Grammar as a Foreign Language”，其把句法分析看成一个翻译的问题，把语法当作是另外一种语言直接套用。

#### 4.2.6 多标签分类

多标签分类（multi-label classification）任务也可以用序列到序列模型。多类的分类跟多标签的分类是不一样的。如图 5.6 所示，在做文章分类的时候，同一篇文章可能属于多个类，文章 1 属于类 1 和类 3，文章 3 属于类 3、9、17。

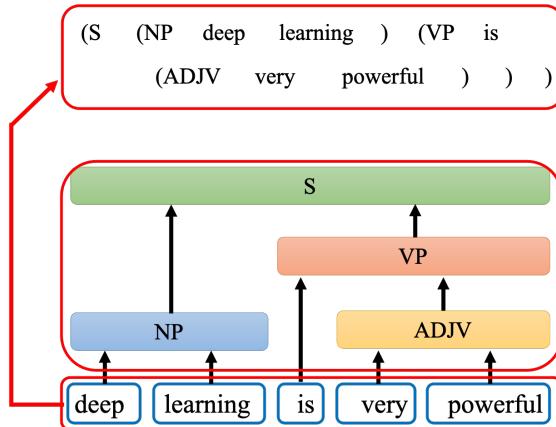


图 4.4 树状结构对应的序列

多分类问题 (multi-class classification) 是指分类的类别数大于 2。而多标签分类是指同一个东西可以属于多个类。

文章 1	文章 2	文章 3	文章 4
类 1	类 1	类 3	类 10
类 3		类 9	
		类 17	

图 4.5 多标签分分类示例

多标签分类 (multi-label classification) 问题不能直接把它当作一个多分类问题的问题来解。比如把这些文章丢到一个分类器 (classifier) 里面，本来分类器只会输出分数最高的答案，如果直接取一个阈值 (threshold)，只输出分数最高的前三名。这种方法是不可行的，因为每篇文章对应的类别的数量根本不一样。因此需要用序列到序列模型来做，如图 5.7 所示，输入一篇文章，输出就是类别，机器决定输出类别的数量。这种看起来跟序列到序列模型无关的问题也可以用序列到序列模型解，比如目标检测问题也可以用序列到序列模型来做，读者可参考论文 “End-to-End Object Detection with Transformers” [2]。



图 4.6 序列到序列模型来解多标签分类问题

### 4.3 Transformer 结构

一般的序列到序列模型会分成编码器和解码器，如图 5.8 所示。编码器负责处理输入的序列，再把处理好的结果“丢”给解码器，由解码器决定要输出的序列。

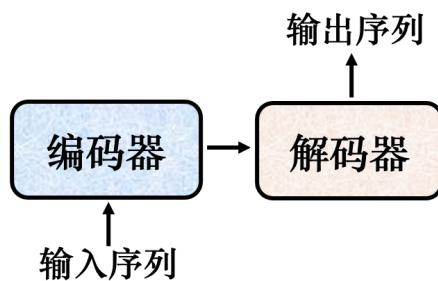


图 4.7 序列到序列模型结构

序列到序列模型的起源其实非常的早，在 14 年的 9 月就有一篇序列到序列模型用在翻译的论文：“Sequence to Sequence Learning with Neural Networks”<sup>[3]</sup>。序列到序列典型的模型就是 Transformer，其有一个编码器架构和一个解码器架构，如图 4.8 所示。

### 4.4 Transformer 编码器

接下来介绍下 Transformer 的编码器。如图 5.9 所示，编码器要做的事情就是给一排向量，输出另外一排向量。自注意力、循环神经网络（Recurrent Neural Network, RNN）、卷积神经网络都能输入一排向量，输出一排向量。Transformer 的编码器使用的是自注意力，输入一排向量，输出另外一个同样长度的向量。

如图 5.10 所示，编码器里面会分成很多的块（block），每一个块都是输入一排向量，输出一排向量。输入一排向量到第一个块，第一个块输出另外一排向量，以此类推，最后一个块会输出最终的向量序列。

Transformer 的编码器的每个块并不是神经网络的一层，每个块的结构如图 5.11 所示，在每个块里面，输入一排向量后做自注意力，考虑整个序列的信息，输出另外一排向量。接下来

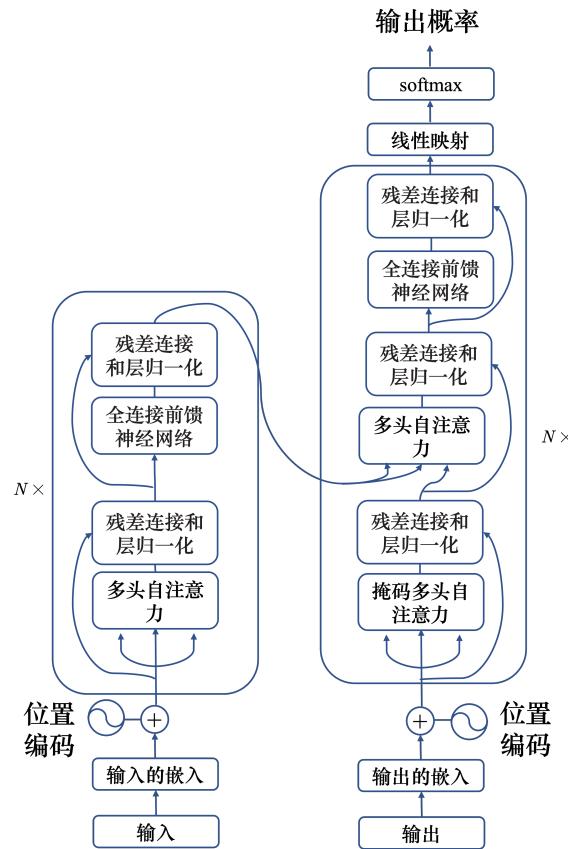


图 4.8 Transformer 结构

这排向量会“丢”到全连接网络里面，输出另外一排向量，这一排向量就是块的输出，事实上在原来的 Transformer 里面做的事情是更复杂的。

Transformer 里面加入了残差连接 (residual connection) 的设计，如图 5.12 所示，最左边的向量  $b$  输入到自注意力层后得到向量  $a$ ，输出向量  $a$  加上其输入向量  $b$  得到新的输出。得到残差的结果以后，再做层归一化 (layer normalization)。层归一化比批量归一化 (batch normalization) 更简单，不需要考虑批量 (batch) 的信息，而批量归一化需要考虑批量的信息。层归一化输入一个向量，输出另外一个向量。层归一化会计算输入向量的均值 (mean) 和标准差 (standard deviation)。

批量归一化是对不同样本 (example) 不同特征的同一个维度去计算均值跟标准差，但层归一化是对同一个特征、同一个样本里面不同的维度去计算均值跟标准差，接着做个归一化。输入向量  $x$  里面每一个维度减掉均值  $m$ ，再除以标准差  $\sigma$  以后得到  $x'$  就是层归一化的输出，如式 (4.1) 所示。得到层归一化的输出以后，该输出才是全连接网络的输入。输入到全连接网络，还有一个残差连接，把全连接网络的输入跟它的输出加起来得到新的输出。接着把残差的

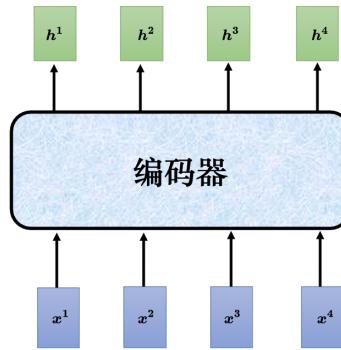


图 4.9 Transformer 编码器的功能

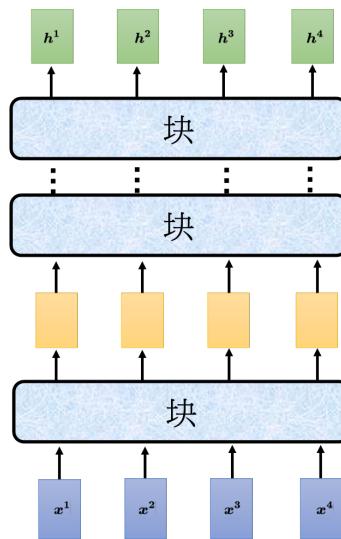


图 4.10 Transformer 编码器结构

结果再做一次层归一化得到的输出才是 Transformer 编码器里面一个块的输出。

$$x'_i = \frac{x_i - m}{\sigma} \quad (4.1)$$

图 4.13 给出了 Transformer 的编码器结构，其中  $N \times$  表示重复  $N$  次。首先，在输入的地方需要加上位置编码 (positional encoding)。如果只用自注意力，没有未知的信息，所以需要加上位置信息。多头注意力 (multi-head attention) 就是自注意力的块。经过自注意力后，还要加上残差连接 (residual connection) 和层归一化。接下来还要经过全连接的前馈神经网络，接着再做一次残差连接和层归一化，这才是一个块的输出，这个块会重复  $N$  次。Transformer 的编码器其实不一定要这样设计，论文 “On Layer Normalization in the Transformer Architecture” 提出了另一种设计，结果比原始的 Transformer 要好。原始的 Transformer 的架构并不是一个最优的设计，永远可以思考看看有没有更好的设计方式。

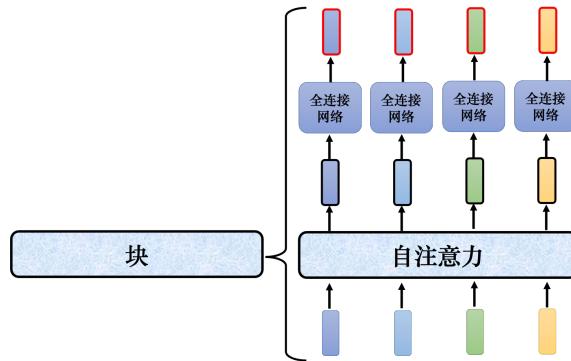


图 4.11 Transformer 编码器中每个块的结构

Q: 为什么 Transformer 中使用层归一化，而不使用批量归一化？

A: 论文 “PowerNorm: Rethinking Batch Normalization in Transformers” 解释了在 Transformers 里面批量归一化不如层归一化的原因，并提出能量归一化（power normalization）。能量归一化跟层归一化性能差不多，甚至好一点。

## 4.5 Transformer 解码器

### 4.5.1 自回归解码器

接下来要讲解解码器，解码器比较常见的叫做自回归的（autoregressive）解码器。以语音识别为例，输入一段声音，输出一串文字。如图 5.16 所示，把一段声音（“机器学习”）输入给编码器，输出会变成一排向量。接下来解码器产生语音识别的结果，解码器把编码器的输出先“读”进去。要让解码器产生输出，首先要先给它一个代表开始的特殊符号  $\langle \text{BOS} \rangle$ ，即 Begin Of Sequence，这是一个特殊的词元（token）。在词表（vocabulary）里面，在本来解码器可能产生的文字里面多加一个特殊的符号  $\langle \text{BOS} \rangle$ 。在机器学习里面，假设要处理自然语言处理的问题，每一个词元都可以用一个独热的向量来表示。独热向量其中一维是 1，其他都是 0，所以  $\langle \text{BOS} \rangle$  也是用独热向量来表示，其中一维是 1，其他是 0。接下来解码器会“吐”出一个向量，该向量的长度跟词表的大小是一样的。在产生这个向量之前，跟做分类一样，通常会先进行一个 softmax 操作。这个向量里面的分数是一个分布，该向量里面的值全部加起来，总和是 1。这个向量会给每一个中文字一个分，分数最高的中文字就是最终的输出。“机”的分数最高，所以“机”就当做是解码器的第一个输出。

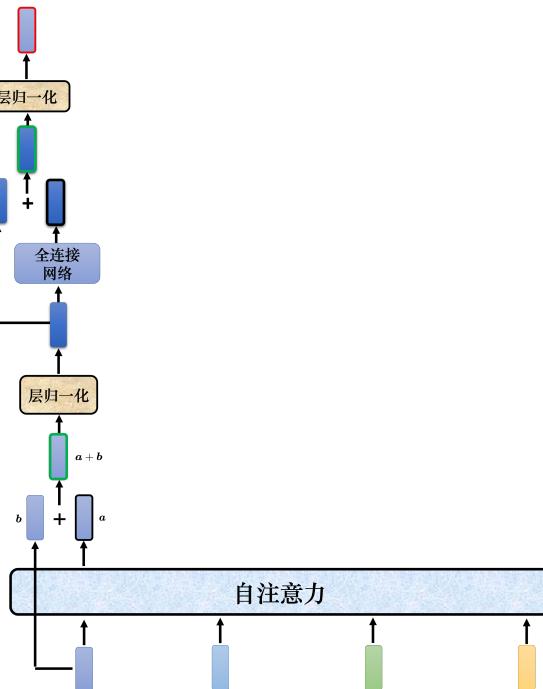


图 4.12 Transformer 中的残差连接

Q: 解码器输出的单位是什么?

A: 假设做的是中文的语音识别，解码器输出的是中文。词表的大小可能就是中文的方块字的数量。常用的中文的方块字大概两三千个，一般人可能认得的四、五千个，更多都是罕见字。比如我们认为解码器能够输出常见的 3000 个方块字就好了，就把它列在词表中。不同的语言，输出的单位不见不会不一样，这取决于对语言的理解。比如英语，选择输出英语的字母。但字母作为单位可能太小了，有人可能会选择输出英语的词汇，英语的词汇是用空白作为间隔的。但如果都用词汇当作输出又太多了，有一些方法可以把英语的字首、字根切出来，拿字首、字根当作单位。中文通常用中文的方块字来当作单位，这个向量的长度就跟机器可以输出的方块字的数量是一样多的。每一个中文的字都会对应到一个数值。

如图 5.17 所示，接下来把“机”当成解码器新的输入。根据两个输入：特殊符号 <BOS> 和“机”，解码器输出一个蓝色的向量。蓝色的向量里面会给出每一个中文字的分数，假设“器”的分数最高，“器”就是输出。解码器接下来会拿“器”当作输入，其看到了 <BOS>、“机”、“器”，可能就输出“学”。解码器看到 <BOS>、“机”、“器”、“学”，它会输出一个向量。这个向量里面“习”的分数最高的，所以它就输出“习”。这个过程就反复地持续下去。

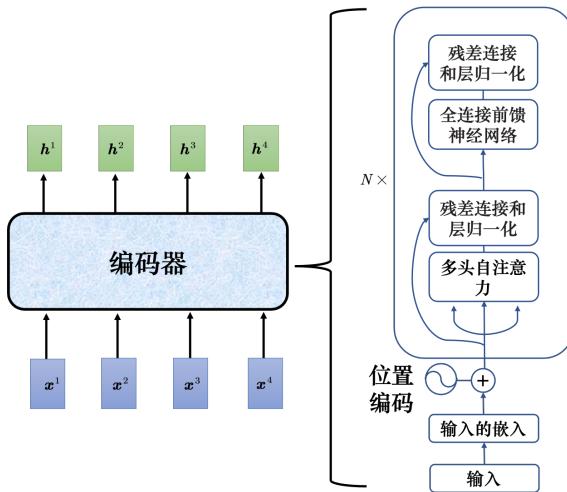


图 4.13 Transformer 编码器结构

解码器的输入是它在前一个时间点的输出，其会把自己的输出当做接下来的输入，因此当解码器在产生一个句子的时候，它有可能看到错误的东西。如图 4.16 所示，如果解码器有语音识别的错误，它把机器的“器”识别错成天气的“气”，接下来解码器会根据错误的识别结果产生它想要产生的期待是正确的输出，这会造成误差传播（error propagation）的问题，一步错导致步步错，接下来可能无法再产生正确的词汇。

Transformer 里面的解码器内部的结构如图 4.17 所示。类似于编码器，解码器也有多头注意力、残差连接和层归一化、前馈神经网络。解码器最后再做一个 softmax，使其输出变成一个概率。此外，解码器使用了掩蔽自注意力（masked self-attention），掩蔽自注意力可以通过一个掩码（mask）来阻止每个位置选择其后面的输入信息。

如图 5.18 所示，原来的自注意力输入一排向量，输出另外一排向量，这一排中每个向量都要看过完整的输入以后才做决定。根据  $a^1$  到  $a^4$  所有的信息去输出  $b^1$ 。掩蔽自注意力的不同点是不能再看右边的部分，如图 5.19 所示，产生  $b^1$  的时候，只能考虑  $a^1$  的信息，不能再考虑  $a^2$ 、 $a^3$ 、 $a^4$ 。产生  $b^2$  的时候，只能考虑  $a^1$ 、 $a^2$  的信息，不能再考虑  $a^3$ 、 $a^4$  的信息。产生  $b^3$  的时候，不能考虑  $a^4$  的信息。产生  $b^4$  的时候，可以用整个输入序列的信息。

一般自注意力产生  $b^2$  的过程如图 5.20 所示。掩蔽自注意力的计算过程如图 5.21 所示，我们只拿  $q^2$  和  $k^1$ 、 $k^2$  计算注意力，最后只计算  $v^1$  跟  $v^2$  的加权和。不管  $a^2$  右边的地方，只考虑  $a^1$ 、 $a^2$ 、 $q^1$ 、 $q^2$ 、 $k^1$  以及  $k^2$ 。输出  $b^2$  的时候，只考虑了  $a^1$  和  $a^2$ ，没有考虑到  $a^3$  和  $a^4$ 。

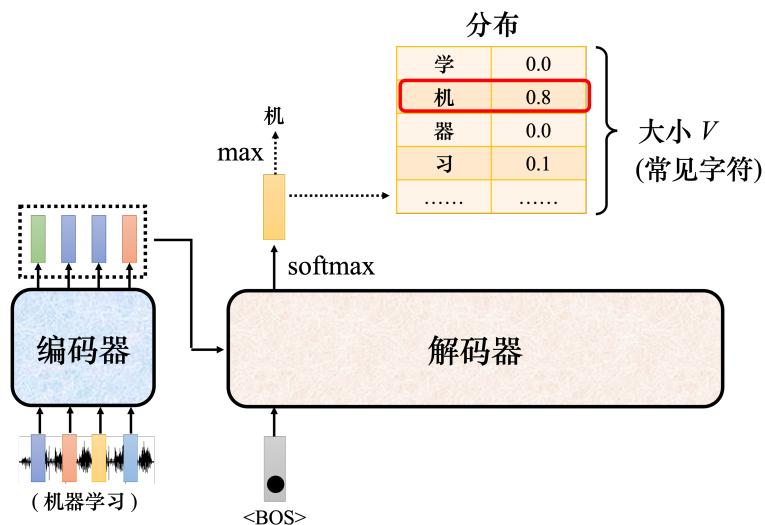


图 4.14 解码器的运作过程

Q: 为什么需要在注意力中加掩码?

A: 一开始解码器的输出是一个一个产生的，所以是先有  $a^1$  再有  $a^2$ ，再有  $a^3$ ，再有  $a^4$ 。这跟原来的自注意力不一样，原来的自注意力  $a^1$  跟  $a^4$  是一次整个输进去模型里面的。编码器是一次把  $a^1$  跟  $a^4$  都整个都读进去。但是对解码器而言，先有  $a^1$  才有  $a^2$ ，才有  $a^3$  才有  $a^4$ 。所以实际上当我们有  $a^2$ ，要计算  $b^2$  的时候，没有  $a^3$  跟  $a^4$  的，所以无法考虑  $a^3$   $a^4$ 。解码器的输出是一个一个产生的，所以只能考虑其左边的东西，没有办法考虑其右边的东西。

了解了解码器的运作方式，但这还有一个非常关键的问题：实际应用中输入跟输出长度的关系是非常复杂的，我们无法从输入序列的长度知道输出序列的长度，因此解码器必须决定输出的序列的长度。给定一个输入序列，机器可以自己学到输出序列的长度。但在目前的解码器运作的机制里面，机器不知道什么时候应该停下来，如图 5.22 所示，机器产生完“习”以后，还可以继续重复一模一样的过程，把“习”当做输入，解码器可能就会输出“惯”，接下来就一直持续下去，永远都不会停下来。

如图 5.23 所示，要让解码器停止运作，需要特别准备一个特别的符号  $<\text{EOS}>$ 。产生完“习”以后，再把“习”当作解码器的输入以后，解码器就要能够输出  $<\text{EOS}>$ ，解码器看到编码器输出的嵌入、 $<\text{BOS}>$ 、“机”、“器”、“学”、“习”以后，其产生出来的向量里面  $<\text{EOS}>$  的概率必须是最大的，于是输出  $<\text{EOS}>$ ，整个解码器产生序列的过程就结束了。

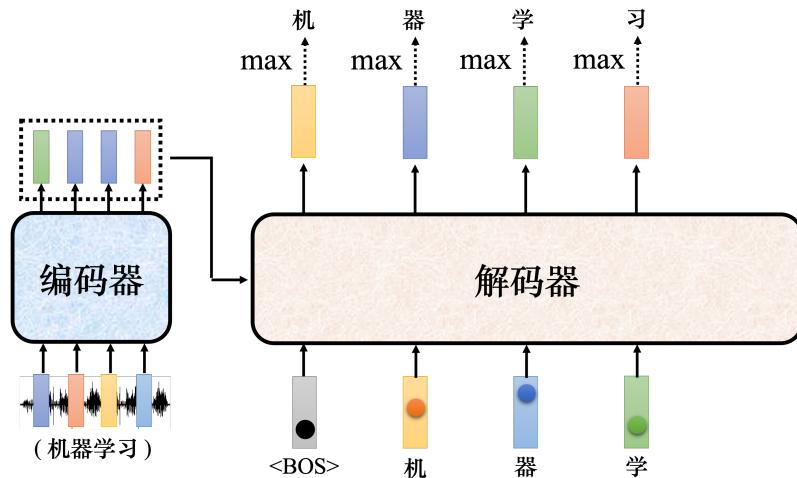


图 4.15 解码器示例

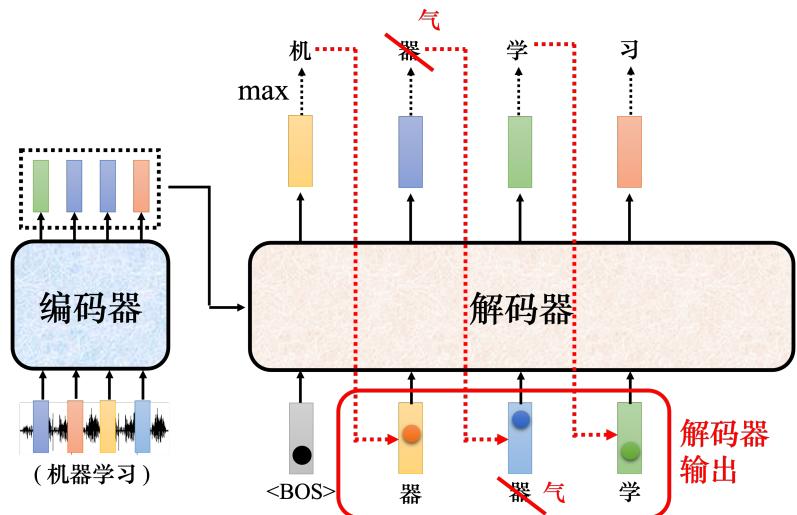


图 4.16 解码器中的误差传播

#### 4.5.2 非自回归解码器

接下来讲下非自回归 (non-autoregressive) 的模型。如图 5.26 所示，自回归的模型是先输入  $\langle \text{BOS} \rangle$ ，输出  $w_1$ ，再把  $w_1$  当做输入，再输出  $w_2$ ，直到输出  $\langle \text{EOS} \rangle$  为止。假设产生的是中文的句子，非自回归不是一次产生一个字，它是一次把整个句子都产生出来。非自回归的解码器可能“吃”的是一整排的  $\langle \text{BOS} \rangle$  词元，一次产生产生一排词元。比如输入 4 个  $\langle \text{BOS} \rangle$  的词元到非自回归的解码器，它就产生 4 个中文的字。因为输出的长度是未知的，所以当做非自回归解码器输入的  $\langle \text{BOS} \rangle$  的数量也是未知的，因此有如下两个做法。

- 用分类器 (classifier) 来解决这个问题。用分类器“吃”编码器的输入，输出是一个数字，

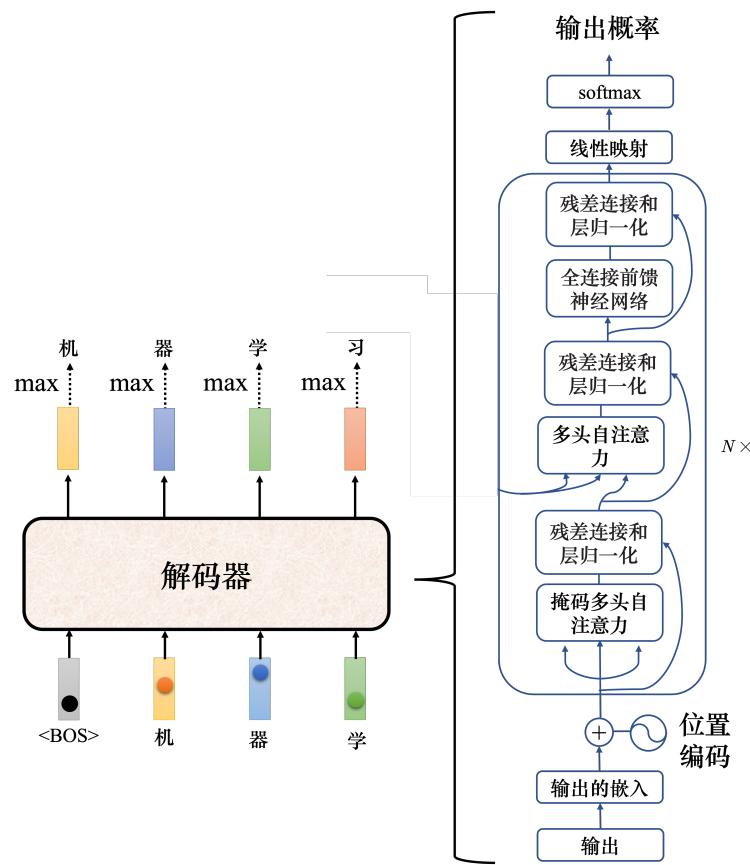


图 4.17 解码器内部结构

该数字代表解码器应该要输出的长度。比如分类器输出 4，非自回归的解码器就会“吃”4 个  $<\text{BOS}>$  的词元，产生 4 个中文的字。

- 给编码器一堆  $<\text{BOS}>$  的词元。假设输出的句子的长度有上限，绝对不会超过 300 个字。给编码器 300 个  $<\text{BOS}>$ ，就会输出 300 个字，输出  $<\text{EOS}>$  右边的输出就当它没有输出。

非自回归的解码器有很多优点。第一个优点是平行化。自回归的解码器输出句子的时候是一个一个字产生的，假设要输出长度一百个字的句子，就需要做一百次的解码。但是非自回归的解码器不管句子的长度如何，都是一个步骤就产生出完整的句子。所以非自回归的解码器会跑得比自回归的解码器要快。非自回归解码器的想法是在有 Transformer 以后，有这种自注意力的解码器以后才有的。以前如果用长短期记忆网络 (Long Short-Term Memory Network, LSTM) 或 RNN，给它一排  $<\text{BOS}>$ ，其无法同时产生全部的输出，其输出是一个一个产生的。

另外一个优点是非自回归的解码器比较能够控制它输出的长度。在语音合成里面，非自

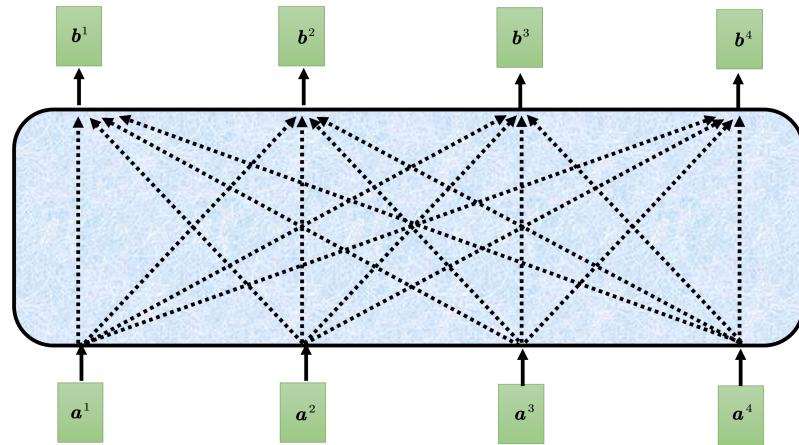


图 4.18 一般的自注意力示例

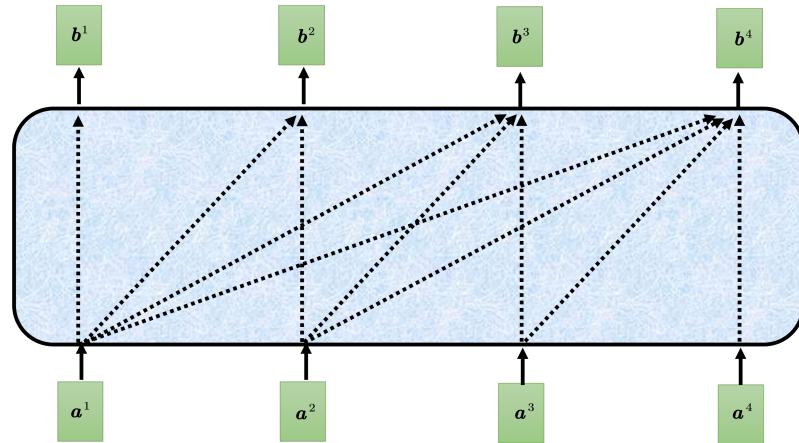


图 4.19 掩蔽自注意力示例

回归解码器算是非常常用的。非自回归的解码器可以控制输出的长度，可以用一个分类器决定非自回归的解码器应该输出的长度。在做语音合成的时候，如果想要让系统讲快一点，就把分类器的输出除以 2，系统讲话速度就变 2 倍快。如果想要讲话放慢速度，就把分类器输出的长度乘 2 倍，解码器说话的速度就变 2 倍慢。因此非自回归的解码器可以控制解码器输出的长度，做出种种的变化。

平行化是非自回归解码器最大的优势，但非自回归的解码器的性能（performance）往往都不如自回归的解码器。所以很多研究试图让非自回归的解码器的性能越来越好，去逼近自回归的解码器。要让非自回归的解码器跟自回归的解码器性能一样好，必须要使用非常多的技巧。

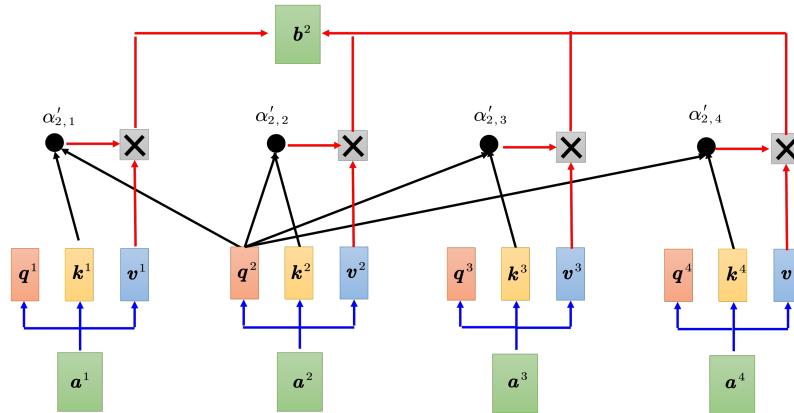


图 4.20 一般自注意力具体计算过程

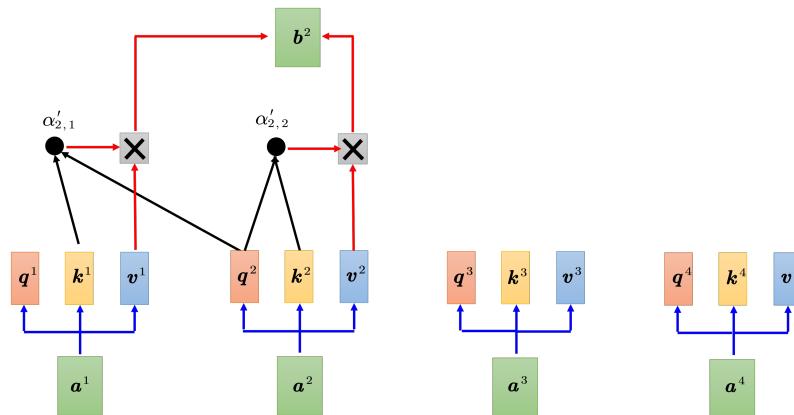


图 4.21 掩蔽自注意力具体计算过程

## 4.6 编码器-解码器注意力

编码器和解码器通过编码器-解码器注意力（encoder-decoder attention）传递信息，编码器-解码器注意力是连接编码器跟解码器之间的桥梁。如图 5.27 所示，解码器中编码器-解码器注意力的键和值来自编码器的输出，查询来自解码器中前一个层的输出。

接下来介绍下编码器-解码器注意力实际的运作过程。如图 5.28 所示，编码器输入一排向量，输出一排向量  $a^1, a^2, a^3$ 。接下来解码器会先“吃”`<BOS>`，经过掩蔽自注意力得到一个向量。接下来把这个向量乘上一个矩阵，做一个变换（transform），得到一个查询  $q$ ， $a^1, a^2, a^3$  也都产生键： $k^1, k^2, k^3$ 。把  $q$  跟  $k^1, k^2, k^3$  去计算注意力的分数，得到  $\alpha_1, \alpha_2, \alpha_3$ ，接下来做 softmax，得到  $\alpha'_1, \alpha'_2, \alpha'_3$ 。接下来通过式 (4.2) 可得加权和  $v$ 。

$$v = \alpha'_1 \times v^1 + \alpha'_2 \times v^2 + \alpha'_3 \times v^3 \quad (4.2)$$

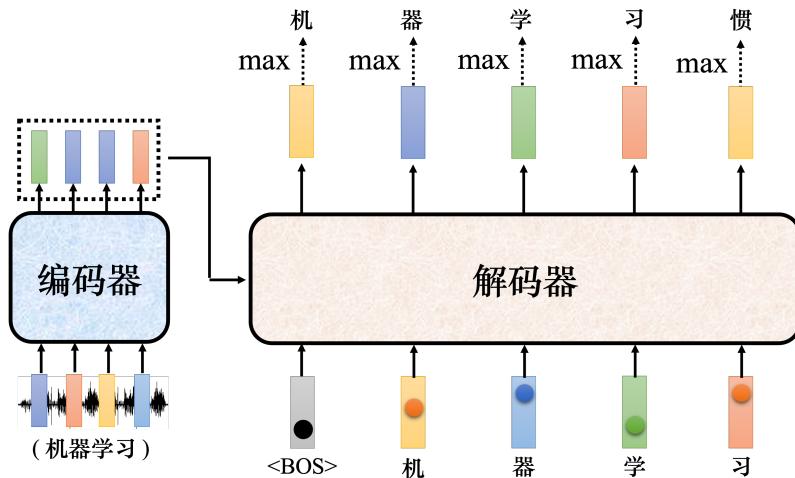


图 4.22 解码器运作的问题

$v$  接下来会“丢”给全连接网络，这个步骤  $q$  来自于解码器， $k$  跟  $v$  来自于编码器，该步骤就叫做编码器-解码器注意力，所以解码器就是凭借着产生一个  $q$ ，去编码器这边抽取信息出来，当做接下来的解码器的全连接网络（fully-connected network）的输入。

如图 5.29 所示，假设产生“机”，输入  $\text{<BOS>} \cdot \text{机}$ ，产生一个向量。这个向量一样乘上一个线性变换得到一个查询  $q'$ 。 $q'$  会跟  $k^1, k^2, k^3$  计算注意力的分数。接着用注意力分数跟  $v^1, v^2, v^3$  做加权和，加起来得到  $v'$ ，最后交给全连接网络处理。

编码器和解码器都有很多层，但在原始论文中解码器是拿编码器最后一层的输出。但不一定要这样，读者可参考论文“Rethinking and Improving Natural Language Generation with Layer-Wise Multi-View Decoding”<sup>[4]</sup>。

## 4.7 Transformer 的训练过程

如图 5.30 所示，Transformer 应该要学到听到“机器学习”的声音信号，它的输出就是“机器学习”这四个中文字。把  $\text{<BOS>}$  丢给编码器的时候，其第一个输出应该要跟“机”越接近越好。而解码器的输出是一个概率的分布，这个概率分布跟“机”的独热向量越接近越好。因此我们会去计算标准答案（ground truth）跟分布之间的交叉熵，希望该交叉熵的值越小越好。每一次解码器在产生一个中文字的时候做了一次类似分类的问题。假设中文字有四千个，就是做有四千个类别的分类的问题。

如图 5.32 所示，实际训练的时候，输出应该是“机器学习”。解码器第一次的输出、第二次的输出、第三次的输出、第四次输出应该分别就是“机”、“器”、“学”、“习”这四个中文字。

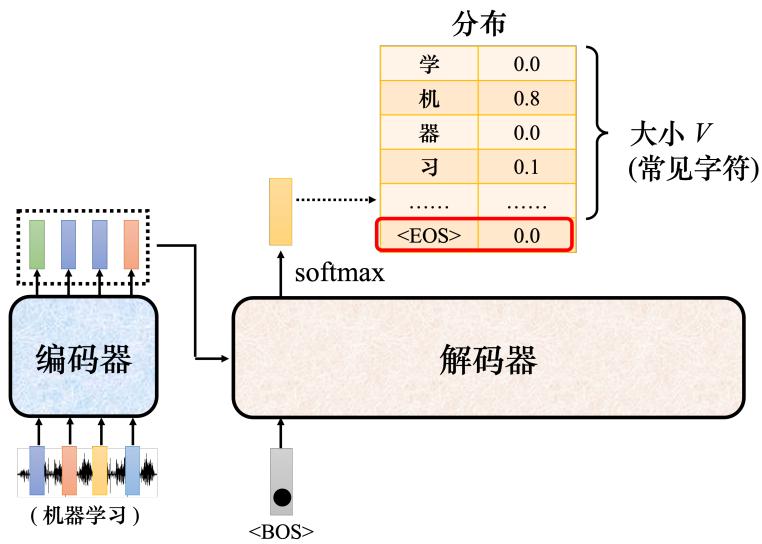


图 4.23 添加 &lt;EOS&gt; 词元

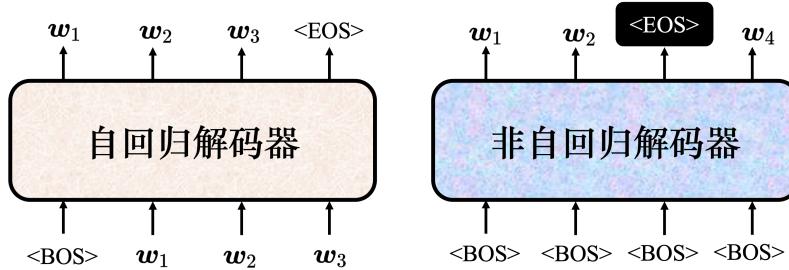


图 4.24 自回归解码器与非自回归解码器对比

的独热向量，输出跟这四个字的独热向量越接近越好。在训练的时候，每一个输出跟其对应的正确答案都有一个交叉熵。图 5.32 中做了四次分类问题，希望这些分类的问题交叉熵总和越小越好。训练的时候，解码器输出的不是只有“机器学习”这四个中文字，还要输出<EOS>。所以解码器的最终第五个位置输出的向量跟<EOS>的独热向量的交叉熵越小越好。我们把标准答案给解码器，希望解码器的输出跟正确答案越接近越好。在训练的时候，告诉解码器在已经有<BOS>、“机”的情况下，要输出“器”，有<BOS>、“机”、“器”的情况下输出“学”，有<BOS>、“机”、“器”、“学”的情况下输出“习”，有<BOS>、“机”、“器”、“学”、“习”的情况下，输出<EOS>。在解码器训练的时候，在输入的时候给它正确的答案，这称为教师强制（teacher forcing）。

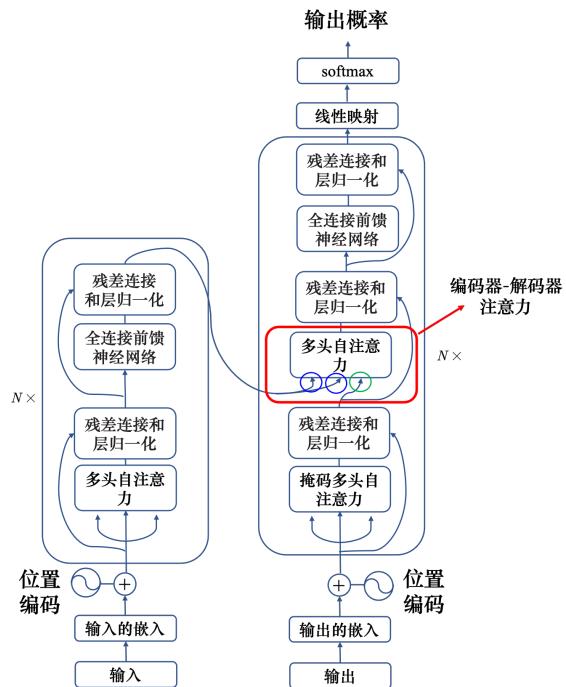


图 4.25 编码器-解码器注意力

## 4.8 序列到序列模型训练常用技巧

接下来介绍下训练序列到序列模型的一些技巧。

### 4.8.1 复制机制

第一个技巧是复制机制 (copy mechanism)。对很多任务而言，解码器没有必要自己创造输出，其可以从输入的东西里面复制一些东西。以聊天机器人为例，用户对机器说：“你好，我是库洛洛”。机器应该回答：“库洛洛你好，很高兴认识你”。机器其实没有必要创造“库洛洛”这个词汇，“库洛洛”对机器来说一定会是一个非常怪异的词汇，所以它可能很难在训练数据里面出现，可能一次也没有出现过，所以它不太可能正确地产生输出。但是假设机器在学的时候，学到的并不是它要产生“库洛洛”，它学到的是看到输入的时候说“我是某某某”，就直接把“某某某”复制出来，说“某某某你好”。这种机器的训练会比较容易，显然比较有可能得到正确的结果，所以复制对于对话任务可能是一个需要的技术。机器只要复述这一段它听不懂的话，它不需要从头去创造这一段文字，它要学的是从用户的输入去复制一些词汇当做输出。

在做摘要的时候，我们可能更需要复制的技巧。做摘要需要搜集大量的文章，每一篇文章都有人写的摘要，训练一个序列到序列的模型就结束了。要训练机器产生合理的句子，通常需

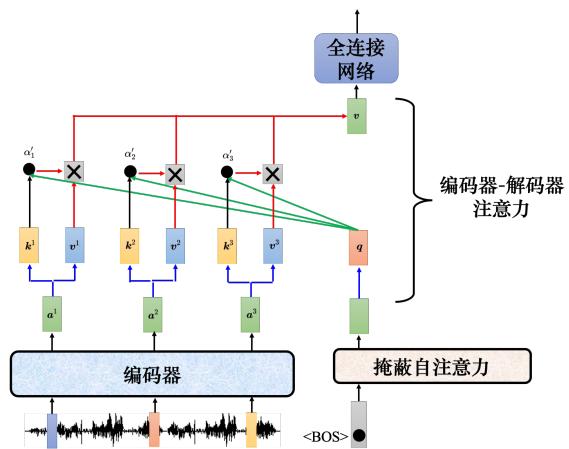


图 4.26 编码器-解码器注意力运作过程

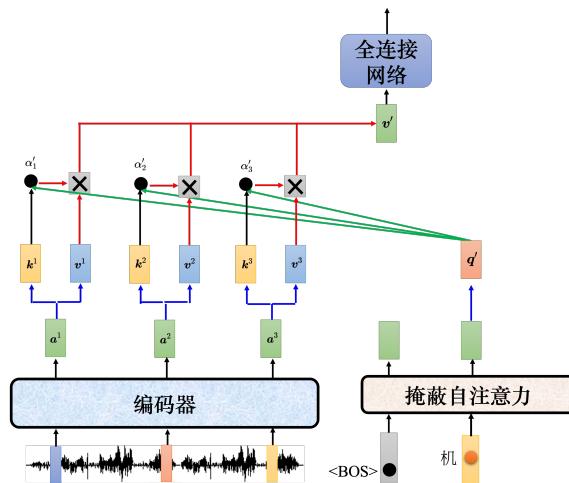


图 4.27 编码器-解码器注意力运作过程示例

要百万篇文章，这些文章都要有人标的摘要。在做摘要的时候，很多的词汇就是直接从原来的文章里面复制出来的，所以对摘要任务而言，从文章里面直接复制一些信息出来是一个很关键的能力，最早有从输入复制东西的能力的模型叫做指针网络（pointer network），后来还有一个变形叫做复制网络（copy network）。

#### 4.8.2 引导注意力

序列到序列模型有时候训练出来会产生莫名其妙的结果。以语音合成为例，机器念 4 次的“发财”，重复 4 次没问题，但叫它只念一次“发财”，它把“发”省略掉只念“财”。也许在训练数据里面，这种非常短的句子很少，所以机器无法处理这种非常短的句子。这个例子并没有常出现，用序列到序列学习出来，语音合成没有这么差。类似于语音识别、语音合成这种任

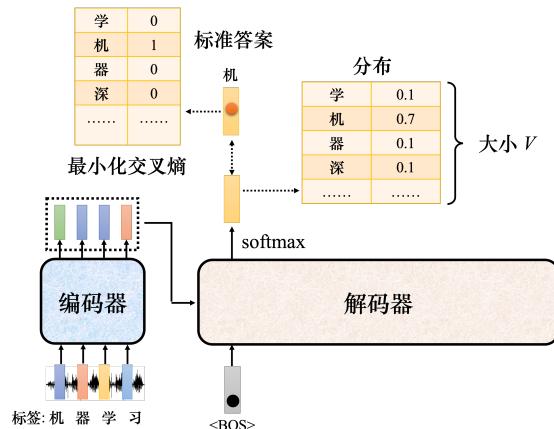


图 4.28 Transformer 的训练过程

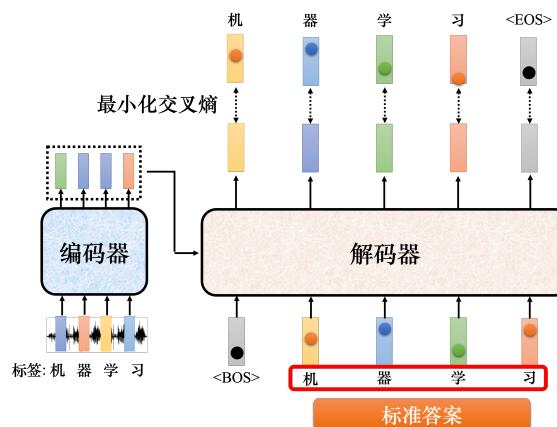


图 4.29 教师强制

务最适合使用引导注意力。因为像语音识别，很难接受，我们讲一句话，识别出来居然有一段机器没听到。或者像语音合成这种任务，输入一段文字，语音合出来居然有一段没有念到。引导注意力要求机器在做注意力的时候有固定的方式。对语音合成或语音识别，我们想像中的注意力应该就是由左向右。如图 5.33 所示，红色的曲线来代表注意力的分数，越高就代表注意力的值越大。以语音合成为例，输入就是一串文字，合成声音的时候，显然是由左念到右。所以机器应该是先看最左边输入的词汇产生声音，再看中间的词汇产生声音，再看右边的词汇产生声音。如果做语音合成的时候，机器的注意力是颠三倒四的，它先看最后面，接下来再看前面，再胡乱看整个句子，显然这样的注意力是有问题的，没有办法合出好的结果。因此引导注意力会强迫注意力有一个固定的样貌，如果我们对这个问题本身就已经有理解，知道对于语音合成这样的问题，注意力的位置都应该由左向右，不如就直接把这个限制放进训练里面，要求机器学到注意力就应该要由左向右。

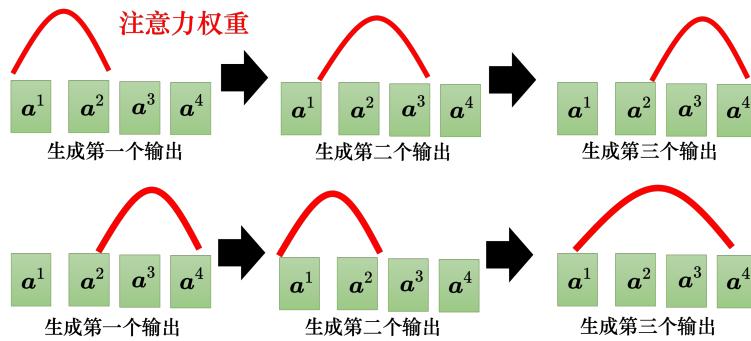


图 4.30 引导注意力

### 4.8.3 束搜索

如图 5.34 所示，假设解码器就只能产生两个字 A 和 B，假如世界上只有两个字 A 跟 B，即词表  $\mathcal{V} = \{A, B\}$ 。对解码器而言，每一次在第一个时间步 (time step)，它在 A、B 里面决定一个。比如解码器可能选 B 当作输入，再从 A、B 中选一个。在上文中，每一次解码器都是选分数最高的那一个。假设 A 的分数是 0.6，B 的分数是 0.4，解码器的第一次就会输出 A。接下来假设 B 的分数为 0.6，A 的分数为 0.4，解码器就会输出 B。再假设把 B 当做输入，现在输入已经有 A、B，接下来 A 的分数是 0.4，B 的分数是 0.6，解码器就会选择输出 B。因此输出就是 A、B、B。这种每次找分数最高的词元来当做输出的方法称为贪心搜索 (greedy search)，其也被称为贪心解码 (greedy decoding)。红色路径就是通过贪心解码得到的路径。

但贪心搜索不一定是最好的方法，第一步可以先稍微舍弃一点东西，第一步虽然 B 是 0.4，但先选 B。选了 B，第二步时 B 的可能性就大增就变成 0.9。到第三步时，B 的可能性也是 0.9。绿色路径虽然第一步选了一个较差的输出，但是接下来的结果是好的。比较下红色路径与绿色路径，红色路径第一步好，但全部乘起来是比较差的，绿色路径一开始比较差，但最终结果其实是比较好的。

如何找到最好的结果是一个值得考虑的问题。穷举搜索 (exhaustive search) 是最容易想到的方法，但实际上并没有办法穷举所有可能的路径，因为每一个转折点的选择太多了。对中国而言，中文有 4000 个字，所以树每一个地方的分叉都是 4000 个可能的路径，走两三步以后，就会无法穷举。

接下来介绍下束搜索 (beam search)，束搜索经常也称为集束搜索或柱搜索。束搜索是用比较有效的方法找一个近似解，在某些情况下效果不好。比如论文 “The Curious Case Of Neural Text Degeneration” [5]。这个任务要做的事情是完成句子 (sentence completion)，也

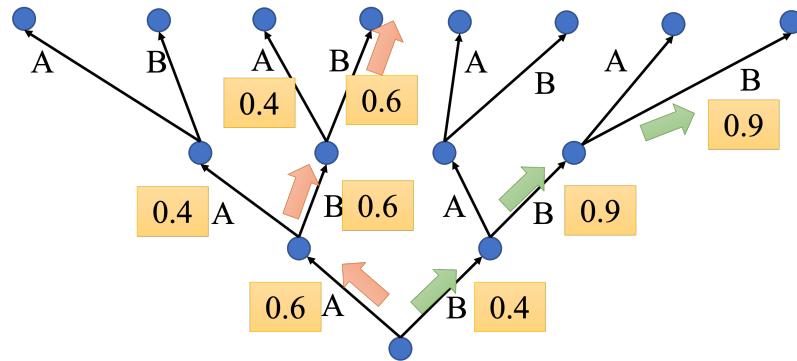


图 4.31 解码器搜索示例

就是机器先读一段句子，接下来它要把这个句子的后半段完成，如果用束搜索，会发现说机器不断讲重复的话。如果不用束搜索，加一些随机性，虽然结果不一定完全好，但是看起来至少是比较正常的句子。有时候对解码器来说，没有找出分数最高的路，反而结果是比较好的，这个就是要看任务本身的特性。假设任务的答案非常明确，比如语音识别，说一句话，识别的结果就只有一个可能。对这种任务而言，通常束搜索就会比较有帮助。但如果任务需要机器发挥一点创造力，束搜索比较没有帮助。

#### 4.8.4 加入噪声

在做语音合成的时候，解码器加噪声，这是完全违背正常的机器学习的做法。在训练的时候会加噪声，让机器看过更多不同的可能性，这会让模型比较鲁棒，比较能够对抗它在测试的时候没有看过的状况。但在测试的时候居然还要加一些噪声，这不是把测试的状况弄得更困难，结果更差。但语音合成神奇的地方是，模型训练好以后。测试的时候要加入一些噪声，合出来的声音才会好。用正常的解码的方法产生出来的声音听不太出来是人声，产生出比较好的声音是需要一些随机性的。对于语音合成或句子完成任务，解码器找出最好的结果不一定是人类觉得最好的结果，反而是奇怪的结果，加入一些随机性的结果反而会是比较好的。

#### 4.8.5 使用强化学习训练

接下来还有另外一个问题，我们评估的标准用的是 BLEU (BiLingual Evaluation Under-study) 分数。虽然 BLEU 最先是用于评估机器翻译的结果，但现在它已经被广泛用于评价许多应用输出序列的质量。解码器先产生一个完整的句子，再去跟正确的答案一整句做比较，拿两个句子之间做比较算出 BLEU 分数。但训练的时候，每一个词汇是分开考虑的，最小化的

是交叉熵，最小化交叉熵不一定可以最大化 BLEU 分数。但在做验证的时候，并不是挑交叉熵最低的模型，而是挑 BLEU 分数最高的模型。一种可能的想法：训练的损失设置成 BLEU 分数乘一个负号，最小化损失等价于最大化 BLEU 分数。但 BLEU 分数很复杂，如果要计算两个句子之间的 BLEU 分数，损失根本无法做微分。我们之所以采用交叉熵，而且是每一个中文的字分开来算，就是因为这样才有办法处理。遇到优化无法解决的问题，可以用强化学习训练。具体来讲，遇到无法优化的损失函数，把损失函数当成强化学习的奖励，把解码器当成智能体，可参考论文“Sequence Level Training with Recurrent Neural Networks”。

#### 4.8.6 计划采样

如图 4.32 所示，测试的时候，解码器看到的是自己的输出，因此它会看到一些错误的东西。但是在训练的时候，解码器看到的是完全正确的，这种不一致的现象叫做曝光偏差（exposure bias）。

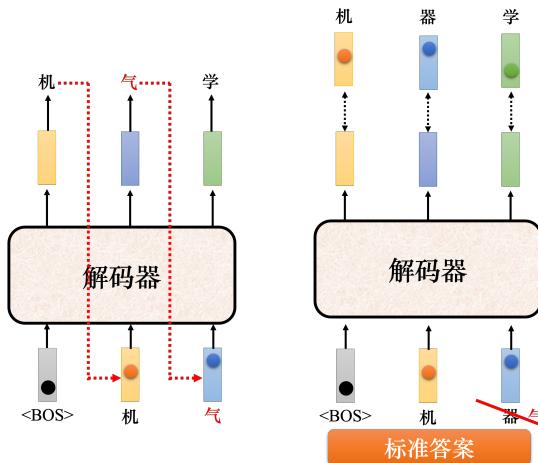


图 4.32 曝光偏差

假设解码器在训练的时候永远只看过正确的东西，在测试的时候，只要有一个错，就会一步错步步错。因为解码器从来没有看过错的东西，它看到错的东西会非常的惊奇，接下来它产生的结果可能都会错掉。有一个可以的思考的方向是：给解码器的输入加一些错误的东西，不要给解码器都是正确的答案，偶尔给它一些错的东西，它反而会学得更好，这一招叫做计划采样（scheduled sampling）<sup>[6]</sup>，它不是学习率调整（schedule learning rate）。很早就有计划采样，在还没有 Transformer、只有 LSTM 的时候，就已经有计划采样。但是计划采样会伤害到 Transformer 的平行化的能力，所以 Transformer 的计划采样另有招数，其跟原来最

早提在这个 LSTM 上被提出来的招数也不太一样。读者可参考论文“Scheduled Sampling for Transformers”<sup>[7]</sup>、“Parallel Scheduled Sampling”<sup>[8]</sup>。

## 参考文献

- [1] VINYALS O, KAISER Ł, KOO T, et al. Grammar as a foreign language[J]. Advances in neural information processing systems, 2015, 28.
- [2] CARION N, MASSA F, SYNNAEVE G, et al. End-to-end object detection with transformers[C]//European conference on computer vision. Springer, 2020: 213-229.
- [3] SUTSKEVER I, VINYALS O, LE Q V. Sequence to sequence learning with neural networks[J]. Advances in neural information processing systems, 2014, 27.
- [4] LIU F, REN X, ZHAO G, et al. Rethinking and improving natural language generation with layer-wise multi-view decoding[J]. arXiv preprint arXiv:2005.08081, 2020.
- [5] HOLTZMAN A, BUYS J, DU L, et al. The curious case of neural text degeneration[J]. arXiv preprint arXiv:1904.09751, 2019.
- [6] BENGIO S, VINYALS O, JAITLEY N, et al. Scheduled sampling for sequence prediction with recurrent neural networks[C]//volume 28. 2015.
- [7] MIHAYLOVA T, MARTINS A F. Scheduled sampling for transformers[J]. arXiv preprint arXiv:1906.07651, 2019.
- [8] DUCKWORTH D, NEELAKANTAN A, GOODRICH B, et al. Parallel scheduled sampling[J]. arXiv preprint arXiv:1906.04331, 2019.

## 第 5 章 自监督学习

### 5.1 自监督学习基础

本章对自监督学习 (Self-Supervised Learning, SSL) 进行介绍。首先介绍下监督学习，如果在模型训练期间使用标注的数据，则称之为监督学习。如果没有使用标注的数据，则称之为无监督学习。如图 5.1a 所示，在监督学习中只有一个模型，模型的输入是  $x$ ，输出是  $\hat{y}$ ，标签是  $y$ 。对于情感分析 (sentiment analysis)，监督学习就是让机器读一篇文章，机器需要对文章是正面还是负面进行分类。我们需要有标注的数据，先找到很多文章并对所有文章进行标注。根据文章的含义将其标注为正面或负面，正面或负面就是标签。

我们需要有标注的文章数据来训练监督模型，而自监督学习是一种没有标注的学习方式。如图 5.1b 所示，假设我们有未标注的文章数据，则可将一篇文章  $x$ ，将  $x$  分为两部分：模型的输入  $x'$  和模型的标签  $x''$ ，将  $x'$  输入模型并让它输出  $\hat{y}$ ，想让  $\hat{y}$  尽可能地接近它的标签  $x''$ （学习目标），这就是自监督学习。Yann LeCun 最初在 2019 年 4 月在 Facebook（后改名为 Meta）上的一篇帖子上提出了“自监督学习”这个词。

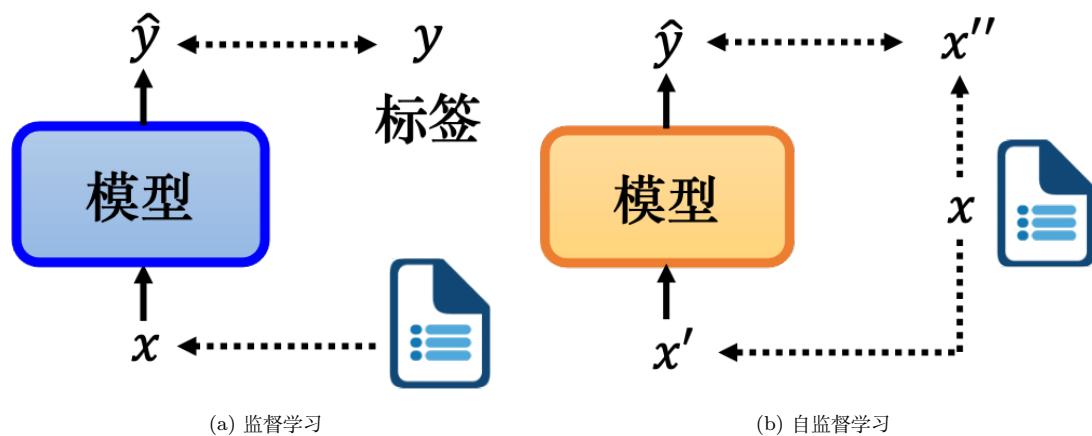


图 5.1 监督学习和自监督学习

由于自监督学习不使用标注的数据，因此自监督学习可以看作是一种无监督学习方法。为什么不直接称其为无监督学习？因为无监督学习是一个比较大的家族，里面有很多不同的方法，自监督学习只是其中之一。为了使定义更清晰，称其为自监督学习。

自监督学习的模型大多都是以芝麻街的角色命名，来让其名称缩写“凑”成电视节目《芝麻街》中的角色，以下是几个例子。

- **ELMo: 来自语言模型的嵌入 (Embeddings from Language Modeling)**, 名称来自《芝麻街》的红色小怪兽 Elmo, ELMo 是最早的自监督学习的模型;
- **BERT: 来自 Transformers 的双向编码器表示 (Bidirectional Encoder Representation from Transformers)**, 名称来自《芝麻街》的另一个角色 Bert;
- BERT 提出后, 马上就出现了两个不同的模型, 都叫 ERNIE, 一个模型是**知识增强的语言表示模型 (Enhanced Representation through Knowledge Integration)**, 另一个模型是**具有信息实体的增强语言表示 (Enhanced Language Representation with Informative Entities)**, 名称来自 Bert 最好的朋友 Ernie;
- **Big Bird: 较长序列的 Transformer (Transformers for Longer Sequences, Big Bird)**, 名称来自《芝麻街》的黄色大鸟 Big Bird.

如表 5.1 所示, 自监督模型的参数都很大. Megatron 的参数量是生成式预训练-2 (Generative Pre Training-2, GPT-2) 的 8 倍左右.GPT-3 的参数量是 Turing NLG 的 10 倍. 目前最大的模型是谷歌的 Switch Transformer, 其参数量比 GTP-3 大了 10 倍.

表 5.1 自监督模型的参数量

模型	参数量 (M)
ELMo	94
BERT	340
GPT-2	1542
Megatron	8000
T5	11000
Turing NLG	17000
GPT-3	175000
Switch Transformer	1600000 (1.6T)

这里我们主要介绍两种典型的自监督学习模型: BERT 和 GPT.

## 5.2 来自 Transformers 的双向编码器表示 (BERT)

BERT 模型是自监督学习的经典模型，如图 5.2 所示，BERT 是一个 Transformer 的编码器，BERT 的架构与 Transformer 的编码器完全相同，里面有很多自注意力和残差连接 (residual connection)、归一化等等。BERT 可以输入一行向量，输出另一行向量。输出的长度与输入的长度相同。

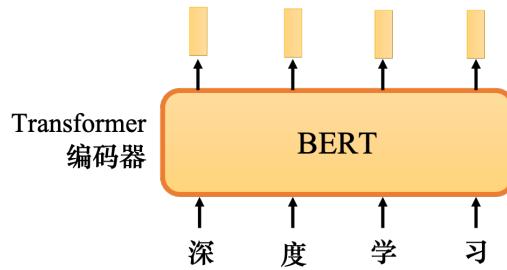


图 5.2 BERT 的架构

BERT 一般用在自然语言处理中，用在文本场景中，所以一般它的输入是一个文本序列，也是一个数据序列。不仅文本是一种序列，语音也可以看作是一种序列，甚至图像也可以看作是一堆向量。因此 BERT 不仅可以用在自然语言处理中，也用在文本中，它还可以用于语音和视频。因为 BERT 最早是用在文本中，所以这里都以文本为例（语音或图像也都是一样的）。BERT 的输入是一段文字。接下来需要随机掩码 (mask) 一些输入文字，被掩码的部分是随机决定的。例如，输入 100 个词元 (token)。什么是词元？词元是处理一段文本时的基本单位，词元的单位大小由我们自己决定。在中文文本中，通常将一个汉字当成一个词元。当输入一个句子时，里面有一些单词会被随机掩码。哪些部分需要掩码？它是随机决定的。

有两种方法来实现掩码，如图 5.3 所示。第一种方法是用特殊符号替换句子中的单词，使用“MASK”词元来表示特殊符号，可以将其看成一个新的汉字，它不在字典里，它的意思是掩码原文。掩码的目的是对向量中某些值进行掩盖，避免无关位置的数值对运算造成影响。另一种方法是用另一个字随机替换一个字。本来是“度”字，可以随机选择另一个汉字来替换它，比如改成“一” / “天” / “大” / “小”，只是用随机选择的某个字替换它。所以有两种方法可以做掩码：

- 添加一个名为“MASK”的特殊词元；
- 用另一个词替换某个词。

这两种方法都可以使用，使用哪种方法也是随机确定的。所以在 BERT 训练的时候，应该给

BERT 输入一个句子，首先随机决定要掩码哪些汉字，之后，再决定如何进行掩码。掩码部分是要被特殊符号“MASK”代替还是只是被另一个汉字代替？这两种方法都可以使用。

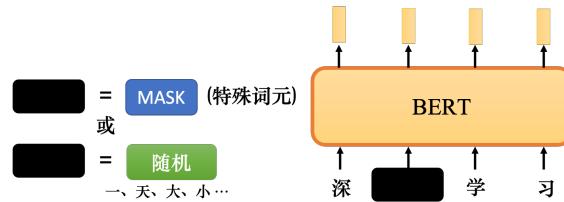


图 5.3 掩码的两种方法

如图 5.4 所示，掩码后，向 BERT 输入了一个序列，BERT 的相应输出就是另一个序列。接下来，查看输入序列中掩码部分的对应输出，仍然在掩码部分输入汉字，它可能是“MASK”词元或随机单词，它仍然输出一个向量，对这个向量使用线性变换（线性变换是指输入向量会乘以一个矩阵）。然后做 softmax 并输出一个分布。输出是一个很长的向量，包含要处理的每个汉字。每个字对应一个分数，它是通过 softmax 函数生成的分布。

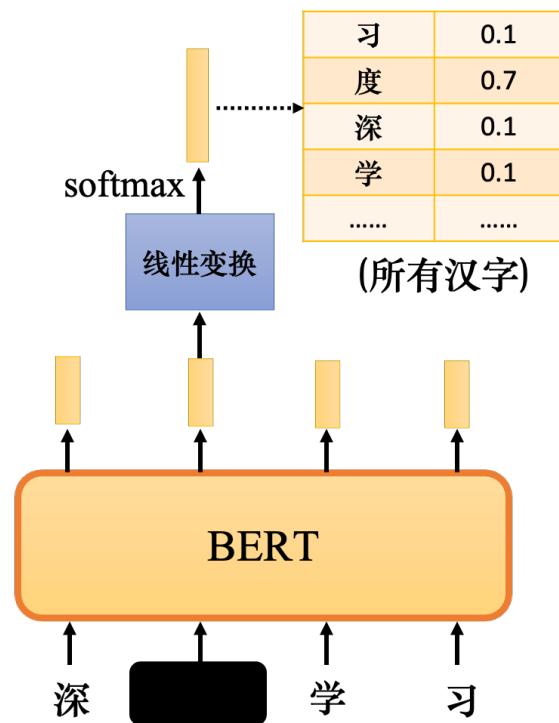


图 5.4 BERT 预测过程

如何训练 BERT 模型？如图 5.5 所示，我们知道被掩码字符是哪个字符，而 BERT 不知道。因为把句子交给 BERT 时，该字符被掩码了，所以 BERT 不知道该字符，但我们知道掩

码字符“深度”一词中的“度”. 因此, 训练的目标是输出一个尽可能接近真实答案的字符, 即“度”字符. 独热 (one-hot) 向量可以用来表示字符, 并最小化输出和独热向量之间的交叉熵损失. 这个问题可以看成一个分类问题, 只是类的数量和汉字的数量一样多. 如果汉字的数量大约在 4000 左右, 该问题就是一个 4000 类的分类问题. BERT 要做的就是成功预测掩码的地方属于的类别, 在这个例子里, 就是“度”类别. 在训练过程中, 在 BERT 之后添加一个线性模型并将它们一起训练. 所以, BERT 内部是一个 Transformer 的编码器, 它有一堆参数. 线性模型是一个矩阵, 它也有一些参数, 尽管与 BERT 相比, 其数量要少得多. 我们需要联合训练 BERT 和线性模型并尝试预测被掩码的字.

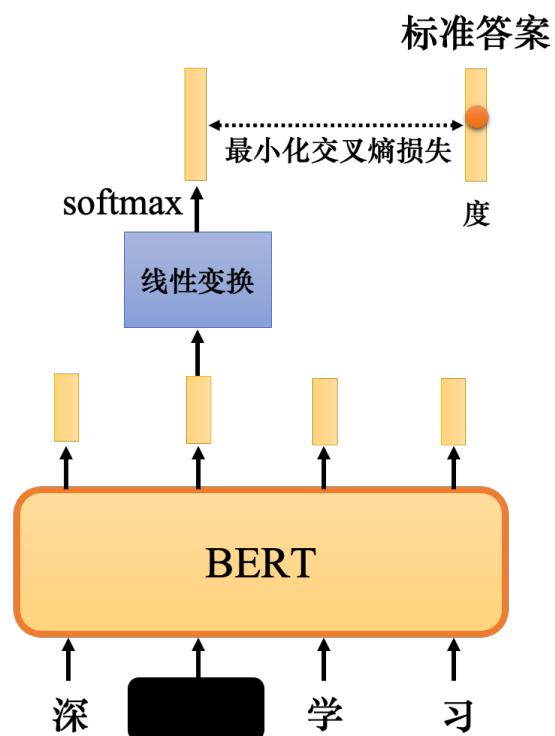


图 5.5 BERT 的训练过程

事实上, 训练 BERT 时, 除了掩码之外, 还有另一种方法: **下一句预测 (next sentence prediction)**. 我们可以通过在互联网上使用爬虫来获得的大量句子来构建数据库, 然后从数据库中拿出两个句子. 如图 5.6 所示, 这两个句子中间加入了一个特殊的词元 [SEP] 来代表它们之间的分隔. 这样, BERT 就可以知道这两个句子是不同的句子, 因为这两个句子之间有一个分隔符号. 我们还将会在整个序列的最前面加入一个特殊词元分类符号 [CLS].

现在给定一个很长的序列, 其中包括两个句子, 中间有个 [SEP] 词元, 前面有一个 [CLS]

词元。如果将这个很长的序列输入到 BERT，它应该输出一个序列，按理说，输入是一个序列，输出应该是另外一个序列，这是编码器可以做的事情。而 BERT 就是一个 Transformer 的编码器，所以 BERT 可以做这件事。我们只取与 [CLS] 对应的输出，忽略其他输出，并将 [CLS] 的输出乘以线性变换。现在它做一个二元分类问题，它有两个可能的输出：是或否。这种方法是下一句预测，即需要预测第二句是否是第一句的后一句（这两个句子是不是相接的）。如果第二句确实是后续句子（这两个句子是相接的），就要训练 BERT 输出“是”。当第二句不是后一句时（这两个句子不是相接的），BERT 需要输出“否”作为预测。

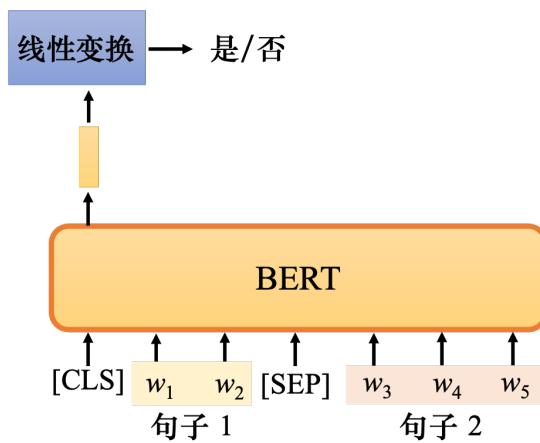


图 5.6 下一句预测

但后来的研究发现，下一句预测对 BERT 将要完成的任务并没有真正的帮助。有一篇题为“Robustly Optimized BERT Approach (RoBERTa)”的论文明确指出使用下一句预测方法几乎没有帮助，之后，这个想法以某种方式成为主流。紧接着，另一篇论文说下一句预测没用，后来又有很多论文开始说它也没用，比如 SCAN-BERT 和 XLNet。下一句预测没用的可能原因之一是下一句预测这个任务太简单了，这是一项容易的任务，预测两个句子是否相接并不是一项特别困难的任务。此任务的通常方法是首先随机选择一个句子，然后从数据库中随机选择将要连接到前一个句子的句子。通常，随机选择一个句子时，它很可能与之前的句子有很大不同。对于 BERT 来说，预测两个句子是否相接并不难。因此，在训练 BERT 完成下一句预测任务时，没有学到太多有用的东西。

还有一种类似于下一句预测的方法——句序预测 (Sentence Order Prediction, SOP)，其在文献上似乎更有用。这种方法的主要思想是最初选择的两个句子本来就是连接在一起，可能有两种可能：要么句子 1 连接在句子 2 后面，要么句子 2 连接在句子 1 后面，有两种可能性，

BERT 要回答是哪一种可能性. 或许是因为这个任务难度更大, 所以句序预测似乎更有效. 它被用于名为 ALBERT 的模型中, 该模型是 BERT 的进阶版本.

### 5.2.1 BERT 的使用方式

如何使用 BERT? 在训练时, 让 BERT 学习两个任务.

- 把一些字符掩盖起来, 让它做填空题, 补充掩码的字符.
- 预测两个句子是否有顺序关系 (两个句子是否应该接在一起), 这个任务没什么用.

总之, BERT 学会了如何填空. 但如果我要解决的任务, 不是填空题怎么办? 如图 5.7 所示, 它的神奇之处在于, 在训练模型填空后, 它可以用于其他任务. 这些任务不一定与填空有关, 它可能是完全不同的东西. 尽管如此, BERT 仍然可以用于这些任务. 这些任务是真正使用 BERT 的任务, 其称为**下游任务 (downstream task)**. 下游任务就是我们实际关心的任务. 但当 BERT 学习完成这些任务时, 仍然需要一些标注的数据. 总之, BERT 只是学会了填空, 但是, 后来它可以用来做各种感兴趣的下游任务. 这就像胚胎中的干细胞, 胚胎干细胞可以分化成各种不同的细胞, 比如心脏细胞、肌肉细胞等等. BERT 的能力还没有发挥出来, 它具有各种无限的潜力, 虽然它只会做填空题, 但后来它具有解决各种任务的能力.

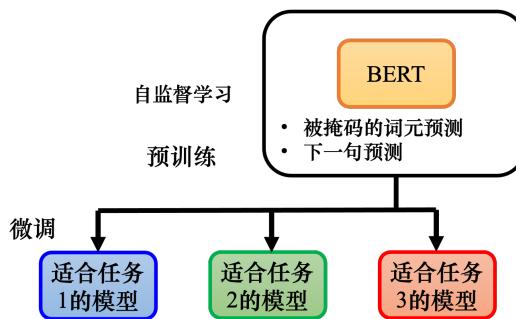


图 5.7 使用 BERT 解决下游任务

给 BERT 一些有标注的数据, 它可以学习各种任务, 将 BERT 分化并用于各种任务称为**微调 (fine-tune)**. 所以微调 BERT, 也就是对 BERT 进行微调, 让它可以做某种任务. 与微调相反, 在微调之前产生此 BERT 的过程称为**预训练 (pre-train)**. 所以产生 BERT 的过程就是自监督学习, 也可以将其称为预训练.

在谈如何对 BERT 进行微调之前, 先看看它的能力. 要测试自监督学习模型的能力, 通常会在多个任务上进行测试. BERT 就像一个胚胎干细胞, 它会分化为做各种任务的细胞, 通常不会只测试它在单个任务上的能力, 可以让 BERT 分化做各种任务来查看它在每个任务上的

正确率，再取个平均值。对模型进行测试的不同任务的这种集合，可以将其称为任务集。任务集中最著名的标杆（基准测试）称为**通用语言理解评估（General Language Understanding Evaluation, GLUE）**。GLUE 里面一共有 9 个任务：语言可接受性语料库（the Corpus of Linguistic Acceptability, CoLA），斯坦福情感树库（the Stanford Sentiment Treebank, SST-2），微软研究院释义语料库（the Microsoft Research Paraphrase Corpus, MRPC），语义文本相似性基准测试（the Semantic Textual Similarity Benchmark, STSB），Quora 问题对（the Quora Question Pairs, QQP），多类型自然语言推理数据库（the Multi-genre Natural Language Inference corpus, MNLI），问答自然语言推断（Qusetion-answering NLI, QNLI），识别文本蕴含数据集（the Recognizing Textual Entailment datasets, RTE），Winograd 自然语言推断（Winograd NLI, WNLI）。如果我们想知道像 BERT 这样的模型是否训练得很好，可以针对 9 个单独的任务对其进行微调。因此，实际上会为 9 个单独的任务获得 9 个模型。这 9 个任务的平均准确率代表该自监督模型的性能。自从有了 BERT，GLUE 分数（9 个任务的平均分）确实逐年增加。如图 5.8 所示，横轴表示不同的模型，除了 ELMo 和 GPT，还有各种 BERT。黑线表示人类在此任务上得到的正确率，可将其视为 1。图 5.8 的每个点代表一个任务。为什么要把它与人类的准确性进行比较？人类的正确率是 1。如果他们比人类好，这些点的值会大于 1。如果他们比人类差，这些点的值会小于 1。这些任务的评估指标不一定是正确率。用于每个任务的评估指标是不同的，不一定是正确率。如果直接比较这些点的值，没什么意思，所以要看模型跟人类之间的差距。在最初的时候，9 个任务中只有 1 个任务，机器比人类做得更好。随着越来越多的技术被提出，越来越多的其他任务可以比人类做得更好。对于那些远不如人类的任务，它们也在逐渐迎头赶上，机器的性能也在慢慢追赶。蓝色曲线表示机器的 GLUE 分数的平均值。最近的一些强模型，例如 XLNET，甚至超过了人类。当然，这只是这些数据集的结果。这并不意味着机器真的超越了人类，XLNET 在这些数据集中超越了人类。这意味着这些数据集还不够难。所以在 GLUE 之后，有人制作了 Super GLUE，让机器解决更难的自然语言处理任务。有了 BERT 这样的技术，机器在自然语言处理方面的能力确实又向前迈进了一步。BERT 究竟是如何使用的？接下来介绍下 4 个使用 BERT 的情况。

## 情况 1：情感分析

假设下游任务是输入一个序列并输出一个类别。这是一个分类问题，只是输入是一个序列。输入一个序列并输出一个类是一种什么样的任务？例如，情感分析，给机器一个句子，并告诉它

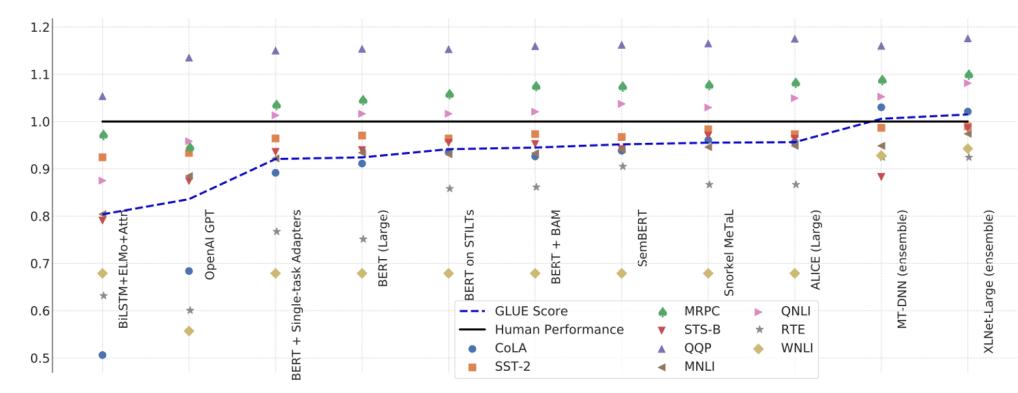


图 5.8 BERT 的训练过程

判断句子是正面的还是负面的. 对于 BERT, 它是如何解决情感分析的问题的? 如图 5.9 所示, 只要给它一个句子, 把 [CLS] 词元放在这个句子前面.[CLS]、 $w_1$ 、 $w_2$ 、 $w_3$  4 个输入对应于 4 个输出. 接着, 对 [CLS] 对应的向量应用线性变换, 将其乘上一个矩阵. 这里省略了 softmax, 通过 softmax 来确定输出类别是正面的或负面的等等. 但是, 必须要有下游任务的标注数据. 换句话说, BERT 没有办法从头开始解决情感分析问题, 其仍然需要一些标注数据, 需要提供很多句子以及它们的正面或负面标签来训练 BERT 模型. 在训练过程中, BERT 与这种线性变换放在一起, 称为完整的情感分析模型. 在训练时, 线性变换和 BERT 模型都利用梯度下降来更新参数. 线性变换的参数是随机初始化的, 而 BERT 初始的参数是从学习了做填空题的 BERT 来的. 在训练模型时, 会随机初始化参数, 接着利用梯度下降来更新这些参数, 最小化损失. 但是, 现在有了 BERT, 不必随机初始化所有参数, 随机初始化的参数只是线性变

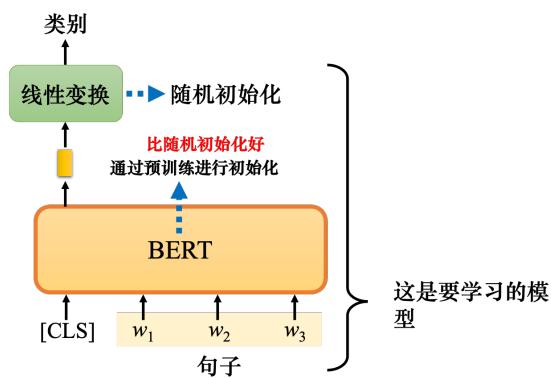


图 5.9 用 BERT 做情感分析

换的参数. BERT 的骨干 (backbone) 是一个巨大的 Transformer 编码器, 该网络的参数不是

随机初始化的。这里直接拿已经学会填空的 BERT 的参数当作初始化的参数，最直观和最简单的原因是它比随机初始化参数的网络表现更好。把学会填空的 BERT 放在这里时，它会获得比随机初始化的 BERT 更好的性能。如图 5.10 所示，横轴是训练的轮次（epoch），纵轴是训练损失。随着训练的进行，损失会越来越低。图 5.10 有各种各样的任务，任务的细节不需要关心。“微调”意味着模型有预训练。网络的 BERT 部分（网络的编码器），该部分的参数是由学会做填空的 BERT 的参数来做初始化的。从头开始训练（scratch）意味着整个模型，包括 BERT 和编码器部分都是随机初始化的。虚线是从头开始训练，如果是从头开始训练，在训练网络时，与使用会做填空的 BERT 进行初始化的模型相比，损失下降的速度相对较慢。随机初始化参数的网络损失仍然高于使用填空题来初始化 BERT 的网络。所以这就是 BERT 的好处。

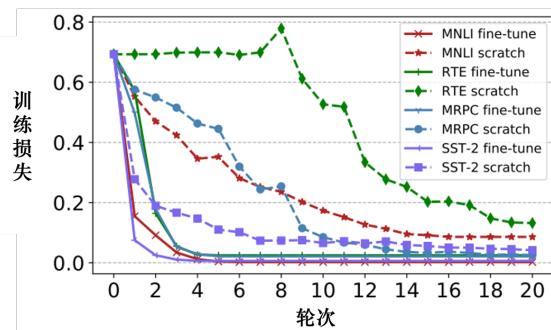


图 5.10 预训练模型的初始化结果对比

Q: BERT 的训练方法是半监督还是无监督？

A: 在学习填空时，BERT 是无监督的。但使用 BERT 执行下游任务时，下游任务需要有标注的数据。自监督学习会使用大量未标注的数据，但下游任务有少量有标注的数据，所以合起来是半监督。半监督是指我们有大量的未标注的数据和少量标注数据，这种情况称为半监督。所以使用 BERT 的整个过程就是使用预训练和微调，它可以被视为一种半监督的方法。

## 情况 2：词性标注

第二种情况是输入一个序列，然后输出另一个序列，但输入和输出的长度是一样的。什么样的任务要求输入输出长度相同？例如，**词性标注**（Part-Of-Speech tagging, POS tagging）。词性标注是指给定机器一个句子，其可以知道该句子中每个单词的词性。即使这个词是相同的，它也可能有不同的词性。BERT 是如何处理词性标注任务的？如图 5.11 所示，只需向 BERT 输入一个句子即可。之后，对于这句话中的每个词元，如果是中文，就是每一个字，每

个字都有一个对应的向量。然后把这些向量依次通过线性变换和 softmax 层。最后，网络预测给定单词所属的类别。例如，词性。如果任务不同，对应的类别也会不同。接下来和情况 1 完全一样。换句话说，要有一些带标签的数据。这仍然是一个典型的分类问题。唯一不同的是 BERT 部分，网络的编码器部分，其参数不是随机初始化的，它已经在预训练过程中找到了一组比较好的初始化的参数。

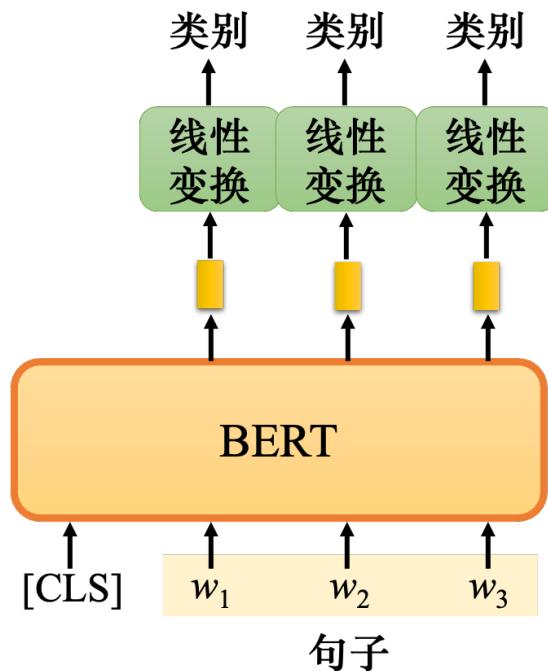


图 5.11 用 BERT 做词性标注

### 情况 3：自然语言推理

在情况 3 中，模型输入两个句子并输出一个类别。这里的例子都是自然语言处理的例子，但可以将这些例子更改为其他任务，例如语音任务或计算机视觉的任务。语音、文本和图像可以表示为一行向量，因此该技术不仅限于处理文本，还可以用于其他任务。情况 3 以两个句子作为输入，输出一个类别。什么样的任务需要这样的输入和输出？最常见的一种是**自然语言推理**（Natural Language Inference, NLI）。给机器两个输入语句：前提（premise）和假设（hypothesis）。机器所做的判断是否可以从前提中推断出假设，即前提与假设矛盾或者不矛盾？例如，如图 5.12 所示，前提是“一个骑马的人跳过一架坏掉的飞机（A person on a horse jumps over a broken down airplane）”，这是一个基准语料库中的例子。而假设是这个人在一家餐馆里（A person is at a diner），这是一个矛盾。机器要做的就是将两个句子作为输

入，输出这两个句子之间的关系。这种任务很常见，例如，立场分析。给定一篇文章，其下面有留言，要判断留言是赞成这篇文章的立场还是反对这篇文章的立场。只需将文章和留言一起放入模型中，模型要预测的是赞成还是反对。

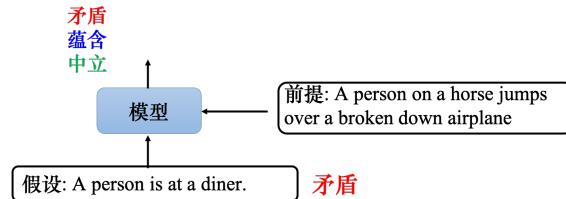


图 5.12 自然语言推理

BERT 如何解决这个问题？如图 5.13 所示，给定两个句子，这两个句子之间有一个特殊的分隔词元 [SEP]，并把 [CLS] 词元放在最前面的位置。这个序列是 BERT 的输入，然后 BERT 将输出另一个长度与输出长度相同的序列。但只将 [CLS] 词元作为线性变换的输入，然后决定输入这两个句子，输出应该是什么类别。对于 NLI，要输出这两个句子是否矛盾，仍然需要一些标注的数据来训练这个模型。BERT 的这部分不再是随机初始化的，它使用预训练的权重进行初始化。

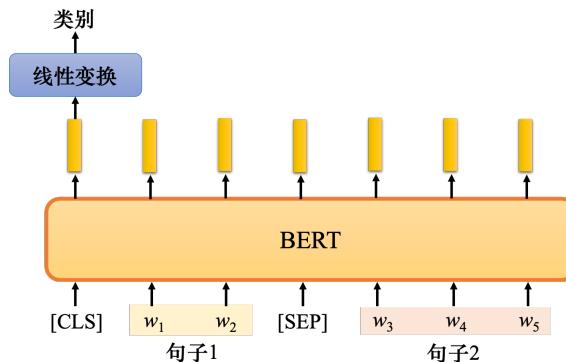


图 5.13 使用 BERT 进行自然语言推理

#### 情况 4：基于提取的问答

第四个情况是问答系统，给机器读一篇文章，问它一个问题，它就会回答一个答案。但这里的问题和答案有些限制，假设答案必须出现在文章里面，答案一定是文章中的一个片段，这是基于提取的问答（extraction-based question answering）。在此任务中，输入序列包含一篇文章和一个问题。文章和问题都是一个序列：

$$D = \{d_1, d_2, \dots, d_N\} \quad (5.1)$$

$$Q = \{q_1, q_2, \dots, q_M\}$$

对于中文，式 (5.1) 中每个  $d$  代表一个汉字，每个  $q$  代表一个汉字。

如图 5.14 所示，将  $D$  和  $Q$  放入问答模型中，希望它输出两个正整数  $s$  和  $e$ 。根据  $s$  和  $e$  可以直接从文章中截出一段就是答案，文章中第  $s$  个单词到第  $e$  个单词的片段就是正确答案。这是当今使用的一种非常标准的方法。



图 5.14 问答模型

例如，如图 5.15 所示，这里有一个问题和一篇文章，正确的答案是“重力（gravity）”。机器如何输出正确答案？问答模型应该输出  $s = 17$  并且  $e = 17$  来表示重力，因为重力是整篇文章的第 17 个字。所以  $s = 17, e = 17$  表示输出第 17 个单词作为答案。再举个例子，答案是“云中（within a cloud）”，其是文章的第 77 到 79 个字，模型要做的就是输出两个正整数 77 和 79，文章中第 77 个词到第 79 个词的文字就是模型的答案。

当然，我们不会从头开始训练问答模型，而会使用 BERT 预训练模型。如何用预先训练好的 BERT 解决这种问答问题呢？如图 5.16 所示，给 BERT 看一个问题、一篇文章。问题和文章之间有一个特殊标记 [SEP]。然后在开头放了一个 [CLS] 词元，这与自然语言推理的情况相同。在自然语言推理中，一个句子是前提，一个句子是结论，而在这里，一个是文章，一个是问题。在此任务中，唯一需要从头开始训练的只有两个向量（“从头开始训练”是指随机初始化），我们使用橙色向量和蓝色向量来表示它们，这两个向量的长度与 BERT 的输出是相同的。假设 BERT 的输出是 768 维向量，这两个向量也就是 768 维向量。如何使用这两个向量呢？如图 5.16a 所示，首先计算橙色向量和文档对应的输出向量的内积（inner product）。由于有 3 个词元代表文章，因此它将输出 3 个向量。计算这 3 个向量与橙色向量的内积可以得到 3 个值。然后将它们传递给 softmax 函数，将得到另外 3 个值。这种内积与注意力非常相似。如果把橙色部分可以视为查询，把黄色部分视为键，这就是一种注意力，应该尝试找到得分最高的位置。橙色向量和  $d_2$  的内积最大，则  $s$  应等于 2，输出的起始位置应为 2。

In meteorology, precipitation is any product of the condensation of 17 spherical water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain 77 atte 79 cations are called "showers".

What causes precipitation to fall?

**gravity**

$$s = 17, e = 17$$

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?

**graupel**

Where do water droplets collide with ice crystals to form precipitation?

**within a cloud**

$$s = 77, e = 79$$

图 5.15 基于提取的问答

如图 5.16b 所示，蓝色部分代表答案结束的地方。计算蓝色向量和文章对应的黄色向量的内积，接着对内积使用 softmax 函数。最后，找到最大值。如果第 3 个值最大， $e$  应为 3。正确答案是  $d_2$  和  $d_3$ ，所以模型要做的实际上是预测正确答案的起始位置，因为答案一定在文章里。如果文章中没有答案，就不能使用这个技巧。这里假设答案一定在文章中，必须在文章中找到答案的起始位置和结束位置。这是问答模型需要做的。当然，我们需要一些训练数据才能训练这个模型。请注意，蓝色和橙色向量是随机初始化的，而 BERT 是由其预训练的权重初始化的。

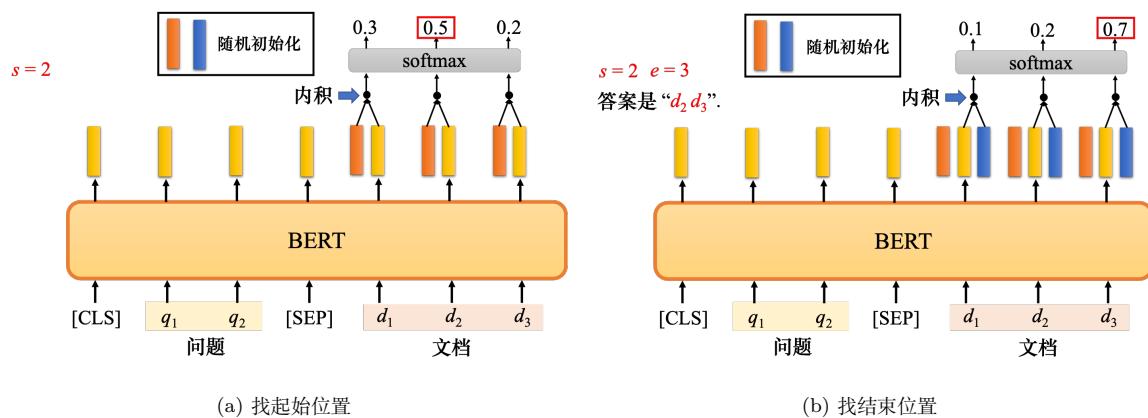


图 5.16 使用 BERT 来进行问答

Q: BERT 的输入长度有限制吗?

A: 理论上没有。实际上有，理论上，因为 BERT 模型是一个 Transformer 编码器，所以它可以输入很长的序列。只要我们有能力做自注意力。但是自注意力的运算量很高，所以在实践中，BERT 无法真正输入太长的序列，最多可以输入 512 长度的序列。如果输入一个 512 长度的序列，中间的自注意力即将生成 512 乘以 512 大小的注意力度量 (metric)，计算量会非常大，所以实际上 BERT 的长度不是无限长的。

因为用一篇文章训练需要很长时间，所以文章会被分成几个段落，每一次只取其中一个进行训练，不会将整篇文章输入到 BERT 中。因为如果想要的距离太长，就会在训练中遇到问题。填空题和问答两件事之间有什么关系？BERT 所能做的事情不仅仅是填空，但我们无法自己训练它。首先是最早的谷歌 BERT，它训练使用的数据量已经很大了，它使用的数据包含了 30 亿个词汇。哈利波特全集大约有 100 万词汇，其是哈利波特全集的 3000 倍。最早的 BERT 使用的数据量是哈利波特全集的 3000 倍。如图 5.17 所示，纵轴代表 GLUE 分数，横轴代表预训练步数。GLUE 有 9 个任务，9 个任务的平均分数是 GLUE 的分数。绿线是谷歌原始的 BERT 的 GLUE 分数。橙线是谷歌的 ALBERT 的 GLUE 分数，ALBERT 是 BERT 的进阶版本，其参数量相比 BERT 大大减少。蓝线是李宏毅团队训练的 ALBERT，但是李宏毅团队训练的并不是最大的版本。原始的 BERT 有基础版本 (BERT-base) 和大版本 (BERT-large)。BERT-large 很难训练，所以用最小的版本 (ALBERT-base) 来训练，看看它是否与谷歌的结果相同。

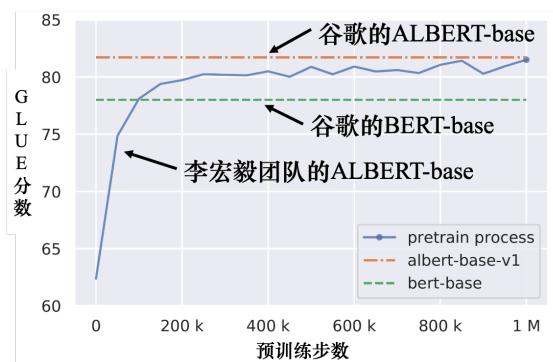


图 5.17 使用 ALBERT 训练 GLUE

30 亿数据看起来很多，但它是未标注的数据。因此，从网络上随便爬取一堆文字就可以有这么多数据，爬取这个等级的数据并不是很难，但训练的部分很困难。总共的预训练步数为一

百万次，也就是参数需要更新一百万次。如果使用 TPU，则需要运行 8 天；如果使用一般的 GPU，这个至少需要运行 200 天。所以训练这种 BERT 模型真的很难，可以在一般的 GPU 上对其进行微调，在一般的 GPU 上微调，BERT 只需要大约半小时到一个小时。但如果从头开始训练它。也就是训练它做填空题，这将花费太多时间，而且无法在一般的 GPU 上完成。为什么要自己训练一个 BERT？谷歌已经训练了 BERT，这些预训练模型也是公开的。如果训练 BERT 的结果和谷歌的 BERT 差不多，这没什么意义。BERT 的训练过程中需要耗费非常大的计算资源，所以是否有可能节省这些计算资源，有没有可能让它训练得更快。要知道如何让它训练得更快，或许可以先观察它的训练过程。过去没有人观察过 BERT 的训练过程，因为在谷歌的论文中只提到了 BERT 在各种任务中都做得很好，但 BERT 在学习填空的过程中学到了什么？观察在这个过程中，BERT 学会填动词、学会填名词和学会填代词的时候。所以训练 BERT 之后，可以观察 BERT 学会填充各种词汇的时候以及它是提高填空能力的方式。得到的结论与想象得不太一样，大家可以参考论文“Pretrained Language Model Embryology: The Birth of ALBERT”。

上述任务均不包括序列到序列（Sequence-to-Sequence，Seq2Seq）模型。如果想解决 Seq2Seq 问题怎么办？BERT 只有预训练编码器，有没有办法预训练 Seq2Seq 模型的解码器？如图 5.18 所示，图中有一个编码器和一个解码器。输入是一串句子，输出是一串句子。将它们与中间的交叉注意力（cross attention）连接起来，然后对编码器的输入做一些扰动来损坏它。解码器是想要输出的句子，跟损坏它之前是完全相同的。编码器看到损坏的结果，然后解码器要输出还原句子被损坏之前的结果。训练这个模型实际上是预训练一个 Seq2Seq 模型。

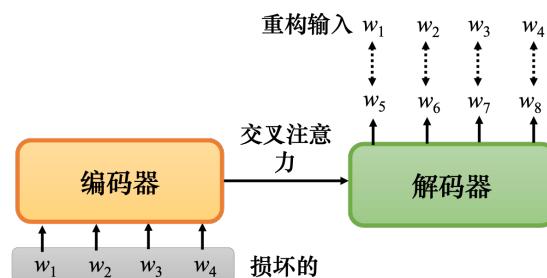


图 5.18 预训练一个 Seq2Seq 模型

损坏句子的方式有多种，如图 5.19 所示，有一篇是题为：“MASS: Masked Sequence to Sequence Pre-training for Language Generation”的论文，它说损坏的方法就像 BERT 那样，只要掩盖一些地方，它就结束了。但其实有多种方法可以损坏句子。例如，删除一些单

词. 打乱词汇顺序 (语序) . 把单词的顺序做个旋转. 或既插入 MASK, 又删除某些单词. 总之, 有各种方法把输入句子损坏, 再通过 Seq2Seq 模型把它还原. 有一篇论文题为 “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension” . 这篇论文中把这些方法都使用上去, 它的结果比 MASS 更好.

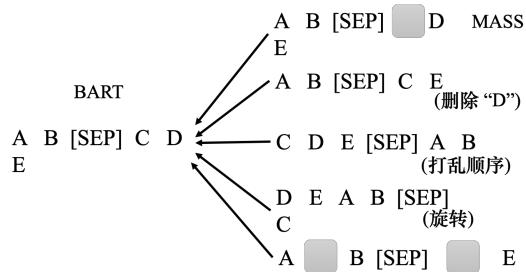


图 5.19 损坏句子的方法

损坏的方法有很多, 掩码的方法也有很多, 到底哪种方法更好呢? 谷歌在题为 “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer” 的论文中做了相关的实验, 并提出了预训练模型——转移文本到文本的 Transformer (Transfer Text-to-Text Transformer, T5) . 这篇论文做了各种尝试, 完成了可以想象的所有组合. T5 是在巨大干净的爬取语料库 (Colossal Clean Crawled Corpus, C4) 上进行训练的. C4 是一个公开数据集, 可以下载它, 但其原始文件大小为 7 TB, 下载完它, 也不一定有足够的存储空间来保存它. 下载完成后, 可以通过谷歌提供的脚本进行预处理. 语料库网站上的文档说使用一个 GPU 进行预处理需要 355 天. 我们可以下载它, 但是在预处理时也是有问题的. 所以, 做深度学习使用的数据量和模型非常惊人.

### 5.2.2 BERT 有用的原因

为什么 BERT 有用? 最常见的解释是, 当输入一串文字时, 每个文字都有一个对应的向量, 这个向量称为嵌入. 如图 5.20 所示, 这个向量很特别, 因为这个向量代表了输入字的意思. 例如, 模型输入 “深度学习”, 输出 4 个向量. 这 4 个向量代表 “深”、“度”、“学”和 “习”的意思. 把这些字对应的向量一起画出来并计算它们之间的距离, 意思越相似的字, 它们的向量就越接近. 如图 5.21 所示, 例如, “果”和 “草”都是植物, 它们的向量就比较接近. “鸟”和 “鱼”是动物, 所以它们可能更接近. “电”既不是动物也不是植物, 所以比较远. 中文会有歧义 (一字多义), 很多语言也都有歧义. BERT 可以考虑上下文, 所以同一个字, 例如

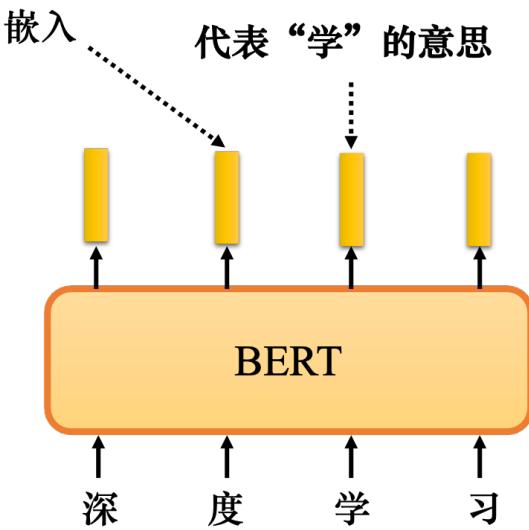


图 5.20 BERT 输出的嵌入代表了输入的字的意思

“果”这个字，它的上下文不同，它的向量是不会一样的。所以吃苹果的果和苹果手机的果都是“果”，但根据上下文，它们的意思不同，所以他们对应的向量就会不一样。吃苹果的“果”可能更接近“草”，苹果手机的“果”可能更接近“电”。

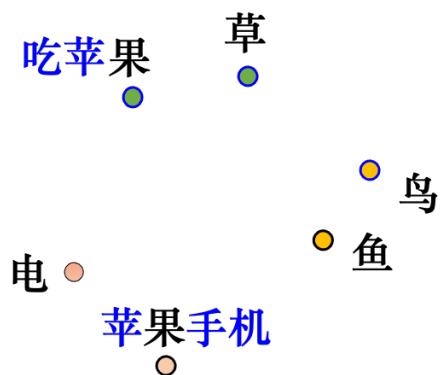


图 5.21 意思相近的字，嵌入更接近

如图 5.22 所示，假设现在考虑“果”这个字，收集很多提到“果”字的句子，比如“喝苹果汁”、“苹果电脑”等等。把这些句子都放入 BERT 里面，接下来，再去计算每个“果”对应的嵌入。输入“喝苹果汁”得到“果”的向量。输入“苹果电脑”，也得到“果”的向量。这两个向量不会相同。因为编码器中有自注意力，所以根据“果”字的不同上下文，得到的向量会不同。接下来，计算这些向量之间的余弦相似度，即计算它们的相似度，结果是这样的。

如图 5.23 所示，这里有 10 个句子，前 5 句中的“果”代表可以吃的苹果。例如，第一句

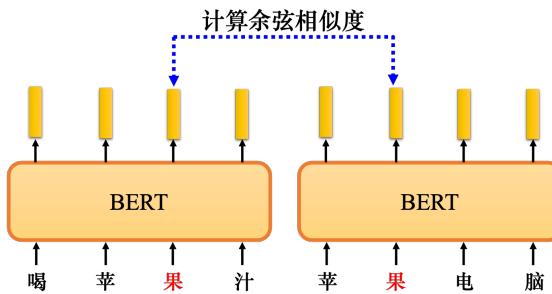


图 5.22 计算余弦相似度

话是“今天买了苹果吃” . 这五个句子都有“果”这个字. 接下来的五个句子也有“果”这个字, 但都是指苹果公司的“果” . 例如, “苹果即将在下个月发布一款新 iPhone.” 这边有 10 个“果”, 两两之间去计算相似度, 得到一个  $10 \times 10$  的矩阵. 图 5.23 中的每一格代表两个“果”的嵌入之间的相似度. 相似度的值越大, 颜色越浅. 前五句中的“果”接近黄色, 自己跟自己算相似度, 一定是最大的. 自己跟别人的相似度一定要小一些. 前五个“果”算相似度较高, 后五个“果”算相似度也较高. 但是前五个“果”和后五个“果”的相似度较低. BERT 知道前五个“果”指的是可以吃的苹果, 所以它们比较像. 后五个“果”指的是苹果公司的“果”, 所以它们比较像. 但这两堆“果”的意思是不一样的. 所以 BERT 的每个输出向量代表输入字的意思, BERT 在填空的过程中学会了每个字的意思. 也许它真的理解中文, 对它而言, 中文的符号不再是没有关系的. 因为它了解中文的意思, 所以它可以在接下来的任务中做得更好.

为什么 BERT 可以输出代表输入字意思的向量? 1960 年代的语言学家 John Rupert Firth 提出了一个假设, 他说要知道一个词的意思, 就要看这个词的“公司 (company)”, 也就是经常和它一起出现的词汇, 也就是它的上下文. 一个词的意思取决于它的上下文. 以苹果中的“果”为例, 如果它经常与吃、树等一起出现, 它可能指的是可以吃的苹果; 如果经常与电、专利、股价等一起出现, 可能指的是苹果公司. 因此, 可以从上下文中推断出单词的意思.

如图 5.24 所示, 而 BERT 在学习填空的过程中所做的, 也许就是学习从上下文中提取信息. 训练 BERT 时, 给它  $w_1$ 、 $w_2$ 、 $w_3$  和  $w_4$ , 掩码  $w_2$ , 并告诉它预测  $w_2$ . 它如何预测  $w_2$ ? 它会从上下文中提取信息来预测  $w_2$ . 所以这个向量就是它的上下文信息的精华, 可以用来预测  $w_2$  是什么.

如图 5.25 所示, 这样的想法在 BERT 之前就已经存在了. 有一种技术是词嵌入 (word embedding), 词嵌入中有一种技术称为连续词袋 (Continuous Bag Of Words, CBOW) 模型. 连续词袋模型所做的与 BERT 完全相同, 把中间挖空, 预测空白处的内容. 连续词袋模

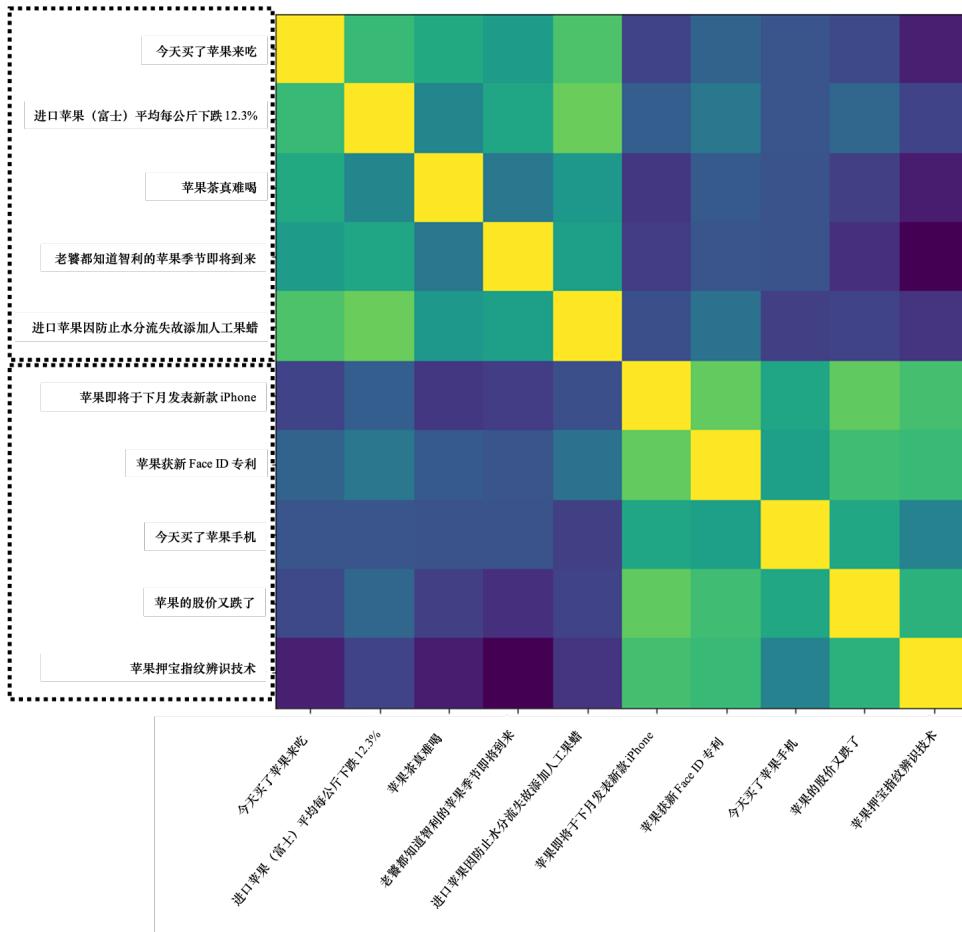


图 5.23 余弦相似度的计算结果

型可以给每个词汇一个向量，代表词汇的意思。连续词袋模型是一个非常简单的模型，它使用了两个变换。为什么它只用两个变换？能不能再复杂点？为什么连续词袋模型只用线性，不用深度学习？连续词袋模型的作者 Thomas Mikolov 的解释是可以用深度学习，他之所以选择线性模型是因为当时的计算能力 (computing power) 和现在的数量级不一样，当时还很难训练一个非常大的模型，所以他选择了一个比较简单的模型。而 BERT 相当于一个深度版本的连续词袋模型。

BERT 还可以根据不同的上下文从相同的词汇中产生不同的嵌入，因为它是词嵌入的高级版本，考虑了上下文。BERT 抽取的这些向量或嵌入也称为**语境化的词嵌入 (contextualized word embedding)**。训练在文字上的 BERT 也可以用来对蛋白质、DNA 和音乐进行分类。以 DNA 链的分类问题为例。如图 5.26 所示，DNA 由脱氧核苷酸组成，脱氧核苷酸由碱基、脱氧核糖和磷酸构成。其中碱基有 4 种：腺嘌呤 (A)、鸟嘌呤 (G)、胸腺嘧啶 (T) 和胞嘧啶 (C)。给定一条 DNA，尝试确定该 DNA 属于哪个类别 (EI、IE 和 N 是 DNA 的类别)。总

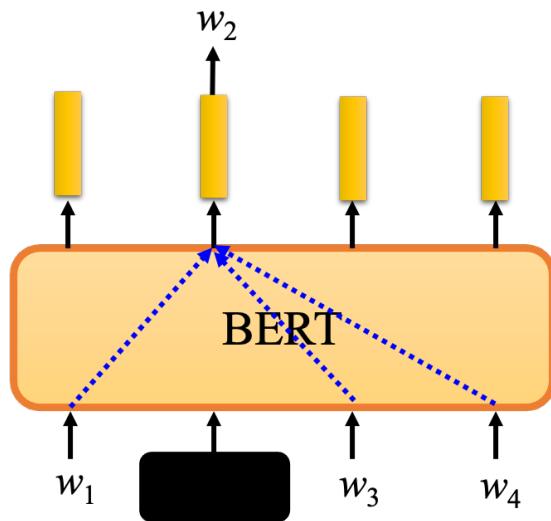


图 5.24 通过上下文信息预测掩码部分

之，这是一个分类问题，只需用训练数据和标注数据来训练 BERT 就可以了。

如图 5.27 所示，DNA 可以用 ATCG 表示，每个字母可以对应到一个英语单词，例如，“A”是“we”，“T”是“you”，“C”是“he”，“G”是“she”。对应的单词并不重要，可以随机生成它们。“A”可以对应任何单词，“T”、“C”和“G”也可以，这并不重要，对结果影响不大。一串 DNA 可以变成一串单词，只是这串文字看不懂而已。例如，“AGAC”变成“we she we he”。然后，将这串文字放入 BERT 中，一样有 [CLS]，产生一个向量，然后通过线性变换，一样进行分类，只是分类是 DNA 的类别。和以前一样，线性变换使用随机初始化，BERT 由预训练模型初始化。但是用于初始化的模型是在英文上学会做填空题的 BERT。

如果将 DNA 序列预处理成一个无意义的序列，那么 BERT 的目的是什么？BERT 可以分析有效句子的语义，怎么能给它一个难以理解的句子？做这个实验有什么意义？

如图 5.28 所示，这里有不同的任务。蛋白质的分类做了 3 种任务。蛋白质是由氨基酸组成的，有十几种氨基酸，给每种氨基酸随便一个词汇。DNA 就是一组 ATCG，音乐也是一组音符，可以给每个音符随便一个词汇，将其作为文章分类问题来做。如果没有使用 BERT，得到的结果是蓝色框中的结果。如果使用 BERT，得到的结果是红色框中的结果。从这两个结果可知，使用 BERT 实际上更好。

BERT 可以学到语义，从嵌入中可以清楚地观察到 BERT 确实知道每个单词的意思，它知道哪些词汇意思比较像，哪些单词意思比较不像。即使给它一个乱七八糟的句子，它仍然可以很好地对句子进行分类。所以也许它的能力并不完全来自他看得懂文章这件事，可能还有

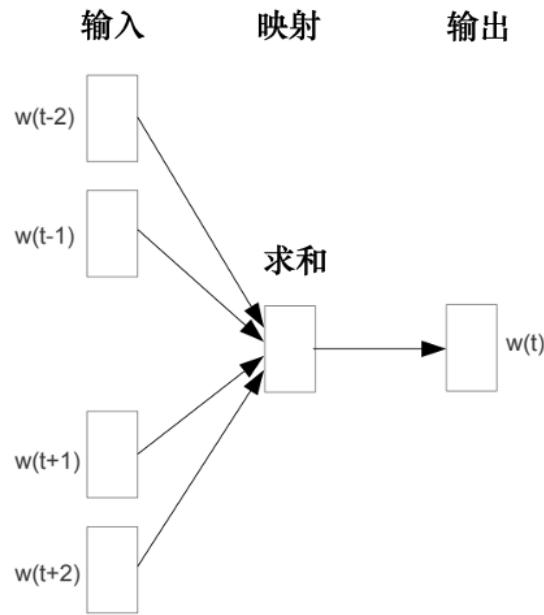


图 5.25 连续词袋模型

EI	CCAGCTGCATCACAGGAGGCCAGCGAGCAGGTCTGTTCAAGGGCCTCGAGCAGTC
EI	AGACCCGGCGGGAGGCGGAGGACCTGCAGGGTGAGCCCCACCGGCCCTCCGTGCCCCCGC
IE	AACTGCCCCTCTTGTGCCCTCCCCACAGTGGCCCTTCCAGGACAAACTGGAGAAGT
IE	CCACTCAGCCAGGCCCTCTCTCCAGGCCCCACGGCCCTCAGGATGAAGCTG
IE	CCTGATCTGGTCTCCCTCCACCCCTCAGGGAGCCAGGCTGGCATTTCTGGCAGCAAG
IE	AGCCCTAACCCCTCTGTCACCCCTCAGCTAAAGCTCCCTGACAACATGGGACAGCGT
IE	CCACTCAGCCAGGCCCTCTCTCCAGGCCCCACGGCCCTCAGGATGAAGCTG
N	CTGTGTTACACATCAAGGGGGGACATCGTGTCAAGTGAGCTGGGGAGGGCG
N	GTGTTACCGAGGGCATTTCTAACAGTCTTACTACGGCCTCGCCGACCCGCGCTCG
N	TCTGAGCTGATTGCTATTCTCAGCTGACCCCTGGTTCTCTTAGCTACCTGC

类别                  DNA 序列

图 5.26 DAN 分类问题

其他原因。例如，BERT 可能本质上只是一组比较好的初始化参数，它不一定与语义有关，也许这组初始参数比较适合训练大型模型，这个问题需要进一步的研究来回答。目前使用的模型往往是非常新的，它们为什么能成功运作，还有很大的研究空间。

### 5.2.3 BERT 的变种

BERT 还有很多其他的变种，比如多语言 BERT (multi-lingual BERT)。如图 5.29 所示，多语言 BERT 使用中文、英文、德文、法文等多种语言进行训练 BERT 做填空题。谷歌发布的多语言 BERT 使用 104 种不同的语言进行训练，所以它可以做 104 种语言的填空题。

多语言 BERT 有一个非常神奇的功能，如果用英文问答数据训练它，它会自动学习如何做中文问答。图 5.30 所示是一个真实实验的例子。这个例子使用了两种数据集进行微调：英文

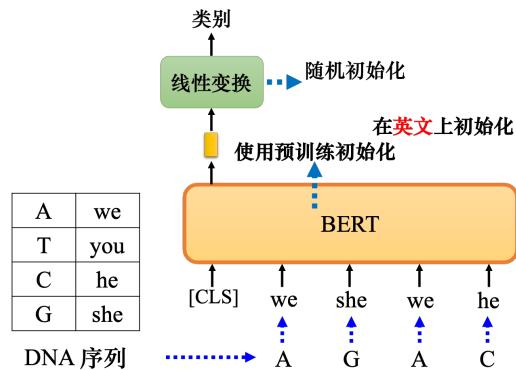


图 5.27 使用 BERT 进行 DNA 分类

	Protein			DNA				Music
	localization	stability	fluorescence	H3	H4	H3K9ac	Splice	composer
specific	69.0	76.0	63.0	87.3	87.3	79.1	94.1	-
BERT	64.8	74.5	63.7	83.0	86.2	78.3	97.5	55.2
re-emb	63.3	75.4	37.3	78.5	83.7	76.3	95.6	55.2
rand	58.6	65.8	27.5	75.6	66.5	72.8	95	36

图 5.28 使用 BERT 处理不同任务

问答数据集——SQuAD 和台达电的中文数据集——DRCD. 实验中所采用的是 F1 分数 (F1 score)，其也称为综合分类率. 在 BERT 提出之前，结果并不好. 在 BERT 之前，最强的模型是 QANet，QANet 的 F1 分数为 78.1%. 如果允许使用中文进行预训练做填空题，然后使用中文问答数据进行微调，BERT 在中文问答测试集上的 F1 分数将达到 89.1%. 事实上，人类在同一个数据集上只能做 93%，所以其表现跟人差不多. 神奇的是，如果使用一个多语言的 BERT，用英文问答数据对其进行微调，它仍然可以回答中文问答问题，并且有 78% 的 F1 分数，这和 QANet 的 F1 分数差不多！从未受过中英互译训练，也从未阅读过中文问答数据集. 它在没有任何准备的情况下参加了这次中文问答测试.

有的人可能会说：“多语言 BERT 在预训练的时候看了 104 种语言，其中包括中文” . 但是在预训练期间，多语言 BERT 的学习目标是做填空题，它只学会了中文填空，接下来教它

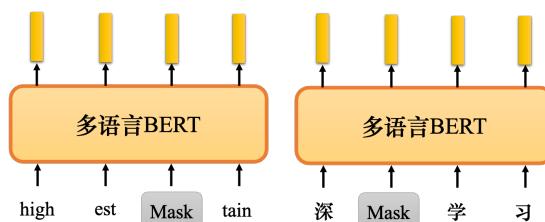


图 5.29 多语言 BERT

做英文问答，它居然自动学会了中文问答。一个简单的解释是：对于多语言的 BERT，不同的语言的差异不大。

模型	预训练	微调	测试	EM	F1
BERT 104种语言	无	中文	中文	66.1	78.1
	中文	中文		82.0	89.1
	104种语言	中文	中文	81.2	88.7
		英文		63.3	78.8
		中文+ 英文		82.6	90.1

图 5.30 使用多语言 BERT 进行问答

如图 5.31 所示，不管使用中文还是英文，对于意思相同的词，它们的嵌入都会很近。所以兔子和 rabbit 的嵌入很近，跳和 jump 的嵌入很近，鱼和 fish 的嵌入很近，游和 swim 的嵌入很近。许多语言 BERT 在看过大量语言的过程中自动学会了这件事情。

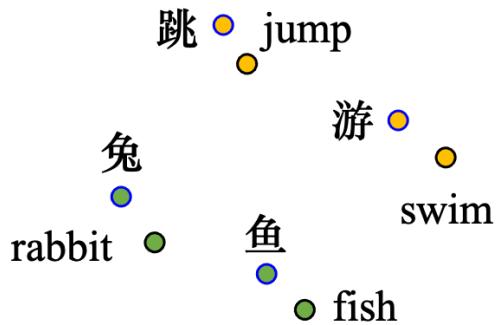


图 5.31 多语言 BERT 对比

如图 5.32 所示，我们可以做一些验证。验证的标准称为平均倒数排名（Mean Reciprocal Ranking, MRR）。MRR 的值越高，不同语言的嵌入对齐就越好。更好的对齐意味着具有相同含义但来自不同语言的单词，它们的向量是接近的。MRR 越高，则同样意思不同语言的单词的向量就越接近。图 5.32 的纵轴是 MRR，越高越好。最右边的深蓝线是谷歌发布的 104 种语言的多语言 BERT 的 MRR，它的值非常高。这代表对该多语言 BERT 来说，不同语言没有太大区别。多语言 BERT 只是看意思，不同语言对它来说没有太大区别。李宏毅团队最先使用的数据较少，每种语言只使用了 20 万个句子，训练的模型的结果并不好。之后，李宏毅团队给每种语言 1000 K 数据。有了更多的数据，多语言 BERT 可以学习对齐。所以数据量是不同语言能否成功对齐的一个非常关键的因素。所以有时神奇的是，很多现象只有在数据量足够时才会显现出来。过去，没有模型具有多语言能力，可以在 A 语言中进行问答训练，然

后直接转移到 B 语言. 一个可能的原因是过去没有足够的数据，现在有足够的数据，有很多计算资源. 因此，现在可以观察到这种现象.

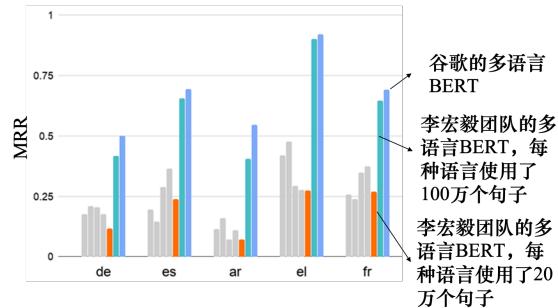


图 5.32 多语言 BERT 对比

BERT 可以将不同语言中具有相同含义的符号放在一起，并使它们的向量很接近. 但是在训练多语种 BERT 的时候，如果给它英文，就可以用英文填空. 如果给它中文，它可以用中文填空，它不会混合在一起. 如果对它来说，不同语言之间没有区别，怎么可能只用英语标记来填充英语句子呢？给它一个英文句子，为什么它不会用中文填空？但是它没有这样做，这意味着它知道语言的信息. 那些来自不同语言的符号毕竟还是不同的. 它不会完全抹掉语言信息，语言信息可以被找到. 语言信息并没有隐藏很深，把所有的英文单词丢到多语言 BERT 中，把它们的嵌入平均起来. 如图 5.33 所示，把所有中文的嵌入平均起来，两者相减就是中文和英文之间的差距. 给多语言 BERT 一个英文句子并得到它的嵌入，把这些嵌入加上蓝色的向量，这就是英文和中文的差距. 对多语言 BERT 来说，这些向量就变成了中文的句子. 要求它填空时，它实际上可以用中文填答案.

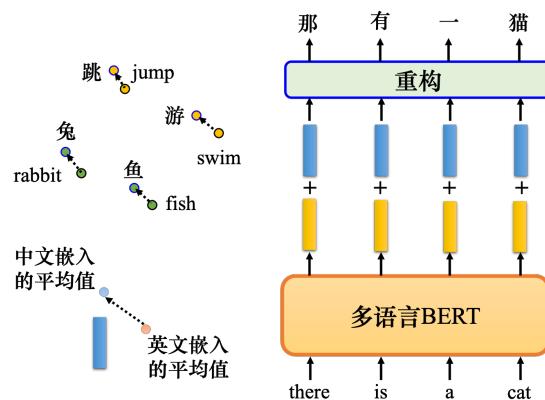


图 5.33 中英文之间的差距

多语种 BERT 可以做一个很棒的无监督翻译，如图 5.34 所示，把 “The girl that can help

me is all the way across town. There is no one who can help me.”这句话扔进多语种 BERT. 再把蓝色的向量加到 BERT 的嵌入上, 本来 BERT 读到的是英文句子的嵌入, 加上蓝色向量, BERT 会觉得它读到的是中文的句子. 然后, 教他做填空题, 把嵌入变成句子以后, 它得到的结果如图 5.34 的表表示, 可以某种程度上做到无监督词元级翻译 (unsupervised token-level translation). 这不是很好的翻译, 多语言 BERT 表面上看起来把不同语言、同样意思的单词拉得很近, 但是语言的信息还是藏在多语言 BRRT 里面.

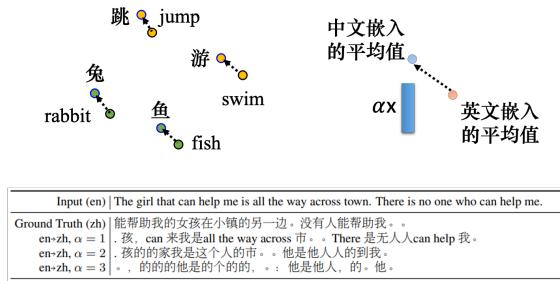


图 5.34 无监督词元级翻译

### 5.3 生成式预训练 (GPT)

在自监督学习中, 除了 BERT 系列的模型, 还有一个非常有名的模型——GPT 系列的模型. BERT 做的是填空题, 而 GPT 就是改一下在自监督学习的时候要模型做的任务. GPT 要做的任务是预测接下来会出现的词元. 如图 5.35 所示, 例如, 假设训练数据里面, 有一个句子是“深度学习”. 给 GPT 输入词元  $\langle \text{BOS} \rangle$  (beginning of sentence), GPT 会输出一个嵌入 (embedding). 接下来用这个嵌入去预测下一个应该出现的词元. 在这个句子里面, 根据这笔训练数据, 下一个应该出现的词元是“深”. 训练模型时, 根据第一个词元, 根据  $\langle \text{BOS} \rangle$  的嵌入, 它要输出词元“深”. 接下来讲下这个部分的具体操作, 对一个嵌入  $h$  进行一个线性变换, 再进行一个 softmax 操作可以得到一个分布. 跟一般做分类的问题是一样的, 输出的分布跟正确答案的交叉熵 (cross entropy) 越小越好. 也就是要去预测下一个出现的词元. 接下来要做的事情就是以此类推了, 给 GPT 输入  $\langle \text{BOS} \rangle$  跟“深”, 它产生嵌入. 接下来它会预测下一个出现的词元, 告诉它说下一个应该出现的词元是“度”. 再反复继续下去, 给它  $\langle \text{BOS} \rangle$ 、“深”和“度”, 然后预测下一个应该出现的词元, 它应该要预测“学”. 给 GPT 输入  $\langle \text{BOS} \rangle$ 、“深”“度”和“学”, 接下来下一个应该出现的词元是“习”, 因此它应该要预测出下一个应该出现的词元是“习”.

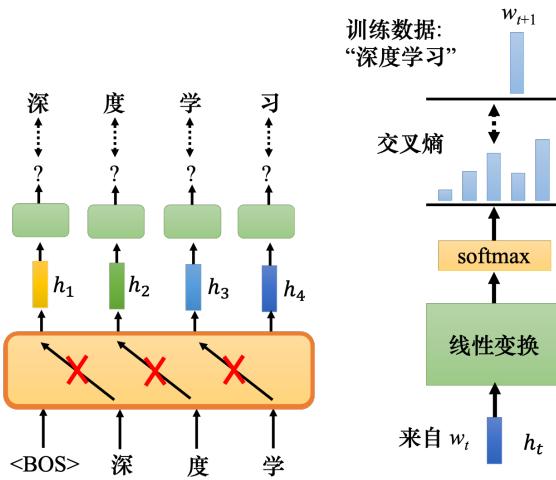


图 5.35 使用 GPT 预测下一个词

实际上不会只用一笔句子训练 GPT，而是用成千上万个句子来训练模型。GPT 厉害的地方就是用了很多资料训练了一个异常巨大的模型。GPT 模型建立在 Transformer 的解码器的基础上，不过其会做 mask 的注意力，给定  $\langle \text{BOS} \rangle$  预测“深”的时候，不会看到接下来出现的词汇。给 GPT “深”要预测“度”的时候，其不会看到接下来要输入的词汇，以此类推。因为 GPT 可以预测下一个词元，所以它有生成的能力，可以让它不断地预测下一个词元产生完整的文章，大家提到 GPT 的时候，往往会想到一只独角兽。因为 GPT 系列最知名的一个例子，就是用 GPT 写了一篇跟独角兽有关的新闻，假新闻里面说在安第斯山脉发现了独角兽，所以大家提到 GPT 的时候都会想到这个假新闻。

GPT 系列可以把一句话补完，如何把一句话补完用在下游的任务上呢？例如，怎么把 GPT 用在问答或者是其他的跟自然语言处理有关的任务上呢？GPT 可以跟 BERT 用一样的做法，BERT 是把 Transformer 编码器后面接一个简单的线性的分类器，也可以把 GPT 拿出来接一个简单的分类器，这也是会有效的，但是在 GPT 的论文没有这样做。GPT 没有这样做有几个原因，首先，BERT 已经用过这一招了，总不能再用一样的东西，这样写论文就没有人觉得厉害了。其次 GPT 模型太大了，大到连微调可能都有困难。在用 BERT 的时候，要把 BERT 模型后面接一个线性分类器，然后 BERT 也是要训练的模型的一部分，所以它的参数也是要调的，只需要微调它就好了，但是微调还是要花时间的。也许 GPT 实在是太过巨大，巨大到要微调它，要训练一个轮次可能都有困难。所以 GPT 系列有一个更狂的使用方式，这个更狂的使用方式和人类更接近，如图 5.36 所示，假设考生在进行托福的听力测验，首先有一个题目的说明，让考生从 A/B/C/D 四个选项里面选出正确的答案。给一个范例，给

出题目和正确答案. 看到新的问题, 期待考生就可以举一反三开始作答. 我们希望 GPT 系列的模型也能够举一反三, 可以进行**小样本学习 (few-shot learning)**. 小样本学习, 即在小样本上的快速学习能力. 每个类只有  $k$  个标注样本,  $k$  非常小. 如果  $k = 1$ , 称为**单样本学习 (one-shot learning)**; 如果  $k = 0$ , 称为**零样本学习 (zero-shot learning)**.

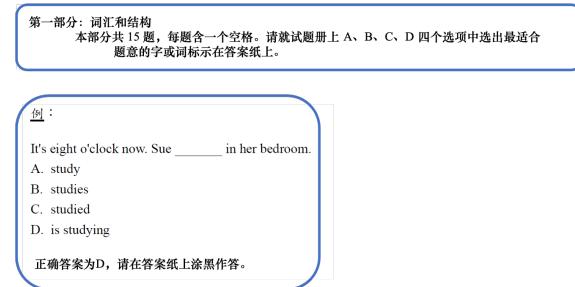


图 5.36 托福听力测验

假设要 GPT 做翻译, 如图 5.37a 所示, 先输入“把英语翻译成法语 (Translate English to French)”, 这个句子代表问题的描述. 然后给它几个范例, 接下来输入 cheese, 让它把后面的补完, 希望它就可以产生翻译的结果. 在训练的时候 GPT 并没有教它做翻译这件事, 它唯一学到的就是给一段文字的前半段把后半段补完. 现在直接给它前半段的文字就长这个样子, 让它翻译. 给几个例子, 告诉模型说翻译是怎么回事, 接下来输入单词 cheese, 后面能不能就直接得到法文的翻译结果. GPT 中的小样本学习不是一般的学习, 这里面完全没有梯度下降, 训练的时候就是要跑梯度下降, 而 GPT 中完全没有梯度下降, 完全没有要去调 GPT 模型参数的意思. 这种训练称为**语境学习 (in-context learning)**, 代表它不是一种一般的学习, 它连梯度下降都没有做.

我们也可以给 GPT 更大的挑战, 在考托福听力测验的时候, 都只给一个例子. 如图 5.37b 所示, 也给 GPT 一个例子, 就知道它要做翻译这件事, 也就是单样本学习. 还有更狂的是零样本学习, 如图 5.37c, 直接给它一个叙述说现在要做翻译, GPT 能不能够自己就看得懂就自动知道要做翻译. GPT 如果能够做到, 就非常地惊人了. GPT 系列到底有没有达成这个目标, 这是一个见仁见智的问题, 它不是完全不可能答对, 但是正确率有点低, 相较于微调模型, 正确率是有点低的.

如图 5.38 所示, 纵轴是正确率. 第 3 代的 GPT (GPT-3) 测试了 42 个任务, 3 条实线分别代表小样本、单样本跟零样本在 42 个任务中的平均正确率. 横轴代表模型的大小, 实验中测试了一系列不同大小的模型, 从 1 亿的参数到 1750 亿的参数. 小样本的部分从 20 几%

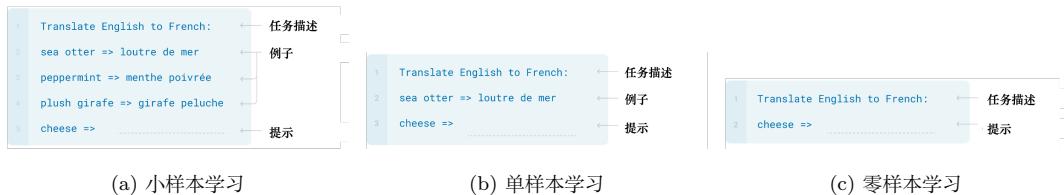


图 5.37 语境学习

的平均正确率一直做到 50 几% 的平均正确率. 至于 50 几% 的平均正确率算是有做起来还是没有做起来, 这是个见仁见智的问题. 有些任务 GPT-3 还真的学会了, 例如加减法, GPT-3 可以得到两个数字相加的正确结果. 但是有些任务, GPT-3 可能怎么学都学不会. 例如一些跟逻辑推理有关的任务, 它的结果就不如人意.

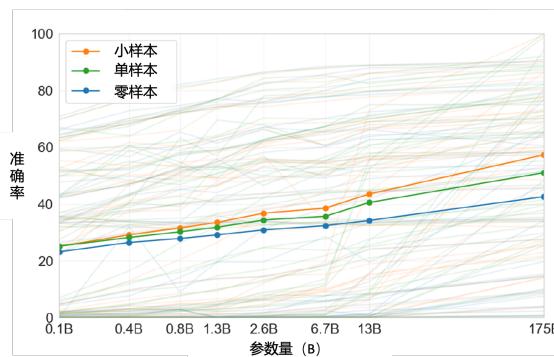


图 5.38 使用 GPT-3 进行语境学习

如图 5.39 所示, 自监督学习不仅可以用在文字上, 还可以用在语音和计算机视觉 (Computer Vision, CV) 上. 自监督学习的技术很多, BERT 跟 GPT 系列只是自监督学习的方法的其中一种, 它们是属于预测那一类. 计算机视觉中比较典型的模型是 SimCLR 和 BYOL. 在语音也可以使用自监督学习的概念, 可以试着训练语音版的 BERT. 怎么训练语音版的 BERT 呢? 我们就看看文字版的 BERT 是怎么训练的, 例如, 做填空题, 语音也可以做填空题, 就把一段声音信号盖起来, 叫机器去猜盖起来的部分是什么, 语音也可以预测接下来会出现的内容. GPT 就是预测接下来要出现的词元, 语音也可以让模型预测接下来会出现的声音. 所以我们也可以做语音版的 GPT, 语音版的 BERT 都已经有很多相关的研究成果了.

在自然语言处理的领域, 有 GLUE 语料库, 这个基准的资料库里面有 9 个自然语言处理的任务. 要知道 BERT 做得好不好, 就让它去跑那 9 个任务, 再去取平均值来代表这个自监督学习模型的好坏. 在语音上, 有个类似的基准语料库——语言处理通用性能基准 (Speech

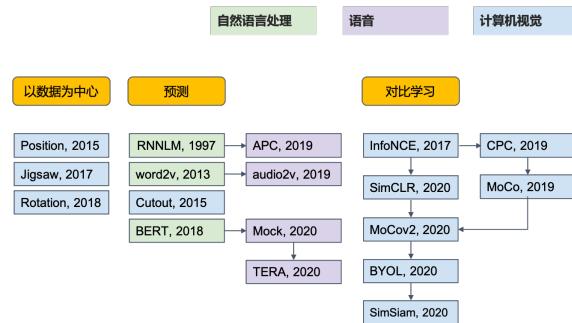


图 5.39 其他领域的自监督学习

processing Universal PERformance Benchmark, SUPERB），可以认为其是一个语音版的 GLUE. 这个基准语料库里面包含了 10 个不同的任务，语音其实有非常多不同的方向，很多人提到语音相关的技术，都只知道语音辨识把声音转成文字，但这并不是语音技术的全貌. 语音其实包含了非常丰富的信息，除了有内容的信息（就是我们说了什么），还有其他的信息，例如这句话是谁说的，这个人说这句话的时候，他的语气是什么样，还有这句话背后它到底有什么样的语意. 所以 SUPERB 里有 10 个不同的任务，这些任务有不同的目的，包括去检测一个模型能够识别内容的能力、识别谁在说话的能力、识别他是怎么说的能力，甚至是识别这句话背后语意的能力，从全方位来检测一个自监督学习的模型在理解人类语言上的能力. 而且有一个工具包 (toolkit) ——s3prl，这个工具包里面就包含了各式各样的自监督学习的模型，它可以做的各式各样语音的下游的任务. 因此自监督学习的技术，不仅能被用在自然语言处理上，还可以用在计算机视觉和语音上.

## 第 6 章 迁移学习

实际应用中很多任务的数据的标注成本很高，无法获得充足的训练数据，这种情况可以使用迁移学习（transfer learning）。假设 A、B 是两个相关的任务，A 任务有很多训练数据，就可以把从 A 任务中学习到的某些可以泛化知识迁移到 B 任务。迁移学习有很多分类，本章介绍了领域自适应（domain adaptation）和领域泛化（domain generalization）。

### 6.1 领域偏移

目前为止，我们学习了很多深度学习的模型，所以训练一个分类器比较简单，比如要训练数字的分类器，给定训练数据，训练好一个模型，应用在测试数据就结束了。

如图 6.1 所示，数字识别这种简单的问题，在基准数据集 MNIST<sup>[1]</sup> 上能做到 99.91% 的正确率<sup>[2]</sup>。但测试数据和训练数据的分布不一样时会导致一些问题。假设训练时数字是黑白的，但测试时数字是彩色的。常见的误区是：虽然数字的颜色不同，但对于模型，其形状是一样的。如果模型能识别出黑白的图片中的数字，其应该也能识别出彩色图片的数字。但实际上，如果使用黑白的数字图像训练一个模型，直接用到彩色的数字上，正确率会非常低。MNIST 数据集中的数字是黑白的，MNIST-M 数据集<sup>[3]</sup> 中的数字是彩色的，如果在 MNIST 数据集上训练，在 MNIST-M 数据集上测试，正确率只有 52.25%<sup>[4]</sup>。一旦训练数据跟测试数据分布不同，在训练数据上训练出来的模型，在测试数据上面可能效果不好，这种问题称为领域偏移（domain shift）。

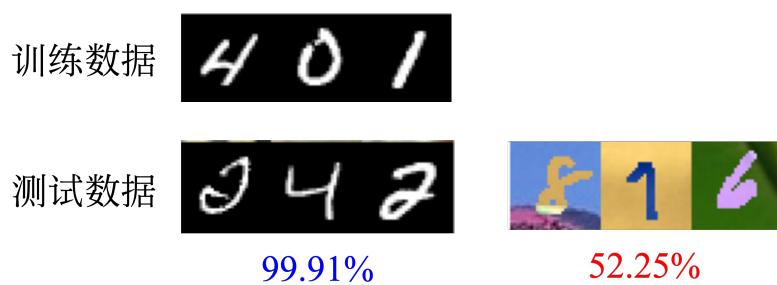


图 6.1 数据分布不同导致的问题

领域偏移其实有很多种不同的类型，模型输入的数据分布有变化的状况是一种类型。另外一种类型是，输出的分布也可能有变化。比如在训练数据上面，可能每一个数字出现的概率都是一样的，但是在测试数据上面，可能每一个数字输出的概率是不一样的，有可能某一个数

字它输出的概率特别大，这也是有可能的。还有一种比较罕见状况的类型是，输入跟输出虽然分布可能是一样的，但它们之间的关系变了。比如同一张图片在训练数据里的标签为“0”，但是在测试数据的标签为“1”。

接下来我们专注于输入数据不同的领域偏移。如图 6.2 所示，接下来我们称测试数据来自目标领域（target domain），训练数据来自源领域（source domain），因此源领域是训练数据，目标领域是测试数据。

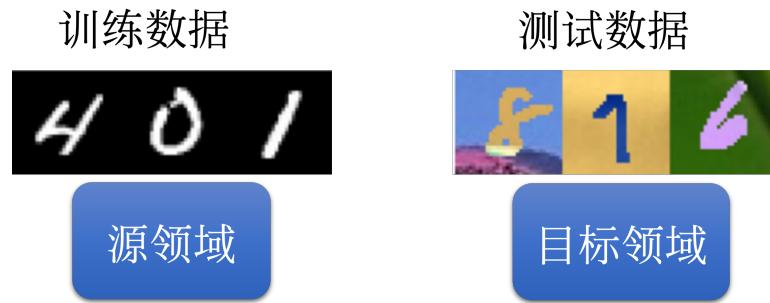


图 6.2 源领域与目标领域

在基准数据集上学习时，很多时候无视领域偏移问题，假设训练数据跟测试数据往往有一样的分布，在很多任务上面都有极高的正确率。但在实际应用时，当训练数据跟测试数据有一点差异时，机器的表现可能会比较差，因此需要领域自适应来提升机器的性能。对于领域自适应，训练数据是一个领域，测试数据是另外一个领域，要把某一个领域上学到的信息用到另外一个领域，领域自适应侧重于解决特征空间与类别空间一致，但特征分布不一致的问题。

## 6.2 领域自适应

接下来介绍下领域自适应，以手写数字识别为例。比如有一堆有标注的训练数据，这些数据来自源领域，用这些数据训练出一个模型，这个模型可以用在不一样的领域。在训练的时候，我们必须要对测试数据所在的目标领域有一些了解。

随着了解的程度不同，领域自适应的方法也不同。如果目标领域上有一大堆有标签的数据，这种情况其实不需要做领域自适应，直接用目标领域的数据训练。如果目标领域上有一点有标签的数据，这种情况可以用领域自适应，可以用这些有标注的数据微调（fine-tuning）在源领域上训练出来的模型。这边的微调跟 BERT 的微调很像，已经有一个在源领域上训练好的模型，只要拿目标领域的数据跑个两、三个回合（epoch）就足够了。在这一种情况下，需

要注意的问题是，因为目标领域的数据量非常少，所以要小心不要过拟合，不要在目标领域的数据上迭代太多次。在目标数据上迭代太多次，可能会过拟合到目标领域的少量数据上，在真正的测试集的表现可能就不好。

为了避免过拟合的情况，有很多的解决方法，比如调小一点学习率。要让微调前、后的模型的参数不要差很多，或者让微调前、后的模型的输入跟输出的关系，不要差很多等等。

下面主要介绍下在目标领域上有大量未标注的数据的这种情况。这种情况其实是很符合实际会发生的情况。比如在实验室里面训练了一个模型，并想要把它用在真实的场景里面，于是将模型上线。上线后的模型确实有一些人来用，但得到的反馈很差，大家嫌弃系统正确率很低。这种情况就可以用领域自适应的技术，因为系统已经上线后会有人使用，就可以收集到一大堆未标注的数据。这些未标注的数据可以用在源领域上训练一个模型，并用在目标领域。

最基本的想法如图 6.3 所示，训练一个特征提取器 (feature extractor)。特征提取器也是一个网络，这个网络输入是一张图片，输出是一个特征向量。虽然源领域与目标领域的图像不一样，但是特征提取器会把它们不一样的部分去除，只提取出它们共同的部分。虽然源领域和目标领域的图片的颜色不同，但特征提取器可以学习到把颜色的信息滤掉，忽略颜色。源领域和目标领域的图片通过特征提取器以后，其得到的特征是没有差异的，分布相同。通过特征提取器可以在源领域上训练一个模型，直接用在目标领域上。通过领域对抗训练 (domain adversarial training) 可以得到领域无关的表示。



图 6.3 通过特征提取器过滤颜色信息

一般的分类器可分成特征提取器和标签预测器 (label predictor) 两个部分。图像的分类

器输入一张图像，输出分类的结果。假设图像的分类器有 10 层，前 5 层是特征提取器，后 5 层是标签预测器。前 5 层可看成特征提取器，一个图像通过前 5 层，其输出是一个向量；如果使用卷积神经网络，其输出是特征映射（feature map），但特征映射“拉直”也可以看做是一个向量，该向量再输入到后面 5 层（标签预测器）来产生类别。

Q：为什么分类器的前 5 层是特征提取器，而不是前 1/2/3/4 层？

A：分类器里面哪些部分算特征提取器，哪些部分算标签预测器，这个是由自己决定的，可以自行调整。

图 6.4 给出了特征提取器和标签预测器的训练过程。对于源领域上标注的数据，把源领域的数据“丢”进去，这跟训练一个一般的分类器一样，它通过特征提取器，再通过标签预测器，可以产生正确的答案。但不一样的地方是，目标领域的一堆数据是没有任何标注的，把这些图片“丢”到图像分类器，把特征提取器的输出拿出来看，希望源领域的图片“丢”进去的特征跟目标领域的图片“丢”进去的特征相同。图 6.4 中蓝色的点表示源领域图片的特征，红色的点表示目标领域图片的特征，通过领域对抗训练让蓝色的点跟红色的点分不出差异。

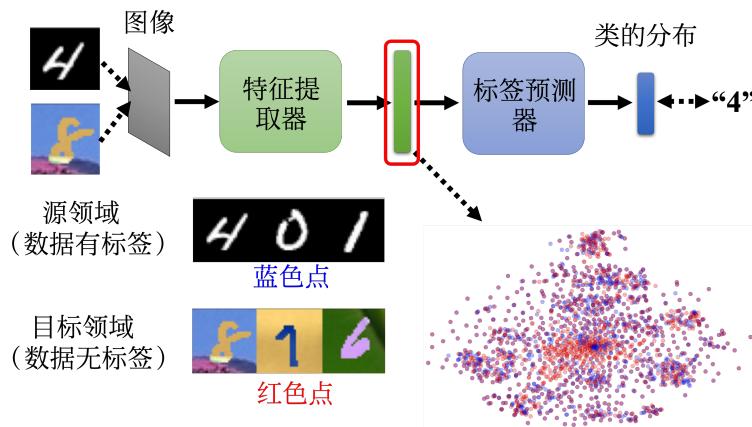


图 6.4 训练特征提取器，让源领域和目标领域的特征无差异<sup>[4]</sup>

如图 6.5 所示，我们要训练一个领域分类器。领域分类器是一个二元的分类器，其输入是特征提取器输出的向量，其目标是判断这个向量是来自于源领域还是目标领域，而特征提取器学习的目标是要去想办法骗过领域分类器。领域对抗训练非常像是生成对抗网络，特征提取器可看成生成器，领域分类器可看成判别器。但在领域对抗训练里面，特征提取器优势太大了，其要骗过领域分类器很容易。比如特征提取器可以忽略输入，永远都输出一个零向量。这样做领域分类器的输入都是零向量，其也无法判断该向量的领域。但标签预测器也需要特

征判断输入的图片的类别，如果特征提取器只会输出零向量，标签预测器无法判断是哪一张图片。特征提取器还是需要产生向量来让标签预测器可以输出正确的预测。因此特征提取器不能永远都输出零向量。

假设标签预测器的参数为  $\theta_p$ ，领域分类器的参数为  $\theta_d$ ，特征提取器的参数为  $\theta_f$ 。源领域的图像是有标签的，所以可以计算它们的交叉熵来得出损失  $L$ 。领域分类器要想办法判断图片是源领域还是目标领域，这就是一个二元分类的问题，该分类问题的损失为  $L_d$ 。我们要去找一个  $\theta_p$ ，它可以让  $L$  越小越好，即

$$\theta_p^* = \min_{\theta_p} L \quad (6.1)$$

我们要去找一个  $\theta_d$ ，它可以让这个  $L_d$  越小越好，即

$$\theta_d^* = \min_{\theta_d} L_d \quad (6.2)$$

标签预测器要让源领域的图像分类越正确越好，领域分类器要让领域的分类越正确越好。而特征提取器站在标签预测器这边，它要做领域分类器相反的事情，所以特征提取器的损失是标签预测器的损失  $L$  减掉领域分类器的损失  $L_d$ ，所以特征提取器的损失是  $L - L_d$ ，找一组参数  $\theta_f$  让  $L - L_d$  的值越小越好，即

$$\theta_f^* = \min_{\theta_f} L - L_d \quad (6.3)$$

假设领域分类器的工作是把源领域跟目标领域分开，根据特征提取器的特征，来判断数据是来自源领域还是目标领域，把源领域和目标领域的两组特征分开。而特征提取器的损失中是  $-L_d$ ，这意味着它要做的事情跟领域分类器相反。如果领域分类器根据某张图片的特征判断这张图片属于源领域，而特征提取器要让领域分类器根据这张图片的特征判断这张图片属于目标领域，这样做也就可以分幵源领域和目标领域的特征。本来领域分类器是让  $L_d$  的值越小越好，特征提取器要让  $L_d$  的值越大越好，其目的都是分幵源领域跟目标领域的特征。以上是最原始的领域对抗训练做法。

领域对抗训练最原始的论文结果，如图 6.6 所示，该论文做了四个从源领域到目标领域的任务。如果用目标领域的图片训练，目标领域的图片测试，结果如表 6.1 所示，每一个任务正确率都是 90% 以上。但如果用源领域训练，目标领域测试，结果比较差。如果使用领域对抗训练，正确率会有明显的提升。

领域对抗训练最早的论文发表在 2015 年的 ICML 上面，其比生成对抗网络还要稍微晚一点，不过它们几乎可以是同时期的作品。

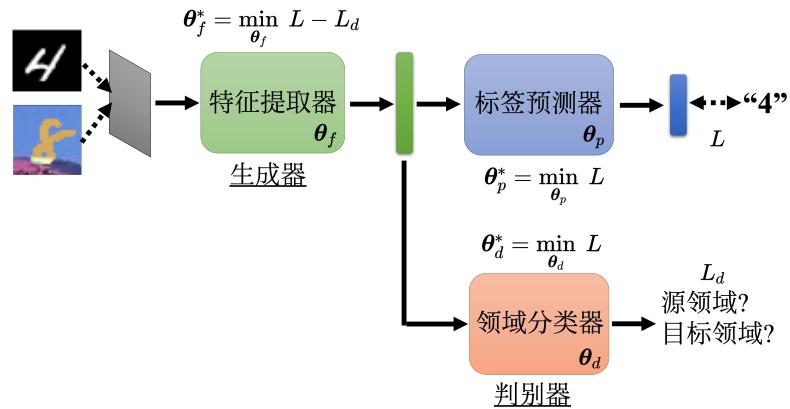


图 6.5 领域对抗训练



图 6.6 领域对的示例

刚才这整套想法，有一个小小的问题。用蓝色的圆圈和三角形表示源领域上的两个类别，用正方形来表示目标领域上无类别标签的数据。可以找一个边界去把源领域上的两个类别分开。训练的目标是要让正方形的分布跟圆圈、三角形合起来的分布越接近越好。在图 6.7 (a) 所示的情况下，红色的点跟蓝色的点是挺对齐在一起的。在图 6.7 (b) 所示的情况下，红色的点跟蓝色的点是分布挺接近的。虽然正方形的类别是未知的，但蓝色的圆圈跟蓝色的三角形的决策边界是已知的，应该让正方形远离决策边界。因此两种情况相比，我们更希望在图 6.7 (b) 的情况发生，而避免让在图 6.7 (a) 的状况发生。

表 6.1 不同源领域和目标领域的数字图像分类的准确率

方法	源领域 目标领域	MNIST	合成数字	SVHN	合成标志
		MNIST-M	SVHN	MNIST	GTSRB
只使用源领域数据训练		55.25%	86.74%	54.90%	79.00%
使用领域对抗训练		76.66%	91.09%	73.85%	88.65%
只使用目标领域数据训练		95.96%	92.20%	99.42%	99.80%

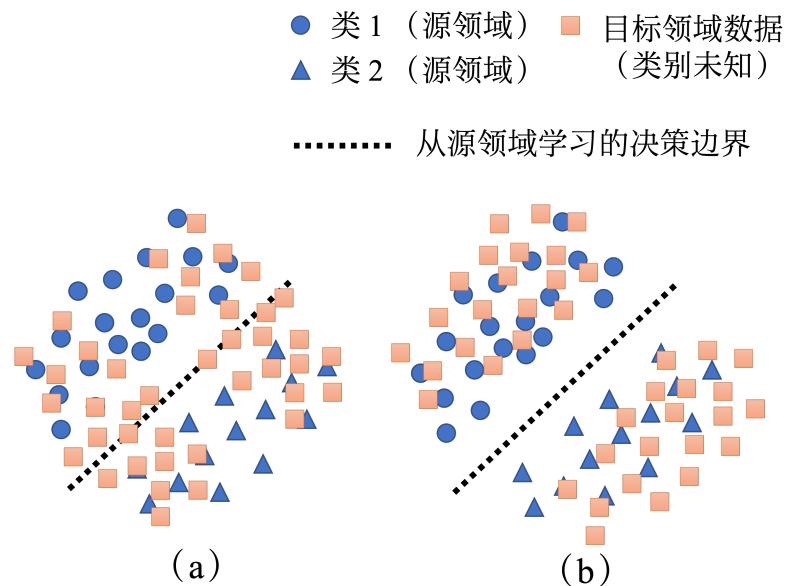


图 6.7 决策边界

让正方形远离边界 (boundary) 最简单的做法如图 6.8 所示。把很多无标注的图片先“丢”到特征提取器，再“丢”到标签预测器，如果输出的结果集中在某个类别上，就是离边界远；如果输出的结果每一个类别非常地接近，就是离边界近。除了上述比较的简单的方法外，还可以使用 DIRT-T<sup>[5]</sup>、最大分类器差异（maximum classifier discrepancy）<sup>[6]</sup> 等方法。这些方法在领域自适应中是不可或缺的。

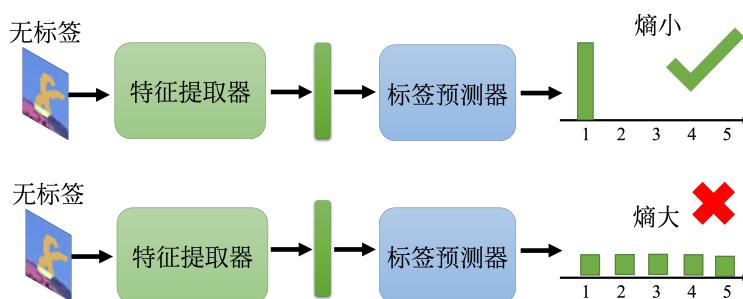


图 6.8 离边界越远越好

目前为止都假设源领域跟目标领域的类别都要是一模一样，比如图像分类，源领域有老虎、狮子跟狗，目标领域也应该要有老虎、狮子跟狗，但实际上目标领域是没有标签的，其里面的类别是未知的。如图 6.9 所示，实线的椭圆圈代表源领域里面有的东西，虚线的椭圆圈代表目标领域里面有的东西。图 6.9 (a) 中源领域里面的东西比较多，目标领域里面的东西比较

少；图 6.9 (b) 中源领域里面的东西比较少，目标领域的东西比较多。图 6.9 (c) 中两者虽然有交集，但是各自都有独特的类别。

强制把源领域跟目标领域完全对齐在一起是有问题的，比如图 6.9 (c) 里面，要让源领域的数据跟目标领域的数据的特征完全匹配，这意味是要让老虎去变得跟狗像，或者让老虎变得跟狮子像，这样老虎这个类别就不能区分了。源领域跟目标领域有不同标签问题的解决方法，可参考论文“Universal Domain Adaptation”<sup>[7]</sup>。

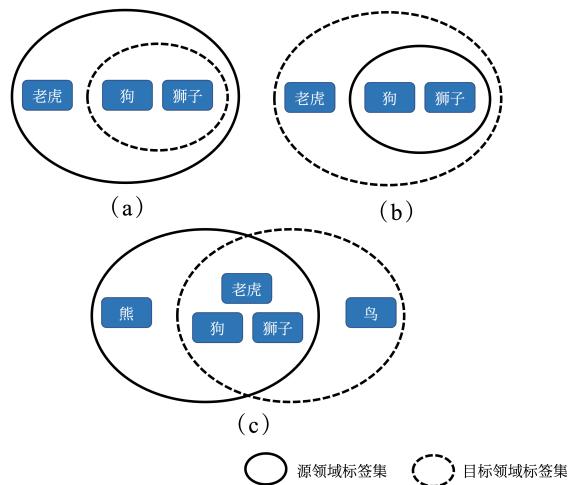


图 6.9 强制完全对齐源领域跟目标领域的问题

Q：如果特征提取器是卷积神经网络，而不是线性层（linear layer）。领域分类器输入是特征映射，特征映射本来就有空间的关系。把两个领域“拉”在一起会不会有影响隐空间（latent space），让隐空间没能学到本来希望它学到的东西？

A：会有影响。领域自适应训练需要同时做好两个方面的事：一方面要骗领域分类器，另一方面是要让分类变正确。即不仅要把两个领域对齐在一起，还要让隐空间的分布是正确的。比如我们认为 1 跟 7 比较像，为了要让分类器做好，特征提取器会让 1 跟 7 比较像。因为要提高标签预测器的性能，所以隐表示（latent representation）里面的空间仍然是一个比较好的隐空间。但如果给领域分类器就是要骗过领域分类器，这件事情的权重太大。模型就会学到只想骗过领域分类器，它就不会产生好的隐空间。

但是有一个可能是目标领域的数据不仅没有标签，而且还很少，比如目标领域只有一张图片，也就无法跟源领域对齐。这种情况可使用测试时训练（Testing Time Training, TTT）方法，读者可参考论文“Test-Time Training with Self-Supervision for Generalization under

Distribution Shifts” [8]。

### 6.3 领域泛化

对目标领域一无所知，并不是要适应到某一个特定的领域上的问题通常称为领域泛化。领域泛化可又分成两种情况。一种情况是训练数据非常丰富，包含了各种不同的领域，测试数据只有一个领域。如图 6.10 (a) 所示，比如要做猫狗的分类器，训练数据里面有真实的猫跟狗的照片、素描的猫跟狗的照片、水彩画的猫跟狗的照片，期待因为训练数据有多个领域，模型可以学到如何弥平领域间的差异。当测试数据是卡通的猫跟狗时，模型也可以处理，具体细节可参考论文“Domain Generalization with Adversarial Feature Learning” [9]。另外一种情况如图 6.10 (b) 所示，训练数据只有一个领域，而测试数据有多种不同的领域。虽然只有一个领域的数据，但可以想个数据增强的方法去产生多个领域的数据，具体可参考论文“Learning to Learn Single Domain Generalization” [10]。

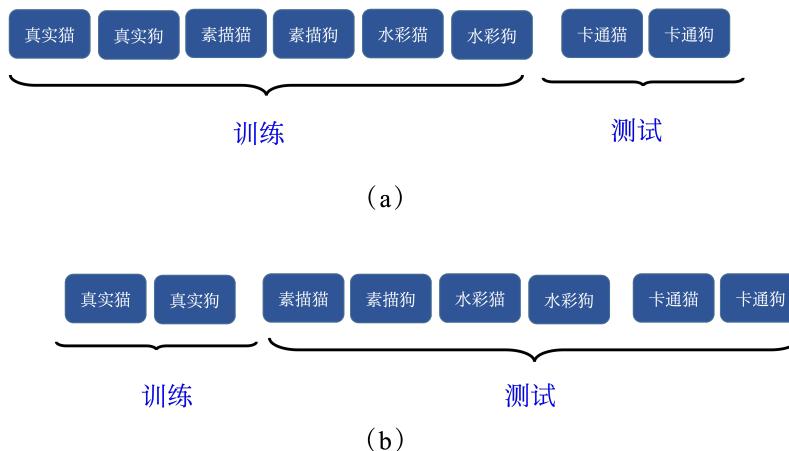


图 6.10 领域泛化示例

## 参考文献

- [1] LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.
- [2] AN S, LEE M, PARK S, et al. An ensemble of simple convolutional neural network models for mnist digit recognition[J]. arXiv preprint arXiv:2008.10400, 2020.

- [3] GANIN Y, USTINOVA E, AJAKAN H, et al. Domain-adversarial training of neural networks[J]. *The journal of machine learning research*, 2016, 17(1): 2096-2030.
- [4] GANIN Y, LEMPITSKY V. Unsupervised domain adaptation by backpropagation[C]// International conference on machine learning. PMLR, 2015: 1180-1189.
- [5] SHU R, BUI H H, NARUI H, et al. A dirt-t approach to unsupervised domain adaptation [J]. arXiv preprint arXiv:1802.08735, 2018.
- [6] SAITO K, WATANABE K, USHIKU Y, et al. Maximum classifier discrepancy for unsupervised domain adaptation[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 3723-3732.
- [7] YOU K, LONG M, CAO Z, et al. Universal domain adaptation[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019: 2720-2729.
- [8] SUN Y, WANG X, LIU Z, et al. Test-time training with self-supervision for generalization under distribution shifts[C]//International conference on machine learning. PMLR, 2020: 9229-9248.
- [9] LI H, PAN S J, WANG S, et al. Domain generalization with adversarial feature learning [C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 5400-5409.
- [10] QIAO F, ZHAO L, PENG X. Learning to learn single domain generalization[C]// Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020: 12556-12565.

## 第 7 章 强化学习

强化学习 (Reinforcement Learning, RL) 有很多的应用，比如玩视频游戏、下围棋等等。之前我们学习过监督学习，可先从监督学习与强化学习的关系来理解强化学习。

### 7.1 监督学习与强化学习的关系

图 7.1 是监督学习 (supervised learning) 的示例，假设我们要训练一个图像的分类器，给定机器一个输入，要告诉机器对应的输出。目前为止，本书所提及的方法都是基于监督学习的方法。自监督学习只是标签，不需要特别雇用人力去标记，它可以自动产生。即使是无监督的方法，比如自编码器 (Auto-Encoder, AE)，其没有用到人类的标记。但事实上还有一个标签，只是该标签的产生不需要耗费人类的力量。

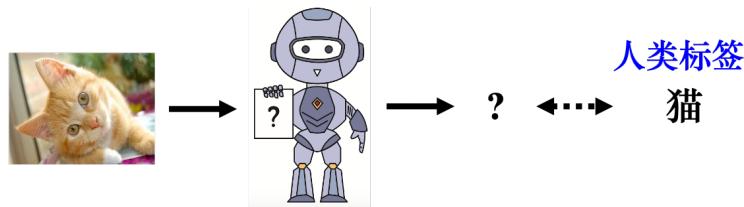


图 7.1 监督学习

但在强化学习里面，给定机器一个输入，最佳的输出也是未知的。如图 7.2 所示，假设要让机器学习下围棋，如果使用监督学习的方法，我们需要告诉机器，给定一个盘势，下一步落子的最佳位置，但该位置也是未知的。即使让机器阅读很多职业高段棋士的棋谱，这些棋谱里面给定某一个盘势，人类下的下一步。但这是一个很好的答案，不一定是最好的位置，因此正确答案是未知的。当正确答案是未知的或者收集有标注的数据很困难，我们可以考虑使用强化学习。强化学习在学习的时候，机器不是一无所知的，虽然其不知道正确的答案，但会跟环境互动得到奖励 (reward)，所以其会知道其输出的好坏。通过与环境的互动，机器可以知道输出的好坏，从而学出一个模型。

如图 7.3 所示，强化学习里面有一个智能体 (agent) 和一个环境 (environment)，智能体会跟环境互动。环境会给智能体一个观测 (observation)，智能体看到这个观测后，它会采取一个动作。该动作会影响环境，环境会给出新的观测，智能体会给出新的动作。

观测是智能体的输入，动作是智能体的输出，所以智能体本身就是一个函数。这个函数的输入是环境给它的观测，输出是智能体要采取的动作。在互动的过程中，环境会不断地给智能

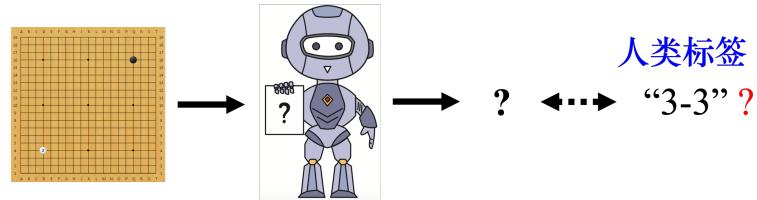


图 7.2 强化学习

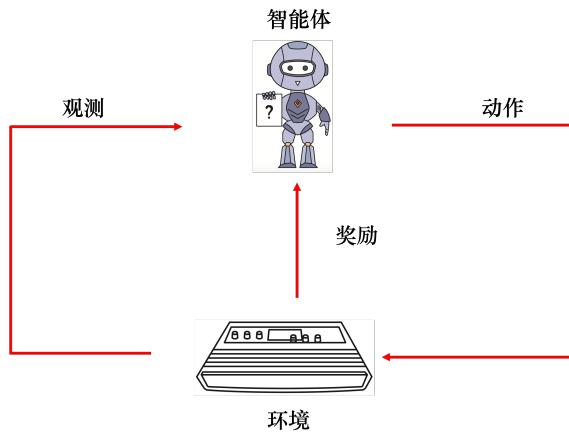


图 7.3 强化学习示意

体奖励，让智能体知道它现在采取的这个动作的好坏。智能体的目标是要最大化从环境获得的奖励总和。

## 7.2 强化学习应用

### 7.2.1 玩视频游戏

强化学习可以用来玩游戏，强化学习最早的几篇论文都是让机器玩《太空侵略者》。在《太空侵略者》里面，如图 7.4 所示，我们要操控太空梭来杀死外星人，可采取的动作有三个：左移、右移和开火，开火击中外星人，外星人就死掉了，我们就得到分数。我们可以躲在防护罩后面来挡住外星人的攻击，如果不小心打到防护罩，防护罩就会消失。在某些版本的《太空侵略者》里面，会有补给包，如果击中补给包，会被加一个很高的分数。分数其实环境给的奖励。当所有的外星人被杀光或者外星人击中母舰的时候，游戏就会终止。

如果用强化学习玩《太空侵略者》，如图 7.5 所示。智能体会去操控摇杆，控制母舰来和外星人对抗。环境是游戏主机，游戏主机操控外星人攻击母舰，所以观测是游戏的画面。输入智能体一个游戏的画面，输出是智能体可以采取的动作。当智能体采取右移动作的时候，不可

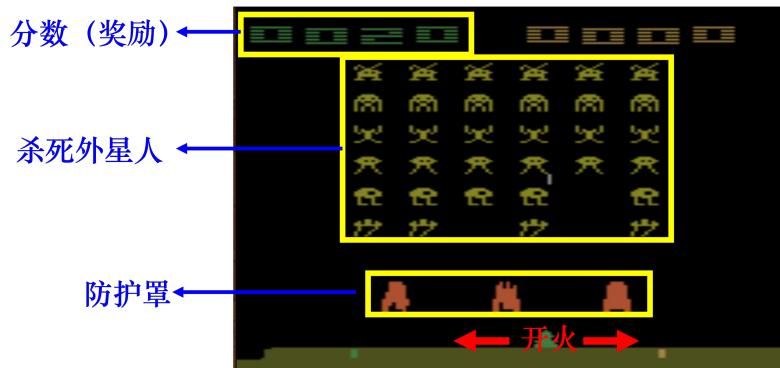


图 7.4 《太空侵略者》游戏

能杀掉外星人，所以奖励为 0。智能体采取一个动作后，游戏的画面就变了，也就有了新的观测。根据新的观测，智能体会决定采取新的动作。假设如图 7.6 所示，智能体采取的动作是开火，这个动作正好杀掉了一只外星人，得到 5 分，奖励等于 5。在玩游戏的过程中，智能体会不断地采取动作得到奖励，我们想要智能体玩这个游戏得到的奖励的总和最大。

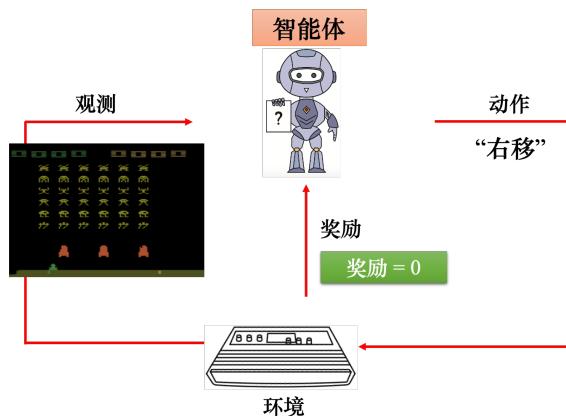


图 7.5 智能体玩《太空侵略者》采取右移的动作

### 7.2.2 下围棋

如果用强化学习下围棋，如图 7.7(a) 所示，智能体是 AlphaGo，环境是 AlphaGo 的人类对手，即棋士。智能体的输入是棋盘上黑子跟白子的位置。一开始，棋盘上是空的。根据该棋盘，智能体要决定下一步的落子，有  $19 \times 19$  个可能性，每个可能性对应棋盘上的一个位置。

如图 7.7(b) 所示，假设现在智能体决定了落子。新的棋盘会输入给棋士，棋士棋士也会再落一子，产生新的观测。智能体看到新的观测就会产生新的动作，这个过程反复进行。在下围棋里面，智能体所采取的动作都无法得到任何奖励，我们可以定义，如果赢了，就得到 1

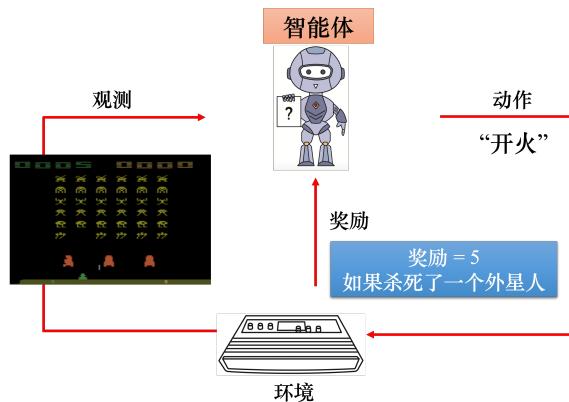


图 7.6 智能体玩《太空侵略者》采取开火的动作

分，如果输了就得到 -1 分，只有整场围棋结束，智能体才能够拿到奖励。智能体学习的目标是要最大化它可能得到的奖励。

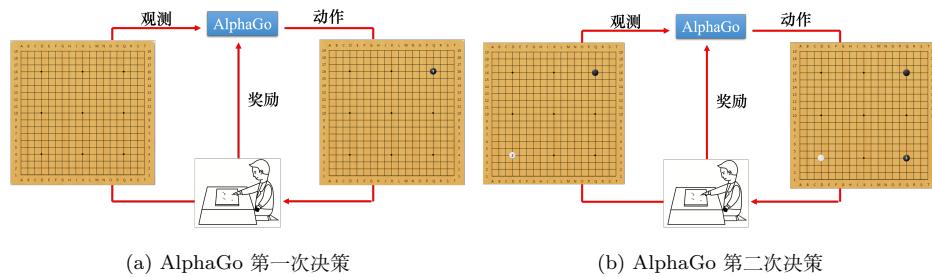


图 7.7 强化学习下围棋

Q: 下围棋是否需要比较好的启发式函数?

A: 在下围棋的时候，假设奖励非常地稀疏 (sparse)，我们可能会需要一个好的启发式函数 (heuristic function)，深蓝的那篇论文，深蓝其实已经在西洋棋上打爆人类了，深蓝就有蛮多启发式函数，它就不是只有下到游戏的中盘，才知道才得到奖励，中间会有蛮多的状况它都会得到奖励。

### 7.3 强化学习框架

强化学习跟机器学习的框架类似，机器学习有三个步骤：第一步是定义函数，函数里面有一些未知变量，这些未知变量是要被学出来的；第二步是定义损失函数；最后一步是优化，即找出未知变量去最小化损失。强化学习也是类似的三个步骤。

### 7.3.1 第 1 步：未知函数

第一个步骤，有未知数的函数是智能体。在强化学习里面，智能体是一个网络，通常称为策略网络（policy network）。在深度学习未被用到强化学习的时候，通常智能体是比较简单的，其不是网络，它可能只是一个查找表（look-up table），告诉我们给定输入对应的最佳输出。网络是一个很复杂的函数，其输入是游戏画面上的像素，输出是每一个可以采取的动作的分数。

网络的架构可以自己设计，只要网络能够输入游戏的画面，输出动作。比如如果输入是一张图片，可以用卷积神经网络来处理。如果我们不要只看当前这一个时间点的游戏画面，而是要看整场游戏到目前为止发生的所有画面，可以考虑使用循环神经网络或Transformer。

如图 7.8 所示，输入游戏画面，策略网络的输出是左移 0.7 分、右移 0.2 分、开火 0.1 分。这类似于分类网络，分类是输入一张图片，输出是决定这张图片的类别，网络会给每一个类别一个分数。分类网络的最后一层是 softmax 层，每个类别有个分数，这些分数的总和是 1。机器会决定采取哪一个动作，取决于每一个动作的分数。常见的做法是把这个分数当做一个概率，按照概率采样，随机决定要采取的动作。比如图 7.8 中的例子里面，智能体有 70% 的概率会采取左移，20% 的概率会采取右移，10% 的概率会采取开火。

Q：为什么不采取分数最大的动作？

A：我们可以采取左的动作，但一般都是使用随机采样。采取有一个好处：看到同样的游戏画面，机器每一次采取的动作，也会略有不同，在很多的游戏里面随机性是很重要的，比如玩石头、剪刀、布游戏，如果智能体总是出石头，很容易输。但如果有一些随机性，就比较不容易输。

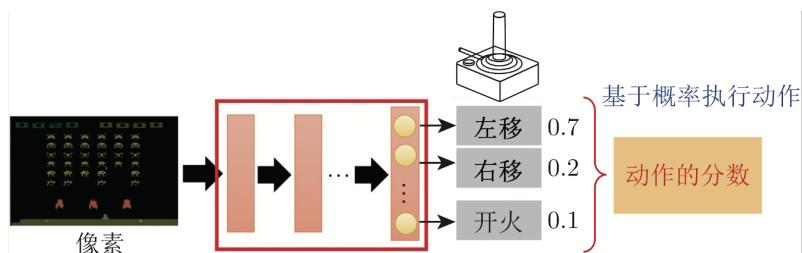


图 7.8 策略网络

### 7.3.2 第 2 步：定义损失

接下来第二步是定义强化学习中的损失。如图 7.9 所示，首先有一个初始的游戏画面  $s_1$ ，该游戏画面会被作为智能体的输入。智能体输出了一个动作  $a_1$  右移，得到 0 分的奖励。接下来会看到新的游戏画面  $s_2$ ，根据  $s_2$ ，智能体会采取新的动作  $a_2$  开火，假设开火恰好杀死一个外星人，智能体得到 5 分的奖励。

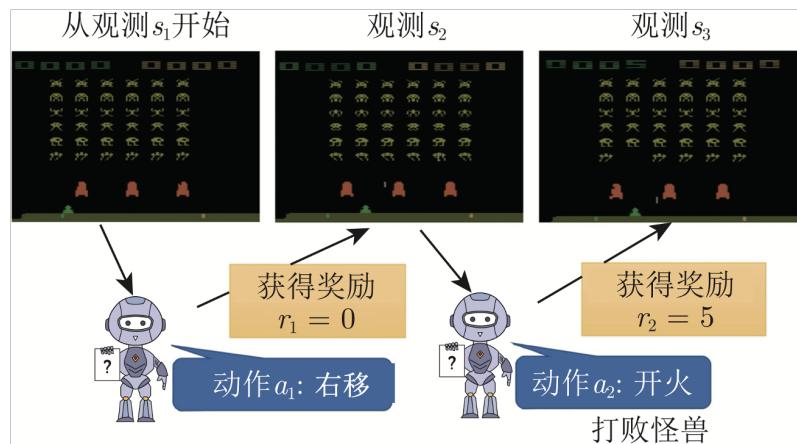


图 7.9 玩视频游戏的例子

如图 7.10 所示，智能体采取开火这个动作以后，接下来会有新的游戏画面，机器又会采取新的动作，这个互动的过程会反复持续下去，直到机器在采取某一个动作以后，游戏结束了。从游戏开始到结束的整个过程称为一个回合 (episode)。整个游戏过程中，机器会采取非常多的动，每一个动作会有奖励，所有的奖励的总和称为整场游戏的总奖励 (total reward)，也称为回报 (return)。回报是从游戏一开始得到的  $r_1$ ，一直累加到游戏最后结束的时候得到的  $r_t$ 。假设这个游戏里面会互动  $T$  次，得到一个回报  $R$ 。我们想要最大化回报，这是训练的目标。回报和损失不一样，损失是要越小越好，回报是要越大越好。如果把负的回报当做损失，回报是越大越好，负的回报是越小越好。

奖励是指智能体采取某个动作的时候，立即得到的反馈。整场游戏里面所有奖励的总和才是回报。

### 7.3.3 第 3 步：优化

图 7.11 给出了智能体与环境互动的示例，环境输出一个观测  $s_1$ ， $s_1$  会变成智能体的输入；智能体接下来输出  $a_1$ ， $a_1$  又变成环境的输入；环境呢，看到  $a_1$  以后，又输出  $s_2$ 。智能

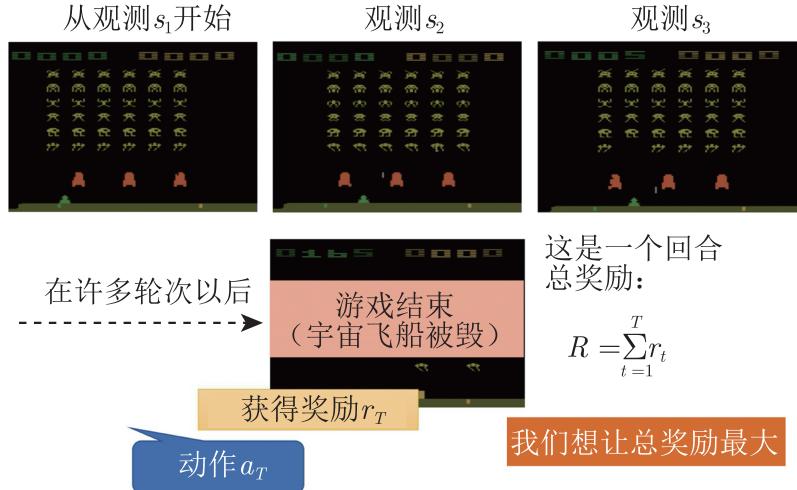


图 7.10 回报的例子

体和环境的互动会反复进行，直到满足游戏中止的条件。在一场比赛中，我们把状态和动作全部组合起来得到的一个序列称为轨迹（trajectory） $\tau$ ，即

$$\tau = \{s_1, a_1, s_2, a_2, \dots, s_t, a_t\} \quad (7.1)$$

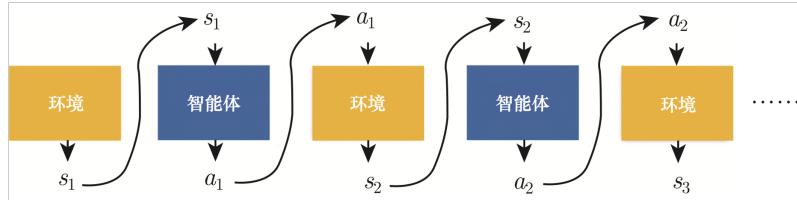


图 7.11 智能体与环境互动

如图 7.12 所示，智能体与环境互动的过程中，会得到奖励，奖励可以看成一个函数。奖励函数有不同的表示方法，在有的游戏里面，智能体采取的动作可以决定奖励。但通常我们在决定奖励的时候，需要动作和观测。比如每次开火不一定能得到分数，外星人在母舰前面，开火要击到外星人才有分数。因此通常定义奖励函数的时候，需要同时看动作跟观测，因此奖励函数的输入是状态和动作。比如图 7.12 中奖励函数的输入是  $a_1$  跟  $s_1$ ，输出是  $r_1$ 。所有奖励的总和是回报，即

$$R(\tau) = \sum_{t=1}^T r_t \quad (7.2)$$

我们需要最大化回报，因此优化问题为：学习网络的参数让回报越大越好，可以通过梯度上升（gradient ascent）来最大化回报。但是强化学习困难的地方是，这不是一般的优化的问题，跟

一般的网络训练不太一样。第一个问题是，智能体的输出是有随机性的，比如图 7.12 中的  $a_1$  是用采样产生的，同样的  $s_1$  每次产生的  $a_1$  不一定一样。假设环境、智能体跟奖励合起来当成一个网络，这个网络不是一般的网络，这个网络里面是有随机性的。这个网络里面的某一个层是每次的输出是不一样的。

另外一个问题就是环境跟奖励是一个黑盒子，其很有可能具有随机性。比如环境是游戏机，游戏机里面发生的事情是未知的。在游戏里面，通常奖励是一条规则：给定一个观测和动作，输出对应的奖励。但对有一些强化学习的问题里面，奖励是有可能有随机性的，比如玩游戏也是有随机性的。给定同样的动作，游戏机的回应不一定是一样的。如果是下围棋，即使智能体落子的位置是相同的，对手的回应每次可能也是不一样的。由于环境和奖励的随机性，强化学习的优化问题不是一般的优化的问题。

强化学习的问题是如何找到一组网络参数来最大化回报。这跟生成对抗网络有异曲同工之妙。在训练生成器（generator）的时候，生成器与判别器（discriminator）会接在一起，我们希望调整生成器的参数，让判别器的输出越大越好。在强化学习里面，智能体就像是生成器，环境跟奖励就像是判别器，我们要调整生成器的参数，让判别器的输出越大越好。但在生成对抗网络里面判别器也是一个神经网络，我可以用梯度下降来训练生成器，让判别器得到最大的输出。但是在强化学习的问题里面，奖励跟环境不是网络，不能用一般梯度下降的方法调整参数来得到最大的输出，所以这是强化学习跟一般机器学习不一样的地方。

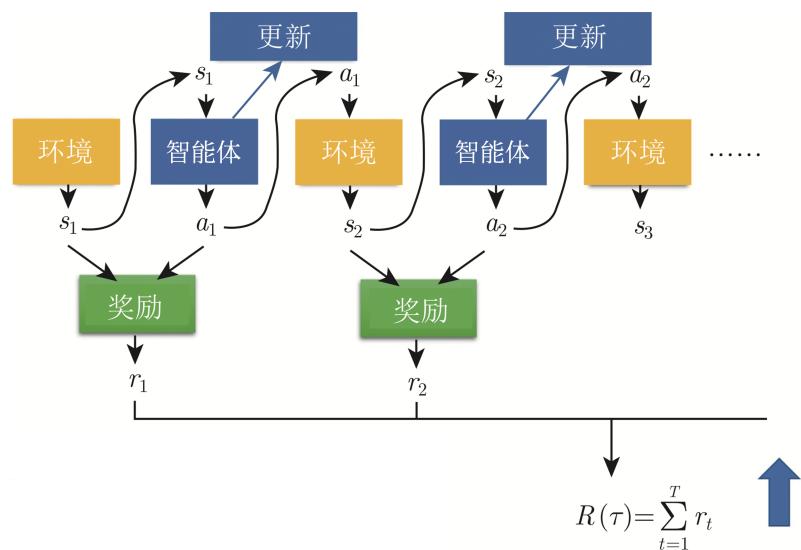


图 7.12 期望的奖励

让一个智能体在看到某一个特定观测的时候，采取某一个特定的动作，这可以看成一个

分类的问题，如图 7.13 所示，比如给定智能体的输入是  $s$ ，让其输出动作  $\hat{a}$ ，假设  $a$  是左移，我们要教智能体看到这个游戏画面左移就是对的。 $s$  是智能体的输入， $a$  就是标签，即标准答案（ground truth）。接下来可以计算智能体的输出跟标准答案之间的交叉熵  $e$ ，学习  $\theta$  让损失（交叉熵）最小，让智能体的输出跟标准答案越接近越好。

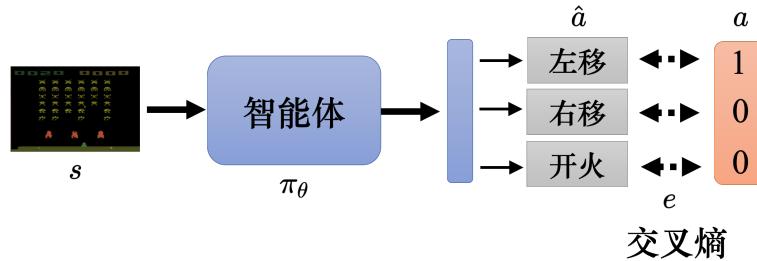


图 7.13 使用交叉熵作为损失

如果想要让智能体看到某一个观测，不要采取某一个动作，只需要在定义损失的时候使用负的交叉熵。如果希望智能体采取动作  $a$ ，可定义损失  $L$  等于交叉熵  $e$ 。如果希望智能体不采取动作  $a$ ，可定义损失  $L$  等于  $-e$ 。假设要让智能体看到  $s$  的时候采取  $a$ ，看到  $s'$  的时候不要采取  $a'$ 。如图 7.14 所示，给定观测  $s'$ ，标准答案为  $\hat{a}$ ，对这两个标准答案可计算交叉熵  $e_1$  跟  $e_2$ 。损失可定义为  $e_1 - e_2$ ，即让  $e_1$  越小越好， $e_2$  越大越好。然后找一个  $\theta$  最小化损失，得到  $\theta^*$ ，如式 (7.3) 所示。因此给定智能体适当的标签和损失来控制控制智能体的输出。

$$\theta^* = \arg \min_{\theta} L \quad (7.3)$$

如图 7.15 所示，如果我们要训练一个智能体，需要收集一些训练数据，希望在  $s_1$  的时候采取  $a_1$ ，在  $s_2$  的时候不要采取  $a_2$ 。这个训练过程类似于训练一个图像的分类器， $s$  可看成图像， $a$  可看成标签，只是有的动作是想要被采取的，有的动作是不想要被采取的。收集一堆这种数据，定义一个损失函数：

$$L = +e_1 - e_2 + e_3 \cdots - e_N \quad (7.4)$$

接着最小化损失函数，这样我们可以训练一个智能体，期待它执行的动作是我们想要的。而且可以更进一步，每一个动作并不是有想要执行跟不想要执行，而且有程度的差别。如果每一个动作就是要执行或不执行，这是一个二分类的问题，可以用  $+1$  和  $-1$  来表示。

但如果考虑动作执行程度的差别，每一个状态-动作对（state-action pair）对应一个分数，这个分数代表希望机器在看到  $s_1$  的时候，执行动作  $a_1$  的程度。比如图 7.16 中第一笔数据的

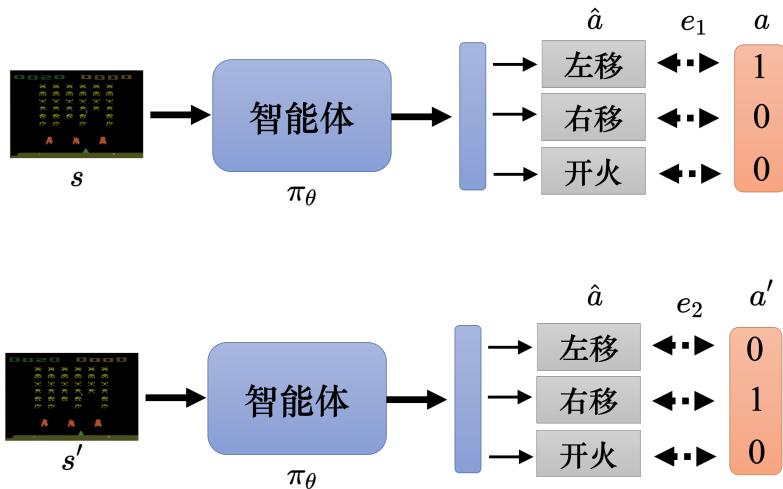


图 7.14 定义合适的损失

**训练数据**

$\{s_1, a_1\}$	+1	对
$\{s_2, a_2\}$	-1	错
$\{s_3, a_3\}$	+1	对
$\vdots$	$\vdots$	
$\{s_N, a_N\}$	-1	错

图 7.15 收集训练数据

分数定为  $+1.5$ ，第三笔数据的分为  $+0.5$ 。这代表我们期待机器看到  $s_1$ ，它可以做  $a_1$ ，看到  $s_3$ ，它可以做  $a_3$ ，但是我们期待它看到  $s_1$  的时候，做  $a_1$  的这个期待更强烈一点，比看到  $s_3$  做  $a_3$  的期待更强烈一点。我们希望它在看到  $s_2$  的时候，不要做  $a_2$ ，期待它看到  $s_N$  的时候，不要做  $a_N$ 。有了这些数据，我们可以定义如式 (7.5) 所示的损失函数，之前的交叉熵本来要乘  $+1$  或  $-1$ ，现在改成乘上  $A_i (i = 1, \dots, n)$ ，通过  $A_i$  来控制每一个动作执行的程度。

$$L = \sum A_n e_n \quad (7.5)$$

综上，强化学习可分为三个阶段，只是在优化的步骤跟一般的方法不同，其会使用策略梯度 (policy gradient) 等优化方法。接下来的难点就是，如何定义  $A$ ，先介绍最容易想到的 4 个版本。

## 训练数据

$\{s_1, a_1\}$	$A_1$	+1.5
$\{s_2, a_2\}$	$A_2$	-0.5
$\{s_3, a_3\}$	$A_3$	+0.5
⋮	⋮	⋮
$\{s_N, a_N\}$	$A_N$	-10

图 7.16 对每个状态-动作对分配不同的分数

## 7.4 评价动作的标准

### 7.4.1 版本 0

智能体跟环境做互动可以收集一些训练数据（状态-动作对）。智能体可以先看成随机的智能体，它执行的动作都是随机的，每一个  $s$  执行的动作  $a$  都记录下来。通常收集数据不会只把智能体跟环境做一个回合，通常需要做多个回合才收集到足够的数据。接下来评价每一个动作的好坏，评价完以后，可以拿评价的结果来训练智能体。 $A$  可评价在每个状态，智能体采取某一个动作的好坏。最简单的评价方式是，假设在某一个状态  $s_1$ ，智能体执行  $a_1$ ，得到奖励  $r_1$ 。如果奖励是正的，代表该动作是好的；如果奖励是负的，代表该动作是不好的。因此，如图 7.17 所示，奖励可当成  $A$ ， $A_1 = r_1$ ， $A_2 = r_2$ ，以此类推。

以上是版本 0，但其并不是一个好的版本。因为把奖励设为  $A$ ，这会让智能体变得短视，不会考虑长期收益。每一个动作，其实都会影响互动接下来的发展。比如智能体在  $s_1$  执行  $a_1$  得到  $r_1$ ，这个并不是互动的全部。因为  $a_1$  影响了导致了  $s_2$ ， $s_2$  会影响到接下来会执行  $a_2$ ，也影响到接下来会产生  $r_2$ ，所以  $a_1$  也会影响到会不会得到  $r_2$ ，每一个动作并不是独立的，每一个动作都会影响到接下来发生的事情。

而且在跟环境做互动的时候，有一个问题叫做延迟奖励（delayed reward），即牺牲短期的利益以换取更长期的利益。比如在《太空侵略者》的游戏里面，智能体要先左右移动一下进行瞄准，射击才会得到分数。而左右移动是没有任何奖励的，其得到的奖励是零。只有射击才会得到奖励，但是并不代表左右移动是不重要的，先需要左右移动进行瞄准，射击才会有效果，所以有时候我们会牺牲一些近期的奖励，而换取更长期的奖励。如果使用版本 0，左移和右移的奖励为 0，开火的奖励为正，智能体会觉得只有开火是对的，它会一直开火。

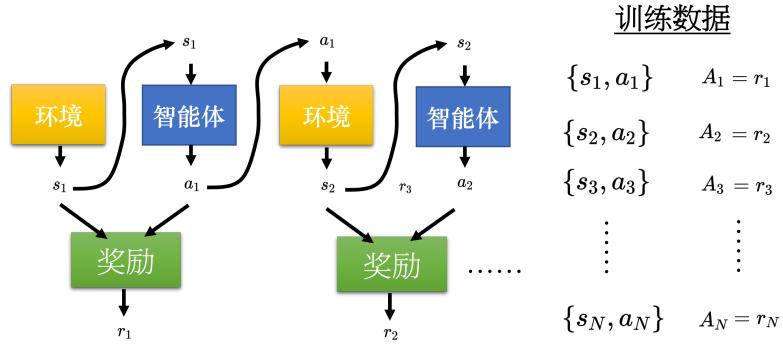


图 7.17 短视的版本

#### 7.4.2 版本 1

在版本 1 里面，把未来所有的奖励加起来即可得到累积奖励  $G$ ，用其来评估一个动作的好坏，如图 7.18 所示。 $G_t$  是从时间点  $t$  开始， $r_t$  一直加到  $r_N$ ，即

$$G_t = \sum_{i=t}^N r_i \quad (7.6)$$

比如  $G_1$ 、 $G_2$ 、 $G_3$  的定义为

$$\begin{aligned} G_1 &= r_1 + r_2 + r_3 + \dots + r_N \\ G_2 &= r_2 + r_3 + \dots + r_N \\ G_3 &= r_3 + \dots + r_N \end{aligned} \quad (7.7)$$

$a_1$  的好坏不是取决于  $r_1$ ，而是取决于  $a_1$  之后所有发生的事情，即执行完  $a_1$  以后所有得到的奖励  $G_1$ ， $A_1$  等于  $G_1$ 。使用累积奖励可以解决版本 0 遇到的问题，因为可能右移移动以后进行瞄准，接下来开火就有打中外星人。因此右移也有累积奖励，虽然右移没有立即的奖励。假设  $a_1$  是右移， $r_1$  可能是 0，但接下来可能会因为右移才能打到外星人，累积的奖励就会正的，因此右移也是一个好的动作。

但是版本 1 也有问题，假设游戏非常长，把  $r_N$  归功于  $a_1$  也不太合适。当智能体采取动作  $a_1$ ，立即有影响的是  $r_1$ ，接下来才会影响到  $r_2$  和  $r_3$ 。假设该过程非常长，智能体采取动作  $a_1$  导致可以得到  $r_N$  的可能性很低，版本 2 解决了该问题。

#### 7.4.3 版本 2

版本 2 的累积的奖励用  $G'$  来表示累积的奖励， $G'_t$  的定义如式 (7.8) 所示， $r$  前面乘一个折扣因子 (discount factor)  $\gamma$ 。折扣因子也会设一个小于 1 的值，比如 0.9 或 0.99 之类的。

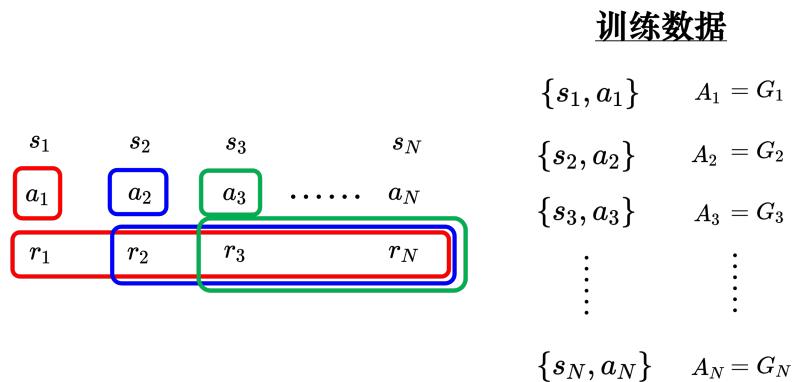


图 7.18 使用累积奖励作为评价标准

$$G'_t = \sum_{i=t}^N \gamma^{i-t} r_i \quad (7.8)$$

图 7.19 给出了折扣累积奖励的示例， $G'_1$  与不同，其奖励函数可以定义为

$$G'_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots \quad (7.9)$$

距离采取动作  $a_1$  越远， $\gamma$  的次方数越多。 $r_2$  距离  $a_1$  一步，就乘个  $\gamma$ ， $r_3$  距离  $a_1$  两步，就乘  $\gamma$  平方，一直加到  $r_N$  的时候， $r_N$  对  $G'_1$  几乎没有影响，因为  $\gamma$  乘了非常多次， $\gamma$  是一个小于 1 的值，即使  $\gamma$  设 0.9， $0.9^{10}$  也很小了。

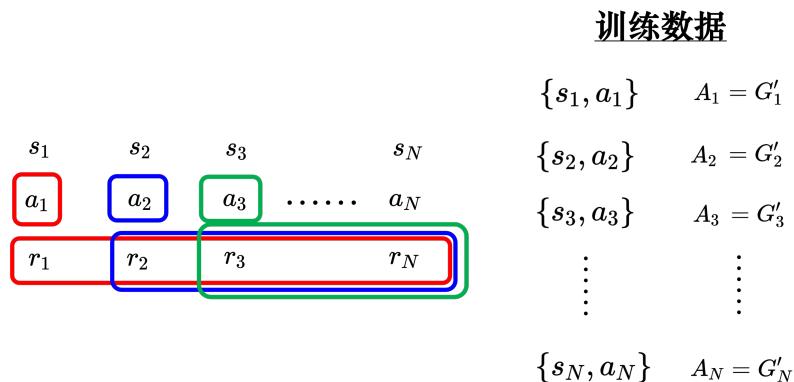


图 7.19 使用折扣奖励作为评价标准

所以加入折扣因子可以把给离  $a_1$  比较近的奖励比较大的权重，给离  $a_1$  比较远的那些奖励比较小的权重。因此新的  $A$  等于  $G'_1$ ，离采取的动作越远的奖励， $\gamma$  就被乘越多次，它对  $G'$  的影响就越小。

Q: 越早的动作累积到的分数越多，越晚的动作累积的分数越少吗？

A: 在游戏等情况下，越早的动作就会累积到越多的分数，因为较早的动作对接下来的影响比较大，其是需要特别在意的。到游戏的终局，外星人基本都没了，智能体做的事情对结果影响都不大。有很多种不同的方法决定  $A$ ，如果不想要较早的动作累积分数比较大，完全可以改变  $A$  的定义。

Q: 折扣累积奖励是不是不适合用在围棋之类的游戏（围棋这种游戏只有结尾才有分数）？

A: 折扣累积奖励可以处理这种结尾才有分数的游戏。假设只有  $r_N$  有分数，其它  $r$  都是 0。智能体采取一系列动作，只要最后赢了，这一系列动作都是好的；如果最后输了，这一系列动作都是不好的。最早版本的 AlphaGo 采用这种方法训练网络，但它还有一些其它的方法，比如价值网络（value network）等等。

#### 7.4.4 版本 3

因为好或坏是相对的，假设在游戏里面，我们每次采取一个行动的时候，最低分预设是 10 分，其实得到 10 分的奖励算是差的，奖励是相对的。用  $G'$  来表示评估标准会有一个问题：假设游戏里面，可能永远都是拿到正的分数，每一个动作都会给出正的分数，只是大小的不同， $G'$  算出来都会是正的，有些动作其实是不好的，但是我们仍然会鼓励模型去采取这些动作。因此我们需要做一下标准化，最简单的方法是把所有的  $G'$  都减掉一个基线（baseline） $b$ ，让  $G'$  有正有负，特别高的  $G'$  让它是正的，特别低的  $G'$  让它是负的，如图 7.20 所示。

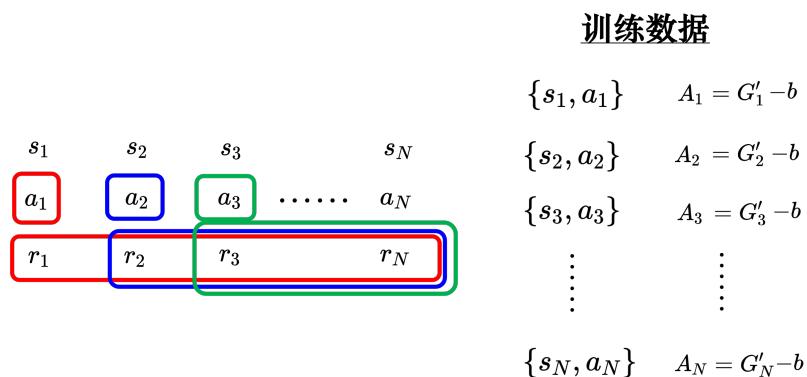


图 7.20 减去基线

策略梯度算法中的评价标准就是  $G' - b$ , 其详细过程如算法 7.1 所示。首先要随机初始化智能体, 给智能体一个随机初始化的参数  $\theta_0$ 。接下来进入训练迭代阶段, 假设要跑  $T$  个训练迭代。一开始智能体什么都不会, 其采取的动作都是随机的, 但它会越来越好。智能体去跟环境做互动, 得到一大堆的状态-动作对。接下来我们就要进行评价, 用  $A_1$  到  $A_N$  来决定说这些动作的好坏。接下来定义损失, 并更新模型, 更新的过程跟梯度下降一模一样。计算  $L$  的梯度, 前面乘上学习率 (learning rate)  $\eta$ , 接着用该梯度更新模型, 把  $\theta_{i-1}$  更新成  $\theta_i$ 。

---

### 算法 7.1 策略梯度

---

```

1 初始化智能体网络参数  $\theta$ ;
2 for  $i = 1$  to  $T$  do
3   使用智能体  $\pi_{\theta_{i-1}}$  进行交互;
4   获取数据  $\{s_1, a_1\}, \{s_2, a_2\}, \dots, \{s_N, a_N\}$ ;
5   计算  $A_1, A_2, \dots, A_N$ ;
6   计算损失  $L$ ;
7    $\theta_i \leftarrow \theta_{i-1} - \eta \nabla L$ 
8 end

```

---

在一般的训练中, 收集数据都是在训练迭代之外, 比如有一堆数据, 用这堆数据拿来做训练, 更新模型很多次, 最后得到一个收敛的参数, 拿这个参数来做测试。但在强化学习不同, 其在训练迭代的过程中收集数据。

如图 7.21 所示, 可以用一个图像化的方式来表示强化学习训练的过程。训练数据中有很多某个智能体的状态-动作对, 对于每个状态-动作对, 可以使用评价  $A$  来判断每一个动作的好坏。通过训练数据训练智能体, 使用评价  $A$  定义出一个损失  $L$ , 并更新参数一次。一旦更新完一次参数, 接下来重新要收集数据了才能更新下一次参数, 所以这就是往往强化学习训练过程非常耗时的原因。强化学习每次更新完一次参数以后, 数据就要重新再收集一次, 再去更新参数。如果参数要更新 400 次, 数据就要收集 400 次, 这个过程非常消耗时间。

策略梯度算法中, 每次更新完模型参数以后, 需要重新再收集数据。如算法 7.1 所示, 这些数据是由  $\pi_{\theta_{i-1}}$  所收集出来的, 这是  $\pi_{\theta_{i-1}}$  跟环境互动的结果, 这个是  $\pi_{\theta_{i-1}}$  的经验, 这些经验可以拿来更新  $\pi_{\theta_{i-1}}$ , 可以拿来更新  $\pi_{\theta_{i-1}}$  的参数, 但它不一定适合拿来更新  $\pi_{\theta_i}$  的参数。

举个例子, 进藤光跟佐为下棋, 进藤光下了小马步飞 (棋子斜放一格叫做小马步飞, 斜放

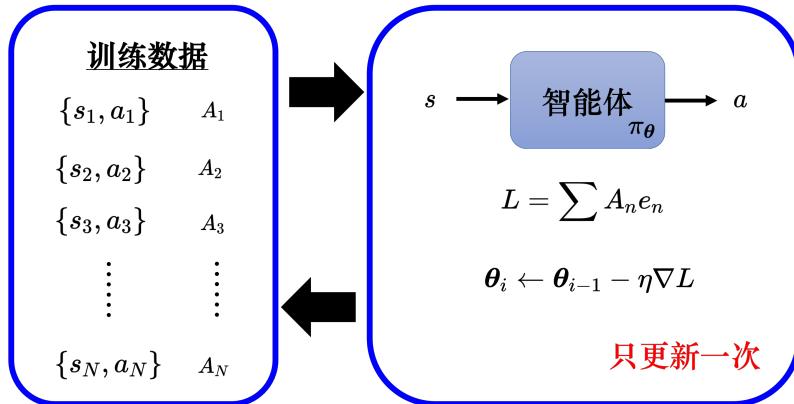


图 7.21 强化学习训练过程

好几格叫做大马步飞)。下完棋以后，佐为让进藤光这种情况不要下小马步飞，而是要下大马步飞。如果大马步飞有 100 手，小马步飞只有 99 手。之前走小马步飞是对的，因为小马步飞的后续比较容易预测，也比较不容易出错，大马步飞的下法会比较复杂。但进藤光假设想要变强的话，他应该要学习下大马步飞，或者是进藤光变得比较强以后，他应该要下大马步飞。同样是下小马步飞，对不同棋力的棋士来说，其作用是不一样的。对于比较弱的进藤光，下小马步飞是对的，因为这样比较不容易出错，但对于已经变强的进藤光来说，下大马步飞比较好。因此同一个动作，对于不同的智能体而言，它的好坏是不一样的。

如图 7.22 所示，假设用  $\pi_{\theta_{i-1}}$  收集了一堆的数据，这些数据只能用来训练  $\pi_{\theta_{i-1}}$ ，不能用来训练  $\pi_{\theta_i}$ 。假设  $\pi_{\theta_{i-1}}$  和  $\pi_{\theta_i}$  在  $s_1$  都会采取  $a_1$ ，但之后到了  $s_2$  以后，它们采取的动作可能就不一样了。因此  $\pi_{\theta_i}$  跟  $\pi_{\theta_{i-1}}$  收集的数据根本就不一样。使用  $\pi_{\theta_{i-1}}$  的收集数据来评估  $\pi_{\theta_i}$  接下来会得到的奖励其实是不合适的。如果收集数据的智能体跟被训练的智能体是同一个智能体，当智能体更新以后，就要重新去收集数据，这就是强化学习非常花时间的原因，异策略学习 (off-policy learning) 可以解决该问题。

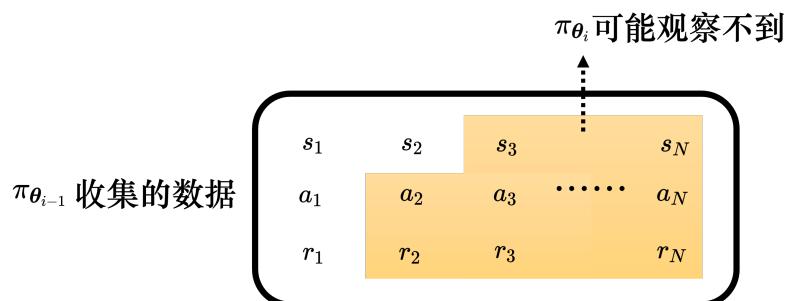


图 7.22 不同智能体收集的数据不能共用

同策略学习 (on-policy learning) 是指要训练的智能体跟与环境互动的智能体是同一个智能体，比如策略梯度算法就是同策略的学习算法。而在异策略学习中，与跟环境互动的智能体跟训练的智能体是两个智能体，要训练的智能体能够根据另一个智能体与环境互动的经验进行学习，因此异策略学习不需要一直收集数据。同策略学习每更新一次参数就要收集一次数据，比如更新 400 次参数，就要收集 400 次数据，而异策略学习收集一次数据，可以更新参数很多次。

探索 (exploration) 是强化学习训练的过程中一个非常重要的技巧。智能体在采取动作的时候是有一些随机性的。随机性非常重要，很多时候随机性不够会训练不起来。假设有一些动作从来没被执行过，这些动作的好坏是未知的，很有可能会训练不出好的结果。比如假设一开始初始的智能体永远都只会右移移动，它从来没有开火，动作开火的好坏就是未知的。只有某一个智能体试图做开火这件事得到奖励，才有办法去评估这个动作的好坏。在训练的过程中，与环境互动的智能体本身的随机性是非常重要的，其随机性大一点，才能够收集到比较多的数据，才不会有一些状况的奖励是未知的。

为了要让智能体的随机性大一点，甚至在训练的时候会刻意加大它的随机性。比如智能体的输出是一个分布，可以加大该分布的熵 (entropy)，让其在训练的时候，比较容易采样到概率比较低的动作。或者会直接在这个智能体的参数上面加噪声，让它每一次采取的动作都不一样。

#### 7.4.5 版本 3.5

在介绍版本 3.5 之前，先介绍下 Critic 及其训练方法。

与环境交互的网络可称为 Actor (演员，策略网络)，而 Critic (评论员，价值网络) 的工作是要来评估一个智能体的好坏。假设有一个智能体的参数为  $\pi_\theta$ ，Critic 的工作是要评估如果智能体看到某个观测，看到某一个游戏画面，接下来它可能会得到的奖励。Critic 有好多种不同的变形，有的 Critic 是只看游戏画面来判断，有的 Critic 是说看到某一个游戏画面，Actor 采取某一个动作，在这两者都具备的前提下，接下来会得到的奖励。

Critic 也被称为价值函数 (value function)，可以用  $V_{\pi_\theta}(s)$  来表示。上标  $\pi_\theta$  代表这个  $V$  观测的 Actor 的策略为  $\pi_\theta$ 。如图 7.23(a) 所示，其输入是  $s$ ， $V_{\pi_\theta}$  就是一个函数，输出是一个标量  $V_{\pi_\theta}(s)$ 。价值函数  $V_{\pi_\theta}(s)$  表示智能体  $\pi_\theta$  看到观测接下来得到的折扣累积奖励 (discounted cumulated reward)  $G'$ 。价值函数看到 图 7.23(b) 所示的游戏画面，直接预测接下来应该会得

到很高的累积奖励，因为该游戏画面里面还有很多的外星人，假设智能体很厉害，接下来它就会得到很多的奖励。图 7.23(c) 所示的画面已经是游戏的残局，游戏快结束了，剩下的外星人不多了，可以得到的奖励就比较少。价值函数跟其观察的智能体是有关系的，同样的观测，不同的智能体得到的折扣累积奖励应该不同。

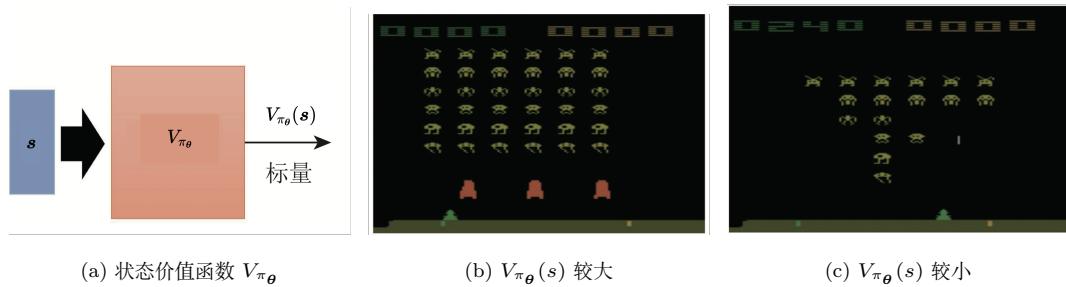


图 7.23 玩《太空侵略者》

Critic 有两种常用的训练方法：蒙特卡洛和时序差分。智能体跟环境互动很多轮会得到一些游戏的记录。从这些游戏记录可知，看到游戏画面  $s_a$ ，累积奖励为  $G'_a$ ；看到游戏画面  $s_b$ ，累积奖励为  $G'_b$ 。如果使用蒙特卡洛 (Monte Carlo, MC) 的方法，如图 7.24 所示，输入  $s_a$  给价值函数  $V_{\pi_\theta}$ ，其输出  $V_{\pi_\theta}(s_a)$  跟  $G'_a$  越接近越好。输入  $s_b$  给价值函数  $V_{\pi_\theta}$ ，其输出  $V_{\pi_\theta}(s_b)$  跟  $G'_b$  越接近越好。

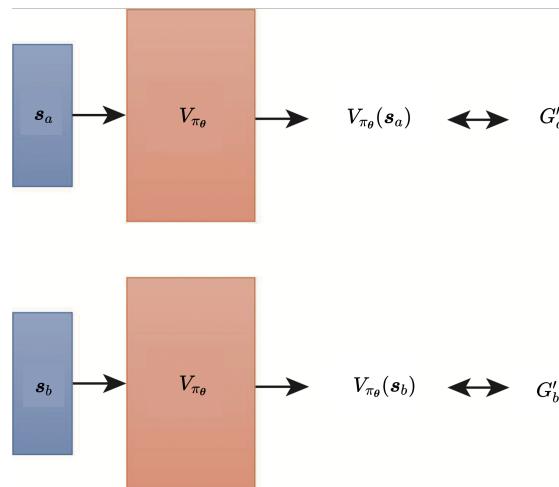


图 7.24 基于蒙特卡洛的方法

时序差分(Temporal-Difference, TD)方法不用玩完整场游戏，只要看到数据  $\{s_t, a_t, r_t, s_{t+1}\}$ ，就能够训练  $V_{\pi_\theta}(s)$ ，就可以更新  $V_{\pi_\theta}(s)$  的参数。蒙特卡洛需要玩完整场游戏，才能得到一笔训练数据。但有的游戏其实很长，甚至有的游戏不会结束，这种游戏不适合用蒙特卡洛方法。

而在时序差分方法中， $V_{\pi_\theta}(s_t)$  跟  $V_{\pi_\theta}(s_{t+1})$  之间的关系如式 (7.10) 所示（为了简化，没有取期望值）。

$$\begin{aligned} V_{\pi_\theta}(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ V_{\pi_\theta}(s_{t+1}) &= r_{t+1} + \gamma r_{t+2} + \dots \\ V_{\pi_\theta}(s_t) &= \gamma V_{\pi_\theta}(s_{t+1}) + r_t \end{aligned} \tag{7.10}$$

假设有一笔数据为  $\{s_t, a_t, r_t, s_{t+1}\}$ ， $s_t$  代到价值函数里面得到  $V_{\pi_\theta}(s_t)$ ， $s_{t+1}$  代到价值函数得到  $V_{\pi_\theta}(s_{t+1})$ 。虽然  $V_{\pi_\theta}(s_t)$  和  $V_{\pi_\theta}(s_{t+1})$  具体的值是未知的，但把两者应该满足如下关系

$$V_{\pi_\theta}(s_t) - \gamma V_{\pi_\theta}(s_{t+1}) \leftrightarrow r_t \tag{7.11}$$

$V_{\pi_\theta}(s_t) - \gamma V_{\pi_\theta}(s_{t+1})$  与  $r_t$  越接近越好。

同样的  $\pi_\theta$  得到的训练数据，用蒙特卡洛跟时序差分计算出的价值很可能是不一样的。

图 7.25 是某个智能体跟环境互动，玩了某个游戏八次的记录。

### Critic 观察了以下 8 个回合

- $s_a, r=0, s_b, r=0$ , 结束
- $s_b, r=1$ , 结束
- $s_b, r=0$ , 结束

图 7.25 时序差分方法与蒙特卡洛方法的差别<sup>[1]</sup>

为了简化计算，假设这些游戏都非常简单，一到两个回合就结束了。比如智能体第一次玩游戏的时候，它先看到画面  $s_a$ ，得到奖励 0，接下来看到画面  $s_b$ ，得到奖励 0，游戏结束。接下来有连续六场游戏，都是看到画面  $s_b$ ，得到奖励 1 就结束了。最后一场游戏，看到画面  $s_b$ ，得到奖励 0 就结束了。

Q: 如果  $s_a$  后面接的不一定是  $s_b$ , 该如何处理?

A: 如果  $s_a$  后面不一定接  $s_b$ , 这个问题在图 7.25 中的例子里面是无法处理的。因为在图 7.25 中,  $s_a$  后面只会接  $s_b$ , 没有观察到其它的可能性, 所以无法处理这个问题。在做强化学习的时候, 采样是非常重要的, 强化学习最后学习得好不好, 跟采样的好坏关系非常大。

为了简化起见, 先忽略动作, 并假设  $\gamma = 1$ , 即不做折扣。 $V_{\pi_\theta}(s_b)$  是指看到画面  $s_b$  得到的奖励的期望值。 $s_b$  画面在这八次游戏中, 总共看到了八次, 每次游戏都有看到  $s_b$  这个画面, 八次游戏里面, 有六次得到 1 分, 两次得到 0 分, 所以平均分为

$$\frac{6 \times 1 + 2 \times 0}{8} = \frac{6}{8} = \frac{3}{4} \quad (7.12)$$

$V_{\pi_\theta}(s_a)$  可以是 0 或  $\frac{3}{4}$ 。如果用蒙特卡洛计算, 因为看到  $s_a$  只有一次, 看到  $s_a$  的得到奖励 0,, 再看到  $s_b$  得到奖励还是 0, 所以累积奖励就是 0, 所以  $V_{\pi_\theta}(s_a) = 0$ 。但如果用时序差分算出来的, 因为  $V_{\pi_\theta}(s_a)$  跟  $V_{\pi_\theta}(s_b)$  中间有关系

$$V_{\pi_\theta}(s_a) = V_{\pi_\theta}(s_b) + r \quad (7.13)$$

因此

$$\begin{aligned} V_{\pi_\theta}(s_a) &= V_{\pi_\theta}(s_b) + r \\ &= \frac{3}{4} + 0 = \frac{3}{4} \end{aligned} \quad (7.14)$$

蒙特卡洛跟时序差分得出的结果都是对的, 它们只是背后的假设是不同的。对蒙特卡洛而言, 它就是直接看我们观察到的数据,  $s_a$  之后接  $s_b$  得到的, 累积奖励就是 0, 所以  $V_{\pi_\theta}(s_a)$  是 0。但对于时序差分而言, 它背后的假设是  $s_a$  跟  $s_b$  是没有关系的, 看到  $s_a$  之后再看到  $s_b$ , 并不会影响看到  $s_b$  的奖励。看到  $s_b$  以后得到的期望奖励应该是  $\frac{3}{4}$ , 所以看到  $s_a$  后看到  $s_b$ , 得到的期望奖励也应该是  $\frac{3}{4}$ , 所以从时序差分的角度来看,  $s_b$  会得到多少奖励跟  $s_a$  是没有关系的, 所以  $s_a$  的累积奖励应该是  $\frac{3}{4}$ 。

接下来介绍下如何用 Critic 训练 Actor。智能体跟环境互动得到一堆如图 7.26 所示的状态-动作对。比如  $s_1$  执行  $a_1$  得到一个分数  $A_1$ , 可令  $A_1 = G'_1 - b$ 。

学习出 Critic  $V_{\pi_\theta}$  后, 给定一个状态  $s$ , 其可以产生分数  $V_{\pi_\theta}(s)$ , 基线  $b$  可设成  $V_{\pi_\theta}(s)$ , 因此  $A$  可设成  $G' - V_{\pi_\theta}(s)$ , 如图 7.27 所示。

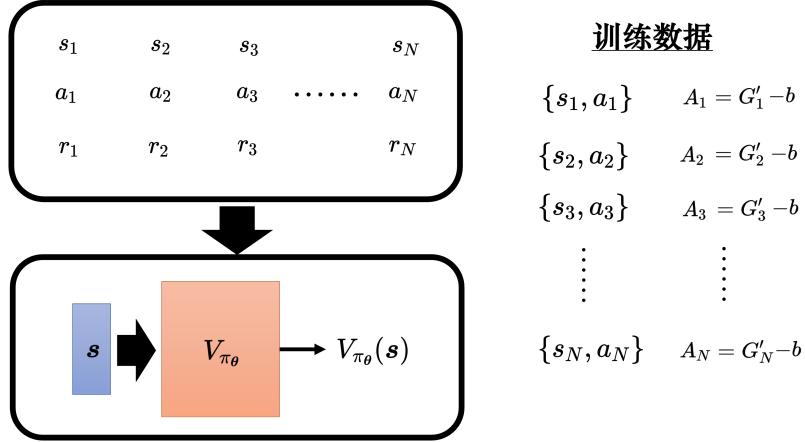


图 7.26 使用折扣累积奖励减去基线作为评价标准

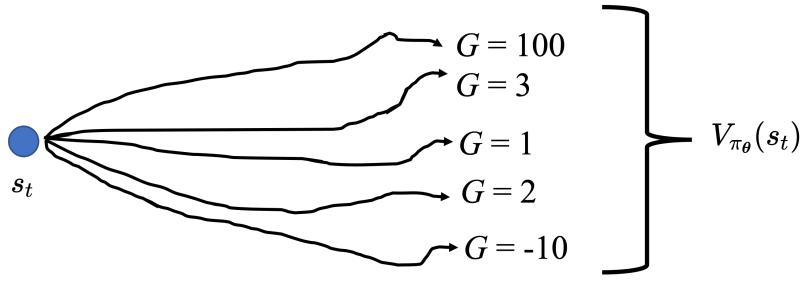
### 训练数据

$$\begin{aligned}
 \{s_1, a_1\} & \quad A_1 = G'_1 - V_{\pi_\theta}(s_1) \\
 \{s_2, a_2\} & \quad A_2 = G'_2 - V_{\pi_\theta}(s_2) \\
 \{s_3, a_3\} & \quad A_3 = G'_3 - V_{\pi_\theta}(s_3) \\
 & \vdots \qquad \vdots \\
 \{s_N, a_N\} & \quad A_N = G'_N - V_{\pi_\theta}(s_N)
 \end{aligned}$$

图 7.27 使用  $V_{\pi_\theta}(s)$  作为基线

$A_t$  代表  $\{s_t, a_t\}$  的好坏，智能体是指看到某一个画面  $s_t$  以后，接下来再继续玩游戏，游戏有随机性，每次得到的奖励都不太一样， $V_{\pi_\theta}(s_t)$  是一个期望值。此外，智能体在看到  $s_t$  的时候，智能体不一定会执行  $a_t$ 。因为智能体本身是有随机性的，在训练的过程中，同样的状态，智能体会输出的动作不一定是一样的。智能体的输出是动作的空间上的概率分布，给每一个动作一个分数，按照这个分数去做采样。有些动作被采样到的概率高，有些动作被采样到的概率低，但每一次采样出来的动作，并不保证一定要是一样的，所以如图 7.28 所示，看到  $s_t$  之后，接下来有很多的可能，可以计算出不同的累积奖励（此处是无折扣的累积奖励）。

把这些可能的结果平均起来，就是  $V_{\pi_\theta}(s_t)$ 。 $G'_t$  这一项的含义是，在  $s_t$  这个画面下，执行  $a_t$  以后，接下来会得到的累积奖励。如果  $A_t > 0$ ，代表  $G'_t > V_{\pi_\theta}(s_t)$ ，则动作  $a_t$  是比随机采样到的动作还要好，因此给  $a_t$  的评价  $A_t > 0$ 。如果  $A_t < 0$ ，这代表随机采样到的动作的累积奖励的期望值大过执行  $a_t$  得到的奖励，因此  $a_t$  是个不太好的动作，其  $A_t < 0$ 。

图 7.28 看到  $s_t$  后不同的累积奖励

接下来我们还可以做一个改进。 $G'_t$  是一个采样的结果，它是执行  $a_t$  以后，一直玩到游戏结束，而  $V_{\pi_\theta}(s_t)$  是很多个可能性平均以后的结果。用一个采样减掉平均，其实不太准，这个采样可能特别好或特别坏。所以其实可以用平均去减掉平均，也就是版本 4。

#### 7.4.6 版本 4

执行完  $a_t$  以后得到奖励  $r_t$ ，然后跑到下一个画面  $s_{t+1}$ 。把  $s_{t+1}$  接下来一直玩下去，有很多不同的可能，每个可能通通会得到一个奖励，把这些累积奖励平均就是  $V_{\pi_\theta}(s_{t+1})$ 。需要玩很多场游戏，才能够得到这个平均值。但可以训练出一个好的 Critic，直接代  $V_{\pi_\theta}(s_{t+1})$ ，在  $s_{t+1}$  这个画面下，接下来会得到的，累积奖励的期望值。在  $s_t$  这边采取  $a_t$  会得到奖励  $r_t$ ，再跳到  $s_{t+1}$ ， $s_{t+1}$  会得到期望的奖励为  $V_{\pi_\theta}(s_{t+1})$ 。所以  $r_t + V_{\pi_\theta}(s_{t+1})$  代表在  $s_t$  这边执行  $a_t$  会得到的奖励的期望值。因此可把  $G'_t$  换成  $r_t + V_{\pi_\theta}(s_{t+1})$ ，如图 7.29 所示。用  $r_t + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t)$ ，即采取动作  $a_t$  得到的期望奖励减掉根据某个分布采样动作得到的期望奖励。如果  $r_t + V_{\pi_\theta}(s_{t+1}) > V_{\pi_\theta}(s_t)$ ，代表  $a_t$  比从一个分布随便采样的动作好。反之，则代表  $a_t$  比从一个分布随便采样的动作差。这就是一个常用的方法，称为优势 Actor-Critic (advantage Actor-Critic)。在优势 Actor-Critic 里面， $A_t$  就是  $r_t + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t)$ 。

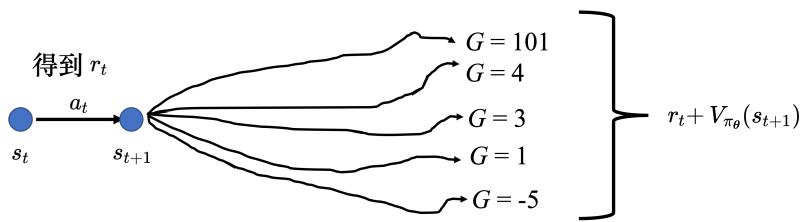


图 7.29 优势 Actor-Critic

Actor-Critic 有一个训练的技巧。Actor 和 Critic 都是一个网络，Actor 网络的输入是一

个游戏画面，其输出是每一个动作的分数。Critic 的输入是游戏画面，输出是一个数值，代表接下来会得到的累积奖励。图 7.30 中有两个网络，它们的输入是一样的东西，所以这两个网络应该有部分的参数可以共用，尤其假设输入又是一个非常复杂的东西，比如说游戏画面，前面几层应该都需要是卷积神经网络。所以 Actor 和 Critic 可以共用前面几个层，所以在实践的时候往往会这样设计 Actor-Critic。

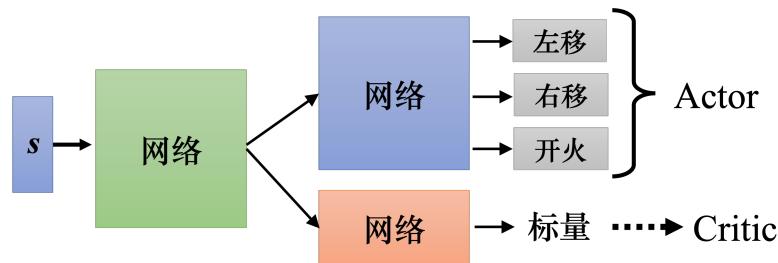


图 7.30 Actor-Critic 训练技巧

强化学习还可以直接用 Critic 决定要执行的动作，比如深度 Q 网络（Deep Q-Network, DQN）。DQN 有非常多的变形，有一篇非常知名的论文“Rainbow: Combining Improvements in Deep Reinforcement Learning”<sup>[2]</sup>，把 DQN 的七种变形集合起来，因为有七种变形集合起来，所以这个方法称为彩虹（rainbow）。

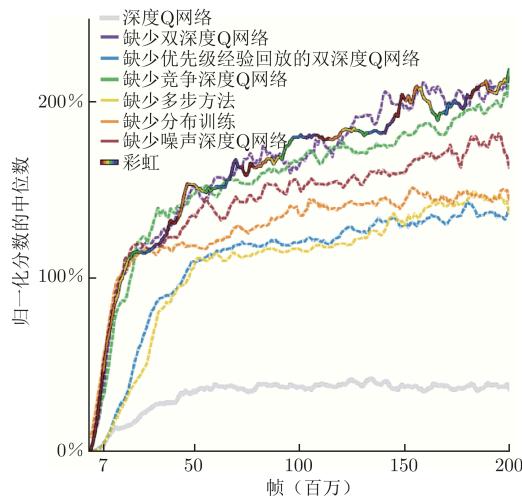


图 7.31 彩虹方法

强化学习里面还有很多技巧，比如稀疏奖励的处理方法以及模仿学习，详细内容可参考《Easy RL：强化学习教程》<sup>[3]</sup>，此处不再赘述。

## 参考文献

- [1] SUTTON R S, BARTO A G. Reinforcement learning: An introduction(second edition)[M]. London:The MIT Press, 2018.
- [2] HESSEL M, MODAYIL J, VAN HASSELT H, et al. Rainbow: Combining improvements in deep reinforcement learning[C]//Thirty-second AAAI conference on artificial intelligence. 2018.
- [3] 王琦, 杨毅远, 江季. Easy RL: 强化学习教程[M]. 北京: 人民邮电出版社, 2022.