李宏毅深度学习教程
LeeDL Tutorial

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

王琦　杨毅远　江季　编著

版本：1.1.9

2024 年 8 月 9 日

前言

李宏毅老师是台湾大学的教授，其《机器学习》（2021 年春）是深度学习领域经典的中文视频之一。李老师幽默风趣的授课风格深受大家喜爱，让晦涩难懂的深度学习理论变得轻松易懂，他会通过很多动漫相关的有趣例子来讲解深度学习理论。李老师的课程内容很全面，覆盖了到深度学习必须掌握的常见理论，能让学生对于深度学习的绝大多数领域都有一定了解，从而可以进一步选择想要深入的方向进行学习，对于想入门深度学习又想看中文讲解的同学是非常推荐的。

本教程主要内容源于《机器学习》（2021 年春），并在其基础上进行了一定的原创。比如，为了尽可能地降低阅读门槛，笔者对这门公开课的精华内容进行选取并优化，对所涉及的公式都给出详细的推导过程，对较难理解的知识点进行了重点讲解和强化，以方便读者较为轻松地入门。此外，为了丰富内容，笔者在教程中选取了《机器学习》（2017 年春） 的部分内容，并补充了不少除这门公开课之外的深度学习相关知识。

(a) 京东购买　　(b) 当当购买

纸质书购买

致谢

特别感谢 Sm1les、LSGOMYP、FuWeiru对本项目的帮助与支持。

扫描下方二维码，然后回复关键词“**李宏毅深度学习**”，即可加入“LeeDL-Tutorial 读者交流群”

Datawhale

一个专注于 AI 领域的开源组织

作者简介

王琦 上海交通大学人工智能教育部重点实验室博士研究生，硕士毕业于中国科学院大学。Datawhale 成员，《Easy RL：强化学习教程》作者，英特尔边缘计算创新大使，Hugging Face 社区志愿者，Al TIME 成员。主要研究方向为强化学习、计算机视觉、深度学习。曾获中国光谷 · 华为杯第十九届中国研究生数学建模竞赛二等奖、中国大学生计算机设计大赛二等奖、亚太地区大学生数学建模竞赛（APMCM）二等奖，发表 SCl/EI 论文多篇，个人主页：`https://qiwang067.github.io/`。

杨毅远 牛津大学计算机系博士研究生，硕士毕业于清华大学。Datawhale 成员，《Easy RL：强化学习教程》作者。主要研究方向为时间序列、数据挖掘、智能传感系统，深度学习。曾获国家奖学金、北京市优秀毕业生、清华大学优秀学位论文、全国大学生智能汽车竞赛总冠军等荣誉，发表 SCI/EI 论文多篇，个人主页：`https://yyysjz1997.github.io/`。

江季 网易高级算法工程师，硕士毕业于北京大学。Datawhale 成员，《Easy RL：强化学习教程》作者。主要研究方向为强化学习、深度学习、大模型、机器人等。曾获得国家奖学金、上海市优秀毕业生等荣誉，发表强化学习与游戏 AI 等相关专利多项，个人主页：`https://github.com/JohnJim0816`。

主要符号表

a	标量
$\boldsymbol{a}$	向量
$\boldsymbol{A}$	矩阵
$\boldsymbol{I}$	单位矩阵
$\mathbb{R}$	实数集
$\boldsymbol{A}^{\mathrm{T}}$	矩阵 $\boldsymbol{A}$ 的转置
$\boldsymbol{A} \odot \boldsymbol{B}$	$\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的按元素乘积
$\frac{dy}{dx}$	y 关于 x 的导数
$\frac{\partial y}{\partial x}$	y 关于 x 的偏导数
$\nabla_{\boldsymbol{x}} y$	y 关于 $\boldsymbol{x}$ 的梯度
$a \sim p$	具有分布 p 的随机变量 a
$\mathbb{E}[f(x)]$	$f(x)$ 的期望
$\mathrm{Var}(f(x))$	$f(x)$ 的方差
$\exp(x)$	x 的指数函数
$\log x$	x 的对数函数
$\sigma(x)$	Sigmoid 函数，$\frac{1}{1+\exp(-x)}$
s	状态
a	动作
r	奖励
π	策略
γ	折扣因子
τ	轨迹
G_t	时刻 t 时的回报
$\arg\min_a f(a)$	$f(a)$ 取最小值时 a 的值

目录

第 1 章 机器学习基础

首先简单介绍一下**机器学习 (Machine Learning, ML)** 和**深度学习 (Deep Learning, DL)** 的基本概念。机器学习，顾名思义，机器具备有学习的能力。具体来讲，机器学习就是让机器具备找一个函数的能力。机器具备找函数的能力以后，它可以做很多事。比如语音识别，机器听一段声音，产生这段声音对应的文字。我们需要的是一个函数，该函数的输入是声音信号，输出是这段声音信号的内容。这个函数显然非常复杂，人类难以把它写出来，因此想通过机器的力量把这个函数自动找出来。还有好多的任务需要找一个很复杂的函数，以图像识别为例，图像识别函数的输入是一张图片，输出是这个图片里面的内容。AlphaGo 也可以看作是一个函数，机器下围棋需要的就是一个函数，该函数的输入是棋盘上黑子跟白子的位置，输出是机器下一步应该落子的位置。

随着要找的函数不同，机器学习有不同的类别。假设要找的函数的输出是一个数值，一个标量（scalar），这种机器学习的任务称为回归。举个回归的例子，假设机器要预测未来某一个时间的 PM2.5 的数值。机器要找一个函数 f，其输入是可能是种种跟预测 PM2.5 有关的指数，包括今天的 PM2.5 的数值、平均温度、平均的臭氧浓度等等，输出是明天中午的 PM2.5 的数值，找这个函数的任务称为**回归 (regression)**。

除了回归以外，另一个常见的任务是**分类 (classification，)**。分类任务要让机器做选择题。人类先准备好一些选项，这些选项称为类别（class），现在要找的函数的输出就是从设定好的选项里面选择一个当作输出，该任务称为分类。举个例子，每个人都有邮箱账户，邮箱账户里面有一个函数，该函数可以检测一封邮件是否为垃圾邮件。分类不一定只有两个选项，也可以有多个选项。

AlphaGo 也是一个分类的问题，如果让机器下围棋，做一个 AlphaGo，给出的选项与棋盘的位置有关。棋盘上有 19×19 个位置，机器下围棋其实是一个有 19×19 个选项的选择题。机器找一个函数，该函数的输入是棋盘上黑子跟白子的位置，输出就是从 19×19 个选项里面，选出一个正确的选项，从 19×19 个可以落子的位置里面，选出下一步应该要落子的位置。

在机器学习领域里面，除了回归跟分类以外，还有结构化学习（structured learning）。机器不只是要做选择题或输出一个数字，而是产生一个有结构的物体，比如让机器画一张图，写一篇文章。这种叫机器产生有结构的东西的问题称为结构化学习。

1.1 案例学习

以视频的点击次数预测为例介绍下机器学习的运作过程。假设有人想要通过视频平台赚钱，他会在意频道有没有流量，这样他才会知道他的获利。假设后台可以看到很多相关的信息，比如：每天点赞的人数、订阅人数、观看次数。根据一个频道过往所有的信息可以预测明天的观看次数。找一个函数，该函数的输入是后台的信息，输出是隔天这个频道会有的总观看的次数.

机器学习找函数的过程，分成 3 个步骤。第一个步骤是写出一个带有未知参数的函数 f，其能预测未来观看次数。比如将函数写成

$$y = b + wx_1 \tag{1.1}$$

其中，y 是准备要预测的东西，要预测的是今天（2 月 26 日）这个频道总共观看的人，y 就假设是今天总共的观看次数。x_1 是这个频道，前一天（2 月 25 日）总共的观看次数，y 跟 x_1 都

是数值，b 跟 w 是未知的参数，它是准备要通过数据去找出来的，w 跟 b 是未知的，只是隐约地猜测。猜测往往来自于对这个问题本质上的了解，即领域知识（domain knowledge）。机器学习就需要一些领域知识。这是一个猜测，也许今天的观看次数，总是会跟昨天的观看次数有点关联，所以把昨天的观看次数，乘上一个数值，但是总是不会一模一样，所以再加上一个 b 做修正，当作是对于 2 月 26 日，观看次数的预测，这是一个猜测，它不一定是对的，等一下回头会再来修正这个猜测。总之，$y = b + w * x_1$，而 b 跟 w 是未知的。带有未知的**参数（parameter）**的函数称为**模型（model）**。模型在机器学习里面，就是一个带有未知的参数的函数，**特征（feature）** x_1 是这个函数里面已知的，它是来自于后台的信息，2 月 25 日点击的总次数是已知的，而 w 跟 b 是未知的参数。w 称为**权重（weight）**，b 称为**偏置（bias）**。这个是第一个步骤。

第 2 个步骤是定义损失（loss），损失也是一个函数。这个函数的输入是模型里面的参数，模型是 $y = b + w * x_1$，而 b 跟 w 是未知的，损失是函数 $L(b, w)$，其输入是模型参数 b 跟 w。损失函数输出的值代表，现在如果把这一组未知的参数，设定某一个数值的时候，这笔数值好还是不好。举一个具体的例子，假设未知的参数的设定是 $b = 500$，$w = 1$，预测未来的观看次数的函数就变成 $y = 500 + x_1$。要从训练数据来进行计算损失，在这个问题里面，训练数据是这一个频道过去的观看次数。举个例子，从 2017 年 1 月 1 日到 2020 年 12 月 31 日的观看次数（此处的数字是随意生成的）如图 1.1 所示，接下来就可以计算损失。

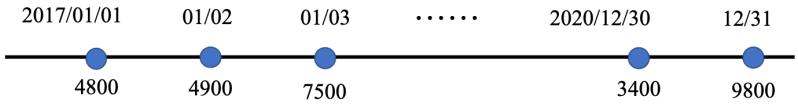


图 1.1　2017 年 1 月 1 日到 2020 年 12 月 31 日的观看次数

把 2017 年 1 月 1 日的观看次数，代入这一个函数里面

$$\hat{y} = 500 + 1x_1 \tag{1.2}$$

可以判断 $b = 500$，$w = 1$ 的时候，这个函数有多棒。x_1 代入 4800，预测隔天实际上的观看次数结果为 $\hat{y} = 5300$，真正的结果是 4900，真实的值称为标签（label），它高估了这个频道可能的点击次数，可以计算一下估测的值 $\hat{y}$ 跟真实值 y 的差距 e。计算差距其实不只一种方式，比如取绝对值：

$$e_1 = |y - \hat{y}| = 400 \tag{1.3}$$

我们不是只能用 1 月 1 日，来预测 1 月 2 日的值，可以用 1 月 2 日的值，来预测 1 月 3 日的值。根据 1 月 2 日的观看次数，预测的 1 月 3 日的观看次数的，值是 5400。接下来计算 5400 跟跟标签（7500）之间的差距，低估了这个频道。在 1 月 3 日的时候的观看次数，才可以算出：

$$e_2 = |y - \hat{y}| = 2100 \tag{1.4}$$

我们可以算过这 3 年来，每一天的预测的误差，这 3 年来每一天的误差，通通都可以算出来，每一天的误差都可以得到 e。接下来把每一天的误差，通通加起来取得平均，得到损失 L

$$L = \frac{1}{N} \sum_n e_n \tag{1.5}$$

其中，N 代表训验数据的个数，即 3 年来的训练数据，就 365 乘以 3，计算出一个 L，，L 是每一笔训练数据的误差 e 相加以后的结果。L 越大，代表现在这一组参数越不好，L 越小，代表现在这一组参数越好。

估测的值跟实际的值之间的差距，其实有不同的计算方法，计算 y 与 $\hat{y}$ 之间绝对值的差距，如式 (1.6) 所示，称为**平均绝对误差（Mean Absolute Error，MAE）**。

$$e = |\hat{y} - y| \tag{1.6}$$

如果算 y 与 $\hat{y}$ 之间平方的差距，如式 (1.7) 所示，则称为**均方误差（Mean Squared Error，MSE）**。

$$e = (\hat{y} - y)^2 \tag{1.7}$$

有一些任务中 y 和 $\hat{y}$ 都是概率分布，这个时候可能会选择**交叉熵（cross entropy）**，这个是机器学习的第 2 步。刚才举的那些数字不是真正的例子，以下的数字是真实的例子，是这个频道真实的后台的数据，所计算出来的结果。可以调整不同的 w 和不同的 b，求取各种 w 和各种 b，组合起来以后，我们可以为不同的 w 跟 b 的组合，都去计算它的损失，就可以画出图 1.2 所示的等高线图。在这个等高线图上面，越偏红色系，代表计算出来的损失越大，就代表这一组 w 跟 b 越差。如果越偏蓝色系，就代表损失越小，就代表这一组 w 跟 b 越好，拿这一组 w 跟 b，放到函数里面，预测会越精准。假设 $w = -0.25, b = -500$，这代表这个频道每天看的人越来越少，而且损失这么大，跟真实的情况不太合。如果 $w = 0.75, b = 500$，估测会比较精准。如果 w 代一个很接近 1 的值，b 带一个小小的值，比如说 100 多，这个时候估测是最精准的，这跟大家的预期可能是比较接近的，就是拿前一天的点击的总次数，去预测隔天的点击的总次数，可能前一天跟隔天的点击的总次数是差不多的，因此 w 设 1，b 设一个小一点的数值，也许估测就会蛮精准的。如图 1.2 所示的等高线图，就是试了不同的参数，计算它的损失，画出来的等高线图称为**误差表面（error surface）**。这是机器学习的第 2 步。

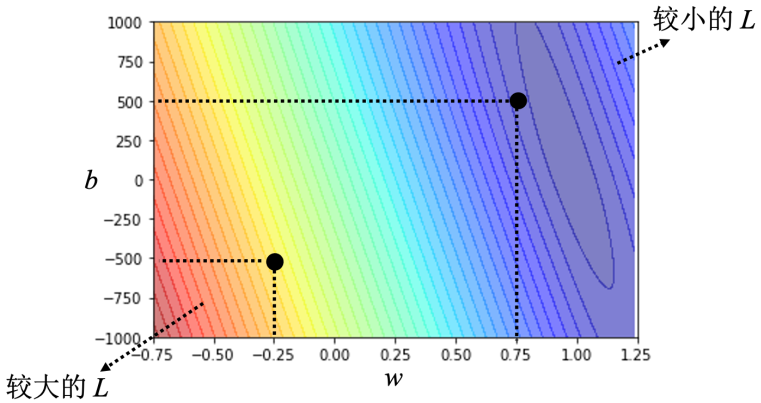


图 1.2　误差表面

接下来进入机器学习的第 3 步：解一个最优化的问题。找一个 w 跟 b，把未知的参数找一个数值出来，看代哪一个数值进去可以让损失 L 的值最小，就是要找的 w 跟 b，这个可以让损失最小的 w 跟 b 称为 w^* 跟 b^* 代表它们是最好的一组 w 跟 b，可以让损失的值最小。**梯度下降（gradient descent）**是经常会使用优化的方法。为了要简化起见，先假设只有一个未知的参数 w，b 是已知的。w 代不同的数值的时候，就会得到不同的损失，这一条曲线就是误

差表面，只是刚才在前一个例子里面，误差表面是 2 维的，这边只有一个参数，所以这个误差表面是 1 维的。怎么样找一个 w 让损失的值最小呢？如图 1.3 所示，首先要随机选取一个初始的点 w^0。接下来计算 $\frac{\partial L}{\partial w}|_{w=w^0}$，在 w 等于 w^0 的时候，参数 w 对损失的微分。计算在这一个点，在 w^0 这个位置的误差表面的切线斜率，也就是这一条蓝色的虚线，它的斜率，如果这一条虚线的斜率是负的，代表说左边比较高，右边比较低。在这个位置附近，左边比较高，右边比较低。如果左边比较高右边比较低的话，就把 w 的值变大，就可以让损失变小。如果算出来的斜率是正的，就代表左边比较低右边比较高。左边比较低右边比较高，如果左边比较低右边比较高的话，就代表把 w 变小了，w 往左边移，可以让损失的值变小。这个时候就应该把 w 的值变小。我们可以想像说有一个人站在这个地方，他左右环视一下，算微分就是左右环视，它会知道左边比较高还是右边比较高，看哪边比较低，它就往比较低的地方跨出一步。这一步的步伐的大小取决于两件事情：

- 第一件事情是这个地方的斜率，斜率大步伐就跨大一点，斜率小步伐就跨小一点。
- 另外，**学习率（learning rate）** η 也会影响步伐大小。学习率是自己设定的，如果 η 设大一点，每次参数更新就会量大，学习可能就比较快。如果 η 设小一点，参数更新就很慢，每次只会改变一点点参数的数值。这种在做机器学习，需要自己设定，不是机器自己找出来的，称为**超参数（hyperparameter）**。

> Q: 为什么损失可以是负的?
> A: 损失函数是自己定义的，在刚才定义里面，损失就是估测的值跟正确的值的绝对值。如果根据刚才损失的定义，它不可能是负的。但是损失函数是自己决定的，比如设置一个损失函数为绝对值再减 100，其可能就有负的。这个曲线并不是一个真实的损失，并不是一个真实任务的误差表面。因此这个损失的曲线可以是任何形状。

把 w^0 往右移一步，新的位置为 w^1，这一步的步伐是 η 乘上微分的结果，即：

$$w^1 \leftarrow w^0 - \eta \left.\frac{\partial L}{\partial w}\right|_{w=w^0} \tag{1.8}$$

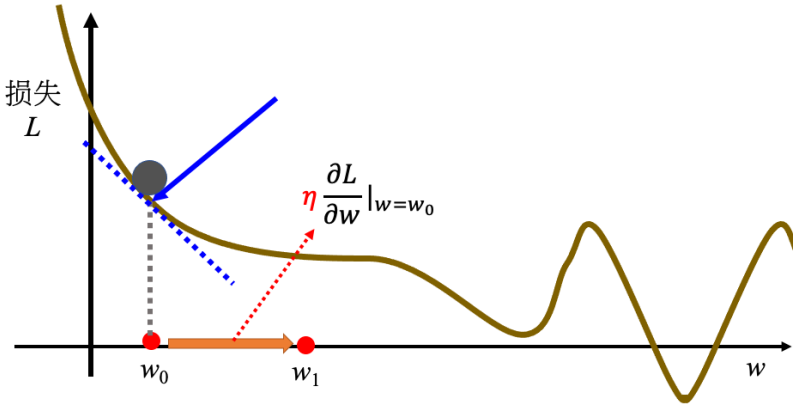


图 1.3　优化过程

接下来反复进行刚才的操作，计算一下 w^1 微分的结果，再决定现在要把 w^1 移动多少，再移动到 w^2，再继续反复做同样的操作，不断地移动 w 的位置，最后会停下来。往往有两种情况会停下来。

- 第一种情况是一开始会设定说，在调整参数的时候，在计算微分的时候，最多计算几次。上限可能会设为 100 万次，参数更新 100 万次后，就不再更新了，更新次数也是一个超参数。
- 还有另外一种理想上的，停下来的可能是，当不断调整参数，调整到一个地方，它的微分的值就是这一项，算出来正好是 0 的时候，如果这一项正好算出来是 0，0 乘上学习率 η 还是 0，所以参数就不会再移动位置。假设是这个理想的情况，把 w^0 更新到 w^1，再更新到 w^2，最后更新到 w^T 有点卡，w^T 卡住了，也就是算出来这个微分的值是 0 了，参数的位置就不会再更新。

梯度下降有一个很大的问题，没有找到真正最好的解，没有找到可以让损失最小的 w。在图 1.4 所示的例子里面，把 w 设定在最右侧红点附近这个地方可以让损失最小。但如果在梯度下降中，w^0 是随机初始的位置，也很有可能走到 w^T 这里，训练就停住了，无法再移动 w 的位置。右侧红点这个位置是真的可以让损失最小的地方，称为**全局最小值（global minima）**，而 w^T 这个地方称为**局部最小值（local minima）**，其左右两边都比这个地方的损失还要高一点，但是它不是整个误差表面上面的最低点。

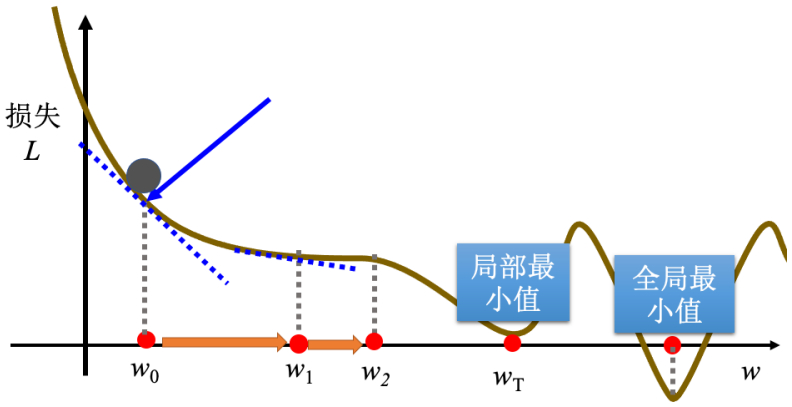


图 1.4　局部最小值

所以常常可能会听到有人讲到梯度下降不是个好方法，这个方法会有局部最小值的问题，无法真的找到全局最小值。事实上局部最小值是一个假问题，在做梯度下降的时候，真正面对的难题不是局部最小值。有两个参数的情况下使用梯度下降，其实跟刚才一个参数没有什么不同。如果一个参数没有问题的话，可以很快的推广到两个参数。

假设有两个参数，随机初始值为 w^0, b^0。要计算 w, b 跟损失的微分，计算在 $w = w^0$ 的位置，$b = b^0$ 的位置，要计算 w 对 L 的微分，计算 b 对 L 的微分

$$\left.\frac{\partial L}{\partial b}\right|_{w=w^0,b=b^0} \\ \left.\frac{\partial L}{\partial w}\right|_{w=w^0,b=b^0} \tag{1.9}$$

计算完后更新 w 跟 b，把 w^0 减掉学习率乘上微分的结果得到 w^1，把 b^0 减掉学习率乘

上微分的结果得到 b^1。

$$w^1 \leftarrow w^0 - \eta \left.\frac{\partial L}{\partial w}\right|_{w=w^0,b=b^0}$$
$$b^1 \leftarrow b^0 - \eta \left.\frac{\partial L}{\partial b}\right|_{w=w^0,b=b^0} \tag{1.10}$$

在深度学习框架里面，比如 PyTorch 里面，算微分都是程序自动帮计算的。就是反复同样的步骤，就不断的更新 w 跟 b，期待最后，可以找到一个最好的 w，w^* 跟最好的 b^*. 如图 1.5 所示，随便选一个初始的值，先计算一下 w 对 L 的微分，跟计算一下 b 对 L 的微分，接下来更新 w 跟 b，更新的方向就是 $\partial L/\partial w$，乘以 η 再乘以一个负号，$\partial L/\partial b$，算出这个微分的值，就可以决定更新的方向，可以决定 w 要怎么更新。把 w 跟 b 更新的方向结合起来，就是一个向量，就是红色的箭头，再计算一次微分，再决定要走什么样的方向，把这个微分的值乘上学习率，再乘上负号，我们就知道红色的箭头要指向那里，就知道如何移动 w 跟 b 的位置，一直移动，期待最后可以找出一组不错的 w, b。实际上真的用梯度下降，进行一番计算以后，这个是真正的数据，算出来的最好的 $w^* = 0.97, b^* = 100$，跟猜测蛮接近的。因为 x_1 的值可能跟 y 很接近，所以这个 w 就设一个接近 1 的值，b 就设一个比较偏小的值。损失 $L(w^*, b^*)$ 算一下是 480，也就是在 2017 到 2020 年的数据上，如果使用这一个函数，b 代 100，w 代 0.97，平均的误差是 480，其预测的观看次数误差，大概是 500 人左右。

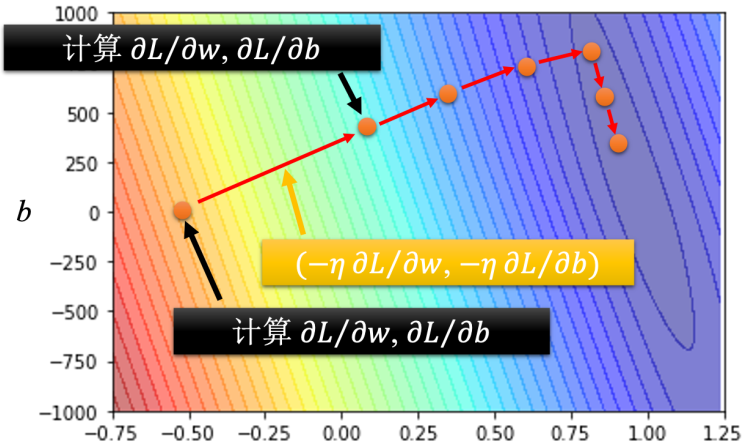


图 1.5　梯度下降优化的过程

1.2　线性模型

w 跟 b 的值刚才已经找出来的，这组 w 跟 b 可以让损失小到 480。在已经知道答案的数据上去计算损失，2017 到 2020 年每天的观看次数是已知的。所以假装不知道隔天的观看次数，拿这一个函数来进行预测，发现误差是 480。接下来使用这个函数预测未来的观看次数。预测从 2021 年开始每一天都拿这个函数去预测次日的观看人次：用 2020 年的 12 月 31 日的观看人次预测 2021 年 1 月 1 日的观看人次，用 2021 年 1 月 1 日的观看人次预测 1 月 2 日的观看人次，用 1 月 2 日的观看人次去预测 1 月 3 日的观看人次……每天都做这件事，一直做到 2 月 14 日，得到平均的值，在 2021 年没有看过的数据上，误差值用 L' 来表示，它是 0.58，所以在有看过的数据上，在训练数据上，误差值是比较小的，在没有看过的数据上，在

2021 年的数据上，看起来误差值是比较大的，每一天的平均误差有 580 人左右，600 人左右。如图 1.6 所示，横轴是代表的是时间，所以 0 这个点代表的是 2021 年 1 月 1 日，最右边点代表的是 2021 年 2 月 14 日，纵轴就是观看的人次，这边是用千人当作单位。红色线是真实的观看人次，蓝色线是机器用这一个函数预测出来的观看人次。蓝色的线几乎就是红色的线往右平移一天而已，这很合理，因为 x_1 也就是前一天的观看人次，跟隔天观看人次的，要怎么拿前一天的观看人次，去预测隔天的观看人次呢，前一天观看人次乘以 0.97，加上 100 加上 100，就是隔天的观看人次。机器几乎就是拿前一天的观看人次来预测隔天的观看人次。这个真实的数据有一个很神奇的现象，它是有周期性的，它每隔 7 天就会有两天特别低（周五和周六），两天观看的人特别少，每隔 7 天，就是一个循环。目前的模型不太行，它只能够看前一天。 每隔 7 天它一个循环，如果一个模型参考前 7 天的数据，把 7 天前的数据，直接复制到拿来当作预测的结果，也许预测的会更准也说不定，所以我们就要修改一下模型。通常一个模型的修改，往往来自于对这个问题的理解，即领域知识。

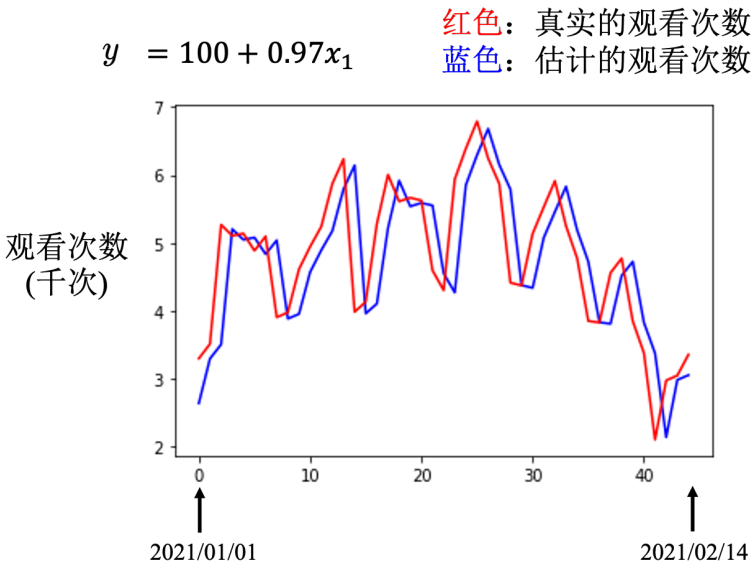


图 1.6　预估曲线图

一开始，对问题完全不理解的时候，胡乱写一个

$$y = b + wx_1 \tag{1.11}$$

并没有做得特别好。接下来我们观察了真实的数据以后，得到一个结论是，每隔 7 天有一个循环。所以要把前 7 天的观看人次都列入考虑，写了一个新的模型

$$y = b + \sum_{j=1}^{7} w_j x_j \tag{1.12}$$

其中 x_j 代表第 j 天的观看测试，也就是 7 天前的数据，通通乘上不同的权重 w_j，加起来，再加上偏置得到预测的结果。使用该模型预测，其在训练数据上的损失是 380，而只考虑 1 天的模型在训练数据上的损失是 480。因为其考虑了 7 天，所以在训练数据上会得到比较低的损失。考虑了比较多的信息，在训练数据上应该要得到更好的、更低的损失。在没有看到的数据上的损失有比较好是 490。只考虑 1 天的误差是 580，考虑 7 天的误差是 490。用梯度下降，算出 w 跟 b 的最优值如 表 1.1 所示。

表 1.1 w 和 b 的最优值

b	w_1^*	w_2^*	w_3^*	w_4^*	w_5^*	w_6^*	w_7^*
50	0.79	-0.31	0.12	-0.01	-0.10	0.30	0.18

机器的逻辑是前一天跟要预测的隔天的数值的关系很大，所以 w_1^* 是 0.79，不过它知道，如果是前两天前四天前五天，它的值会跟未来要预测的，隔天的值是成反比的，所以 w_2, w_4, w_5 最佳的值（让训练数据上的损失为 380 的值）是负的。但是 w_1, w_3, w_6, w_7 是正的，考虑前 7 天的值，其实可以考虑更多天，本来是考虑前 7 天，可以考虑 28 天，即

$$y = b + \sum_{j=1}^{28} w_j x_j. \tag{1.13}$$

28 天是一个月，考虑前一个月每一天的观看人次，去预测隔天的观看人次，训练数据上是 330。在 2021 年的数据上，损失是 460，看起来又更好一点。如果考虑 56 天，即

$$y = b + \sum_{j=1}^{56} w_j x_j \tag{1.14}$$

在训练数据上损失是 320，在没看过的数据上损失还是 460。考虑更多天没有办法再更降低损失了。看来考虑天数这件事，也许已经到了一个极限。这些模型都是把输入的特征 x 乘上一个权重，再加上一个偏置就得到预测的结果，这样的模型称为**线性模型 (linear model)**。接下来会看如何把线性模型做得更好。

1.2.1 分段线性曲线

线性模型也许过于简单，x_1 跟 y 可能中间有比较复杂的关系，如图 1.7 所示。对于线性模型，x_1 跟 y 的关系就是一条直线，随着 x_1 越来越高，y 就应该越来越大。设定不同的 w 可以改变这条线的斜率，设定不同的 b 可以改变这一条蓝色的直线跟 y 轴的交叉点。但是无论如何改 w 跟 b，它永远都是一条直线，永远都是 x_1 越大，y 就越大，前一天观看的次数越多，隔天的观看次数就越多。但现实中也许在 x_1 小于某一个数值的时候，前一天的观看次数跟隔天的观看次数是成正比；也许当 x_1 大于一个数值的时候，x_1 太大，前天观看的次数太高，隔天观看次数就会变少；也许 x_1 跟 y 中间，有一个比较复杂的、像红色线一样的关系。但不管如何设置 w 跟 b，永远制造不出红色线，永远无法用线性模型制造红色线。显然线性模型有很大的限制，这一种来自于模型的限制称为模型的偏差，无法模拟真实的情况。

所以需要写一个更复杂的、更有灵活性的、有未知参数的函数。红色的曲线可以看作是一个常数再加上一群 Hard Sigmoid 函数。Hard Sigmoid 函数的特性是当输入的值，当 x 轴的值小于某一个阈值（某个定值）的时候，大于另外一个定值阈值的时候，中间有一个斜坡。所以它是先水平的，再斜坡，再水平的。所以红色的线可以看作是一个常数项加一大堆的蓝色函数（Hard Sigmoid）。常数项设成红色的线跟 x 轴的交点一样大。常数项怎么加上蓝色函数后，变成红色的这一条线？蓝线 1 函数斜坡的起点，设在红色函数的起始的地方，第 2 个斜坡的终点设在第一个转角处，让第 1 个蓝色函数的斜坡和红色函数的斜坡的斜率是一样的，这个时候把 0+1 就可以得到红色曲线左侧的线段。接下来，再加第 2 个蓝色的函数，所以第 2 个蓝色函数的斜坡就在红色函数的第一个转折点到第 2 个转折点之间，让第 2 个蓝色函数

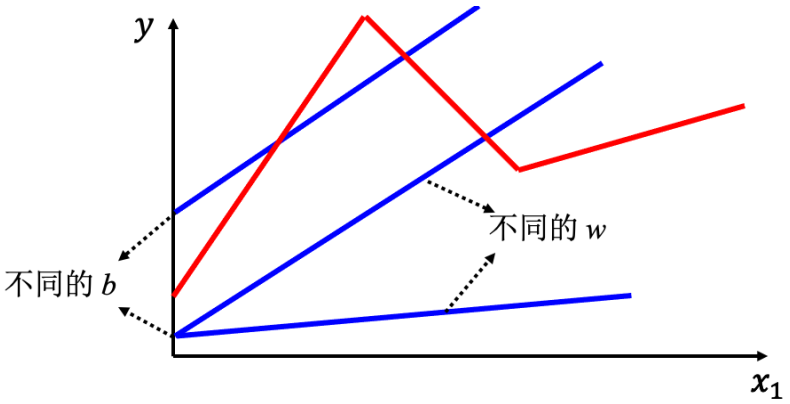
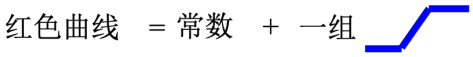


图 1.7　线性模型的局限性

的斜率跟红色函数的斜率一样，这个时候把 0+1+2，就可以得到红色函数左侧和中间的线段。接下来第 3 个部分，第 2 个转折点之后的部分，就加第 3 个蓝色的函数，第 3 个蓝色的函数坡度的起始点设的跟红色函数转折点一样，蓝色函数的斜率设的跟红色函数斜率一样，接下来把 0+1+2+3 全部加起来，就得到完整红色的线。

所以红色线，即分段线性曲线（piecewise linear curve）可以看作是一个常数，再加上一堆蓝色的函数。分段线性曲线可以用常数项加一大堆的蓝色函数组合出来，只是用的蓝色函数不一定一样。要有很多不同的蓝色函数，加上一个常数以后就可以组出这些分段线性曲线。如果分段线性曲线越复杂，转折的点越多，所需的蓝色函数就越多。



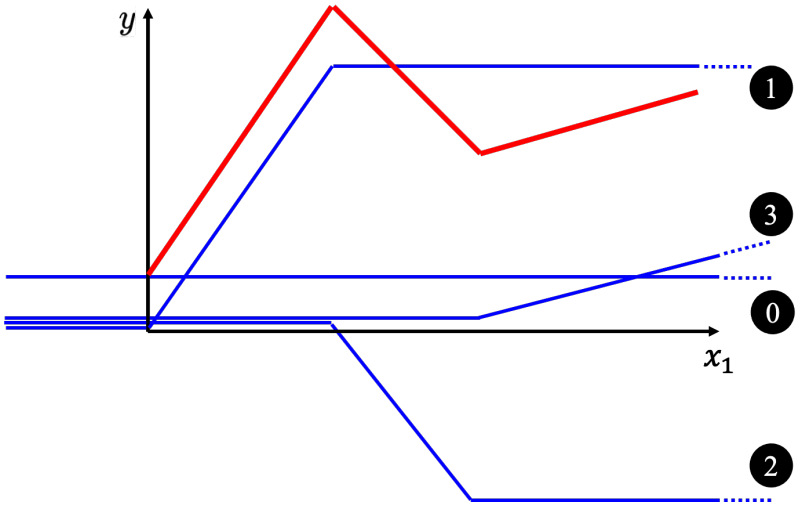


图 1.8　构建红色曲线

也许要考虑的 x 跟 y 的关系不是分段线性曲线，而是如图 1.9 所示的曲线。可以在这样的曲线上面，先取一些点，再把这些点点起来，变成一个分段线性曲线。而这个分段线性曲线跟原来的曲线，它会非常接近，如果点取的够多或点取的位置适当，分段线性曲线就可以逼近这一个连续的曲线，就可以逼近有角度的、有弧度的这一条曲线。 所以可以用分段线性曲线去逼近任何的连续的曲线，而每个分段线性曲线都可以用一大堆蓝色的函数组合起来。也就

是说，只要有足够的蓝色函数把它加起来，就可以变成任何连续的曲线。

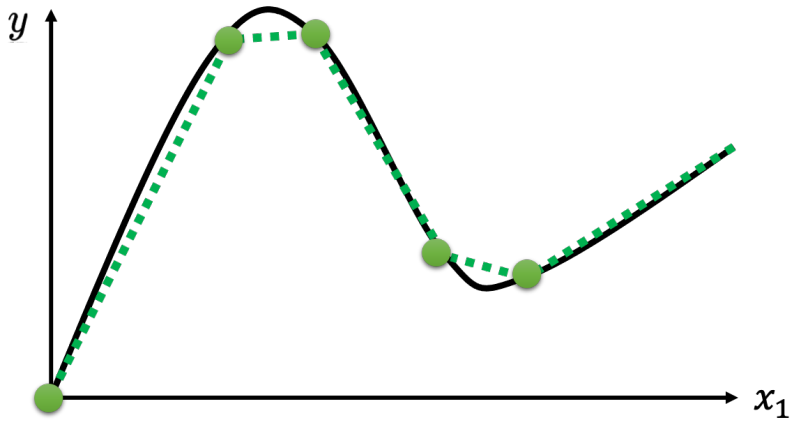


图 1.9　分段曲线可以逼近任何连续曲线

假设 x 跟 y 的关系非常复杂也没关系，就想办法写一个带有未知数的函数。直接写 Hard Sigmoid 不是很容易，但是可以用一条曲线来理解它，用 Sigmoid 函数来逼近 Hard Sigmoid，如图 1.10 所示。Sigmoid 函数的表达式为

$$y = c\frac{1}{1+e^{-(b+wx_1)}}$$

其横轴输入是 x_1，输出是 y，c 为常数。

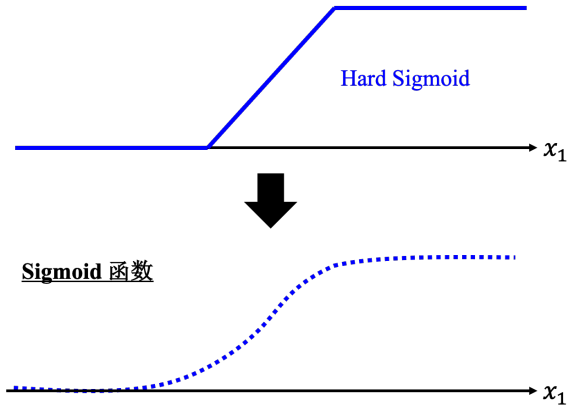


图 1.10　使用 Sigmoid 逼近 Hard Sigmoid

如果 x_1 的值，趋近于无穷大的时候，$e^{-(b+wx_1)}$ 这一项就会消失，当 x_1 非常大的时候，这一条就会收敛在高度为 c 的地方。如果 x_1 负的非常大的时候，分母的地方就会非常大，y 的值就会趋近于 0。

所以可以用这样子的一个函数逼近这一个蓝色的函数，即 Sigmoid 函数，Sigmoid 函数就是 S 型的函数。因为它长得是有点像是 S 型，所以叫它 Sigmoid 函数。

为了简洁，去掉了指数的部分，蓝色函数的表达式为

$$y = c\sigma(b+wx_1) \tag{1.15}$$

所以可以用 Sigmoid 函数逼近 Hard Sigmoid 函数。

$$y = c\frac{1}{1+e^{-(b+wx_1)}} \tag{1.16}$$

调整这里的 b、w 和 c 可以制造各种不同形状的 Sigmoid 函数，用各种不同形状的 Sigmoid 函数去逼近 Hard Sigmoid 函数。如图 1.11 所示，如果改 w，就会改变斜率，就会改变斜坡的坡度。如果改了 b，就可以把这一个 Sigmoid 函数左右移动；如果改 c，就可以改变它的高度。所以只要有不同的 w 不同的 b 不同的 c，就可以制造出不同的 Sigmoid 函数，把不同的 Sigmoid 函数叠起来以后就可以去逼近各种不同的分段线性函数；分段线性函数可以拿来近似各种不同的连续的函数。

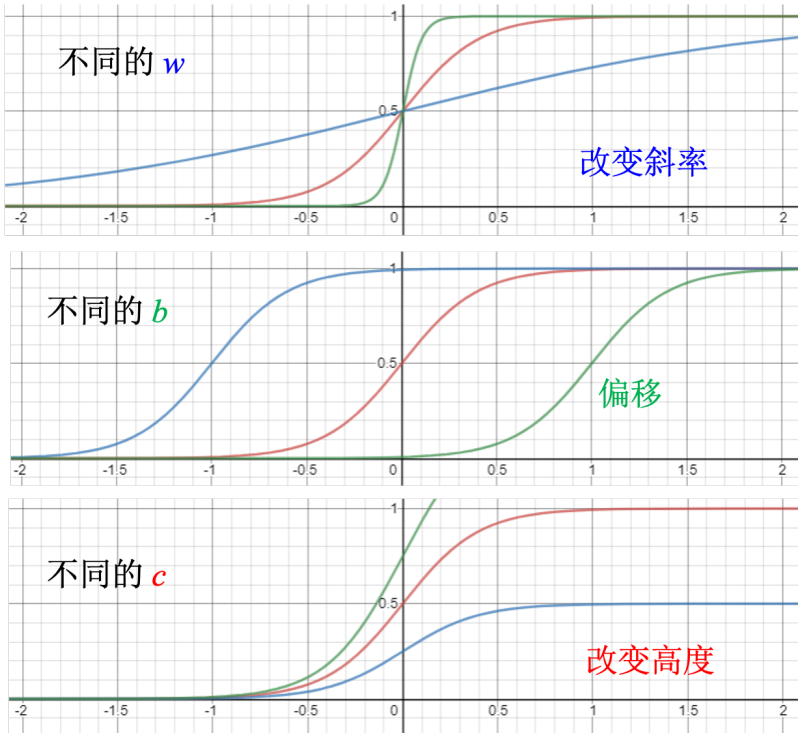


图 1.11　调整参数，制造不同的 Sigmoid 函数

如图 1.12 所示，红色这条线就是 0 加 1+2+3，而 1、2、3 都是蓝色的函数，其都可写成 $(b+wx_1)$，去做 Sigmoid 再乘上 c_i1，只是 1、2、3 的 w、b、c 不同。

$$y = b + \sum_i c_i \sigma(b_i + w_i x_1) \tag{1.17}$$

所以这边每一个式子都代表了一个不同蓝色的函数，求和就是把不同的蓝色的函数相加，再加一个常数 b。假设里面的 b 跟 w 跟 c，它是未知的，它是未知的参数，就可以设定不同的 b 跟 w 跟 c，就可以制造不同的蓝色的函数，制造不同的蓝色的函数叠起来以后，就可以制造出不同的红色的曲线，就可以制造出不同的分段线性曲线，逼近各式各样不同的连续函数。

此外，我们可以不只用一个特征 x_1，可以用多个特征代入不同的 c, b, w，组合出各种不同的函数，从而得到更有**灵活性（flexibility）**的函数，如图 1.13 所示。用 j 来代表特征的编号。如果要考虑前 28 天，j 就是 1 到 28。

直观来讲，先考虑一下 j 就是 1、2、3 的情况，就是只考虑 3 个特征。举个例子，只考虑前一天、前两天跟前 3 天的情况，所以 j 等于 1,2,3，所以输入就是 x_1 代表前一天的观看

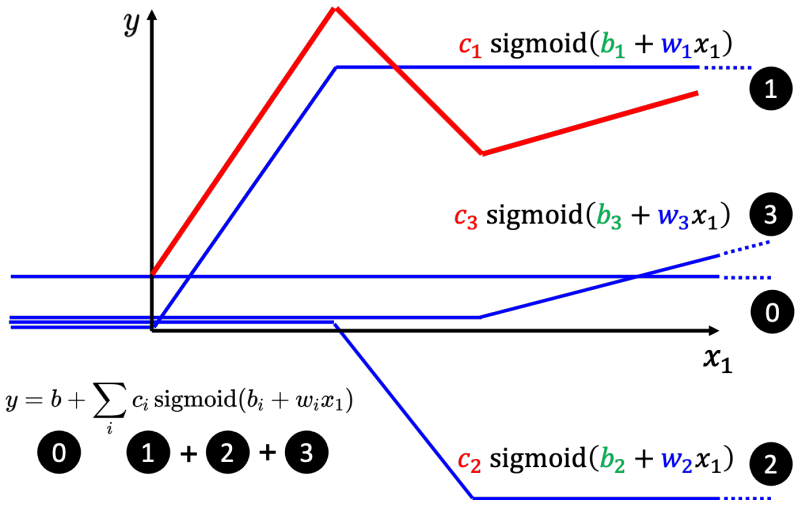


图 1.12　使用 Hard Sigmoid 来合成红色

次数，x_2 两天前观看次数，x_3 3 天前的观看次数，每一个 i 就代表了一个蓝色的函数。每一个蓝色的函数都用一个 Sigmoid 函数来比近似它，1,2,3 代表有个 Sigmoid 函数。

$$b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \tag{1.18}$$

w_{ij} 代表在第 i 个 Sigmoid 里面，乘给第 j 个特征的权重，w 的第一个下标代表是现在在考虑的是第一个 Sigmoid 函数。为了简化起见，括号里面的式子为

$$\begin{aligned} r_1 &= b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ r_2 &= b_2 + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \\ r_3 &= b_3 + w_{31}x_1 + w_{32}x_2 + w_{33}x_3 \end{aligned} \tag{1.19}$$

我们可以用矩阵跟向量相乘的方法，写一个比较简洁的写法。

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1.20}$$

将其改成线性代数比较常用的表示方式为

$$\boldsymbol{r} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x} \tag{1.21}$$

蓝框里面的括号里面做的如式 (1.21) 所示，$\boldsymbol{r}$ 对应的是 r_1, r_2, r_3。r_1, r_2, r_3 分别通过 Sigmoid 函数得到 a_1, a_2, a_3，即

$$\boldsymbol{a} = \sigma(\boldsymbol{r}) \tag{1.22}$$

因此蓝色虚线框里面做的事情，是从 x_1, x_2, x_3 得到了 a_1, a_2, a_3，如图 1.14 所示。

上面这个比较有灵活性的函数，如果用线性代数来表示，即

$$y = b + \boldsymbol{c}^{\mathrm{T}}\boldsymbol{a} \tag{1.23}$$

接下来，如图 1.15 所示，$\boldsymbol{x}$ 是特征，绿色的 $\boldsymbol{b}$ 是一个向量，灰色的 b 是一个数值。$\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}^{\mathrm{T}}, b$ 是未知参数。把这些东西通通拉直，“拼”成一个很长的向量，我们把 $\boldsymbol{W}$ 的每一行或者是每一列

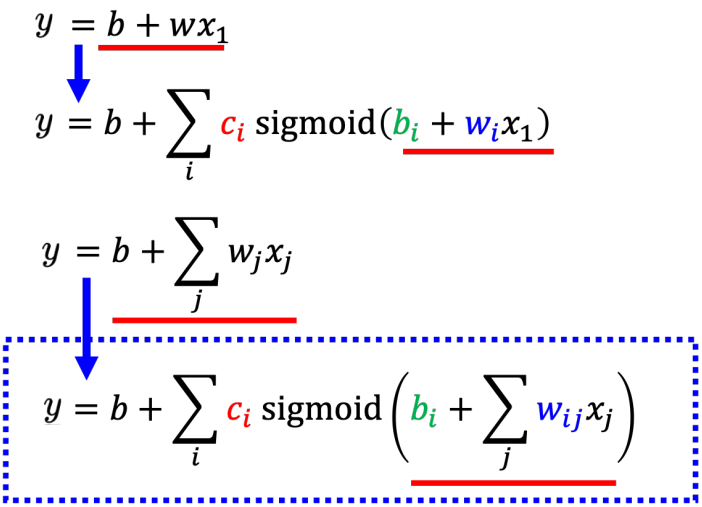


图 1.13 构建更有灵活性的函数

拿出来。无论是拿行或拿列都可以，把 $\boldsymbol{W}$ 的每一列或每一行“拼”成一个长的向量，把 $\boldsymbol{b}, \boldsymbol{c}^{\mathrm{T}}, b$ ” 拼” 上来，这个长的向量直接用 $\boldsymbol{\theta}$ 来表示。所有的未知的参数，一律统称 $\boldsymbol{\theta}$。

Q: 优化是找一个可以让损失最小的参数，是否可以穷举所有可能的未知参数的值?
A：只有 w 跟 b 两个参数的前提之下，可以穷举所有可能的 w 跟 b 的值，所以在参数很少的情况下。甚至可能不用梯度下降，不需要优化的技巧。但是参数非常多的时候，就不能使用穷举的方法，需要梯度下降来找出可以让损失最低的参数。

Q：刚才的例子里面有 3 个 Sigmoid，为什么是 3 个，能不能 4 个或更多?
A：Sigmoid 的数量是由自己决定的，而且 Sigmoid 的数量越多，可以产生出来的分段线性函数就越复杂。Sigmoid 越多可以产生有越多段线的分段线性函数，可以逼近越复杂的函数。Sigmoid 的数量也是一个超参数。

接下来要定义损失。之前是 $L(w, b)$，因为 w 跟 b 是未知的。现在未知的参数很多了，再把它一个一个列出来太累了，所以直接用 $\boldsymbol{\theta}$ 来统设所有的参数，所以损失函数就变成 $L(\boldsymbol{\theta})$。损失函数能够判断 $\boldsymbol{\theta}$ 的好坏，其计算方法跟刚才只有两个参数的时候是一样的。

先给定 $\boldsymbol{\theta}$ 的值，即某一组 $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}^{\mathrm{T}}, b$ 的值，再把一种特征 $\boldsymbol{x}$ 代进去，得到估测出来的 y，再计算一下跟真实的标签之间的误差 e。把所有的误差通通加起来，就得到损失。

接下来下一步就是优化

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \end{bmatrix} \tag{1.24}$$

要找到 $\boldsymbol{\theta}$ 让损失越小越好，可以让损失最小的一组 $\boldsymbol{\theta}$ 称为 $\boldsymbol{\theta}^*$。一开始要随机选一个初始

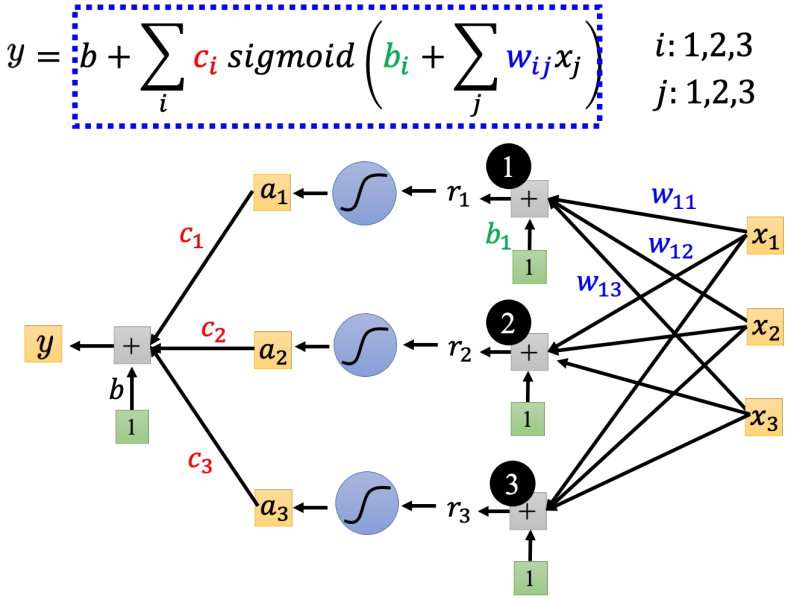


图 1.14　比较有灵活性函数的计算过程

的数值 $\boldsymbol{\theta}_0$。接下来计算每一个未知的参数对 L 的微分，得到向量 $\boldsymbol{g}$，即可以让损失变低的函数

$$\boldsymbol{g} = \nabla L\left(\boldsymbol{\theta}_0\right) \tag{1.25}$$

$$\boldsymbol{g} = \begin{bmatrix} \left.\frac{\partial L}{\partial \theta_1}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \\ \left.\frac{\partial L}{\partial \theta_2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \\ \vdots \end{bmatrix} \tag{1.26}$$

假设有 1000 个参数，这个向量的长度就是 1000，这个向量也称为梯度，∇L 代表梯度。$L(\boldsymbol{\theta}_0)$ 是指计算梯度的位置，是在 $\boldsymbol{\theta}$ 等于 θ_0 的地方。计算出 $\boldsymbol{g}$ 后，接下来跟新参数，$\boldsymbol{\theta}^0$ 代表它是一个起始的值，它是一个随机选的起始的值，代表 θ_1 更新过一次的结果，θ_2^0 减掉微分乘以，减掉 η 乘上微分的值，得到 θ_2^1，以此类推，就可以把 1000 个参数都更新了。

$$\begin{bmatrix} \theta_1^1 \\ \theta_1^2 \\ \vdots \end{bmatrix} \leftarrow \begin{bmatrix} \theta_0^1 \\ \theta_0^2 \\ \vdots \end{bmatrix} - \begin{bmatrix} \left.\eta\frac{\partial L}{\partial \theta_1}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \\ \left.\eta\frac{\partial L}{\partial \theta_2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \\ \vdots \end{bmatrix} \tag{1.27}$$

$$\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_0 - \eta \boldsymbol{g} \tag{1.28}$$

假设参数有 1000 个，$\boldsymbol{\theta}_0$ 就是 1000 个数值，1000 维的向量，$\boldsymbol{g}$ 是 1000 维的向量，$\boldsymbol{\theta}_1$ 也是 1000 维的向量。 整个操作就是这样，由 $\boldsymbol{\theta}_0$ 算梯度，根据梯度去把 $\boldsymbol{\theta}_0$ 更新成 $\boldsymbol{\theta}_1$，再算一次梯度，再根据梯度把 $\boldsymbol{\theta}_1$ 再更新成 $\boldsymbol{\theta}_2$，再算一次梯度把 $\boldsymbol{\theta}_2$ 更新成 $\boldsymbol{\theta}_3$，以此类推，直到不想

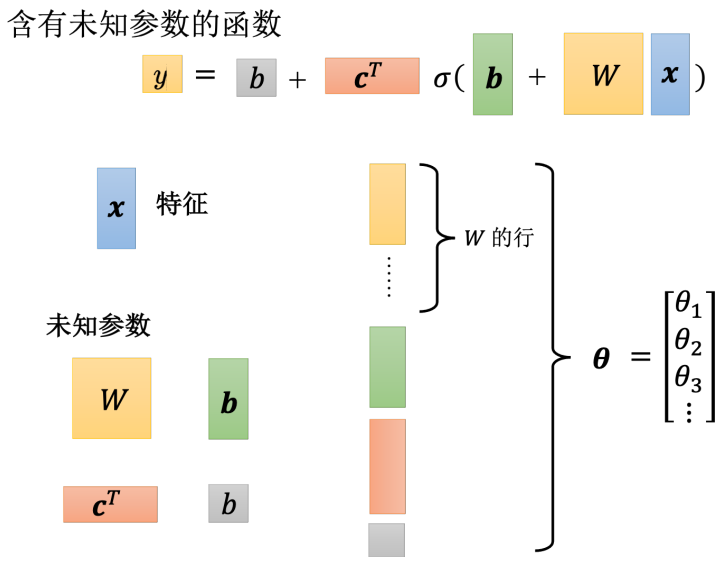


图 1.15 未知参数“拼”成一个向量

做。或者计算出梯度为 **0** 向量，导致无法再更新参数为止，不过在实现上几乎不太可能梯度为 **0**，通常会停下来就是我们不想做了。

- （随机）选取初始值 $\boldsymbol{\theta}_0$
- 计算梯度 $\boldsymbol{g} = \nabla L(\boldsymbol{\theta}_0)$

  更新 $\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_0 - \eta \boldsymbol{g}$
- 计算梯度 $\boldsymbol{g} = \nabla L(\boldsymbol{\theta}_1)$

  更新 $\boldsymbol{\theta}_2 \leftarrow \boldsymbol{\theta}_1 - \eta \boldsymbol{g}$
- 计算梯度 $\boldsymbol{g} = \nabla L(\boldsymbol{\theta}_2)$

  更新 $\boldsymbol{\theta}_3 \leftarrow \boldsymbol{\theta}_2 - \eta \boldsymbol{g}$

图 1.16 使用梯度下降更新参数

但实现上有个细节的问题，实际使用梯度下降的时候，如图 1.17 所示，会把 N 笔数据随机分成一个一个的**批量（batch）**，一组一组的。每个批量里面有 B 笔数据，所以本来有 N 笔数据，现在 B 笔数据一组，一组叫做批量。本来是把所有的数据拿出来算一个损失，现在只拿一个批量里面的数据出来算一个损失，记为 L_1 跟 L 以示区别。假设 B 够大，也许 L 跟 L_1 会很接近。所以实现上每次会先选一个批量，用该批量来算 L_1，根据 L_1 来算梯度，再用梯度来更新参数，接下来再选下一个批量算出 L_2，根据 L_2 算出梯度，再更新参数，再取下一个批量算出 L_3，根据 L_3 算出梯度，再用 L_3 算出来的梯度来更新参数。

所以并不是拿 L 来算梯度，实际上是拿一个批量算出来的 L_1, L_2, L_3 来计算梯度。把所有的批量都看过一次，称为一个**回合（epoch）**，每一次更新参数叫做一次更新。更新跟回合是不同的东西。每次更新一次参数叫做一次更新，把所有的批量都看过一遍，叫做一个回合。

更新跟回合的差别，举个例子，假设有 10000 笔数据，即 N 等于 10000，批量的大小是设 10，也就 B 等于 10。10000 个**样本（example）**形成了 1000 个批量，所以在一个回合里面更新了参数 1000 次，所以一个回合并不是更新参数一次，在这个例子里面一个回合，已经

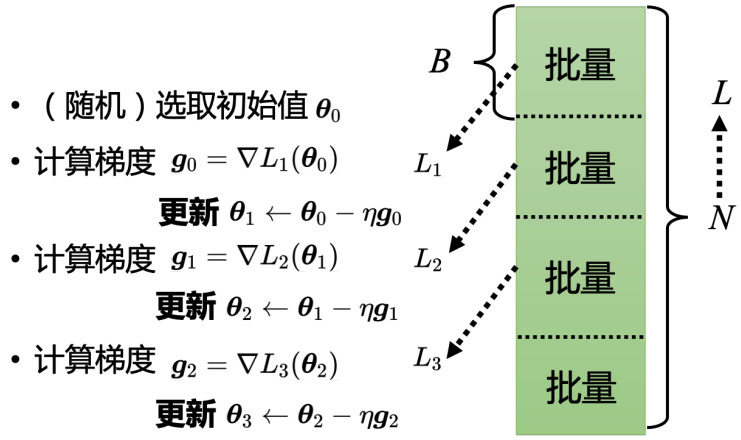


图 1.17 分批量进行梯度下降

更新了参数 1000 次了。

第 2 个例子，假设有 1000 个数据，**批量大小 (batch size)** 设 100，批量大小和 Sigmoid 的个数都是超参数。1000 个样本，批量大小设 100，1 个回合总共更新 10 次参数。所以做了一个回合的训练其实不知道它更新了几次参数，有可能 1000 次，也有可能 10 次，取决于它的批量大小有多大。

1.2.2 模型变形

其实还可以对模型做更多的变形，不一定要把 Hard Sigmoid 换成 Soft Sigmoid。Hard Sigmoid 可以看作是两个**修正线性单元 (Rectified Linear Unit，ReLU)** 的加总，ReLU 的图像有一个水平的线，走到某个地方有一个转折的点，变成一个斜坡，其对应的公式为

$$c * \max(0, b + wx_1) \tag{1.29}$$

$\max(0, b + wx_1)$ 是指看 0 跟 $b + wx_1$ 谁比较大，比较大的会被当做输出；如果 $b + wx_1 < 0$，输出是 0；如果 $b + wx_1 > 0$，输出是 $b + wx_1$。通过 w, b, c 可以挪动其位置和斜率。把两个 ReLU 叠起来就可以变成 Hard 的 Sigmoid，想要用 ReLU，就把 Sigmoid 的地方，换成 $\max(0, b_i + w_{ij}x_j)$。

如图 1.19 所示，2 个 ReLU 才能够合成一个 Hard Sigmoid。要合成 i 个 Hard Sigmoid，需要 i 个 Sigmoid，如果 ReLU 要做到一样的事情，则需要 $2i$ 个 ReLU，因为 2 个 ReLU 合起来才是一个 Hard Sigmoid。因此表示一个 Hard 的 Sigmoid 不是只有一种做法。在机器学习里面，Sigmoid 或 ReLU 称为**激活函数 (activation function)**。

当然还有其他常见的激活函数，但 Sigmoid 跟 ReLU 是最常见的激活函数，接下来的实验都选择用了 ReLU，显然 ReLU 比较好，实验结果如图 1.20 所示。如果是线性模型，考虑 56 天，训练数据上面的损失是 320，没看过的数据 2021 年数据是 460。连续使用 10 个 ReLU 作为模型，跟用线性模型的结果是差不多的，

但连续使用 100 个 ReLU 作为模型，结果就有显著差别了，100 个 ReLU 在训练数据上的损失就可以从 320 降到 280，有 100 个 ReLU 就可以制造比较复杂的曲线，本来线性就是一直线，但 100 个 ReLU 就可以产生 100 个折线的函数，在测试数据上也好了一些. 接下来

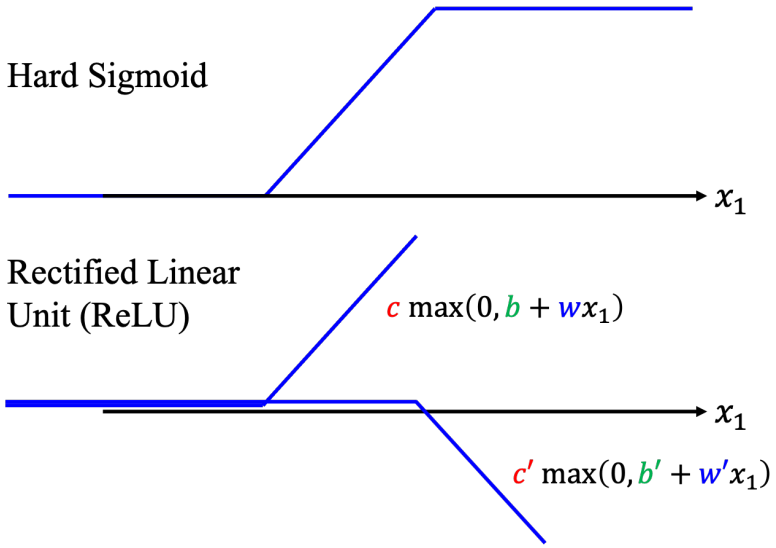


图 1.18　ReLU 函数

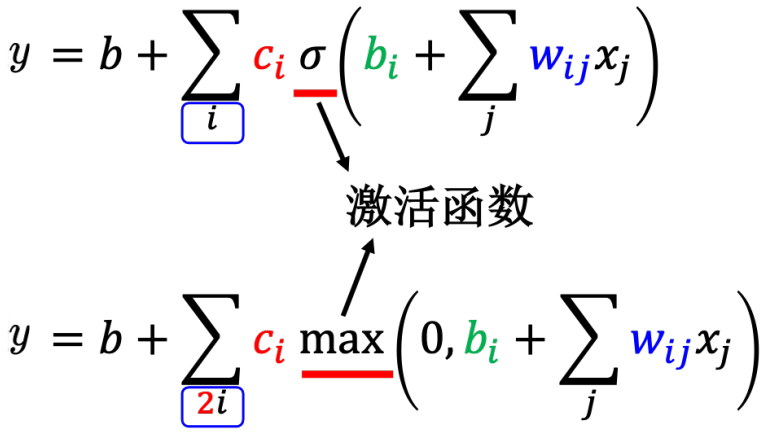


图 1.19　激活函数

使用 1000 个 ReLU 作为模型，在训练数据上损失更低了一些，但是在没看过的数据上，损失没有变化。

接下来可以继续改模型，如图 1.21 所示，从 x 变成 a，就是把 x 乘上 w 加 b，再通过 Sigmoid 函数。不一定要通过 Sigmoid 函数，通过 ReLU 也可以得到 a，同样的事情再反复地多做几次。 所以可以把 x 做这一连串的运算产生 a，接下来把 a 做这一连串的运算产生 a'。反复地多做的次数又是另外一个超参数。注意，w, b 和 w', b' 不是同一个参数，是增加了更多的未知的参数。

每次都加 100 个 ReLU，输入特征，就是 56 天前的数据。如图 1.22 所示，如果做两次，损失降低很多，280 降到 180。如果做 3 次，损失从 180 降到 140，通过 3 次 ReLU，从 280 降到 140，在训练数据上，在没看过的数据上，从 430 降到了 380。

通过 3 次 ReLU 的实验结果如图 1.23 所示。横轴就是时间，纵轴是观看次数。红色的线是真实的数据，蓝色的线是预测出来的数据在这种低点的地方啊，看红色的数据是每隔一段时间，就会有两天的低点，在低点的地方，机器的预测还算是蛮准确的，机器高估了真实的观

	线性	10 ReLU	100 ReLU	1000 ReLU
2017 – 2020	320	320	280	270
2021	460	450	430	430

图 1.20　激活函数实验结果

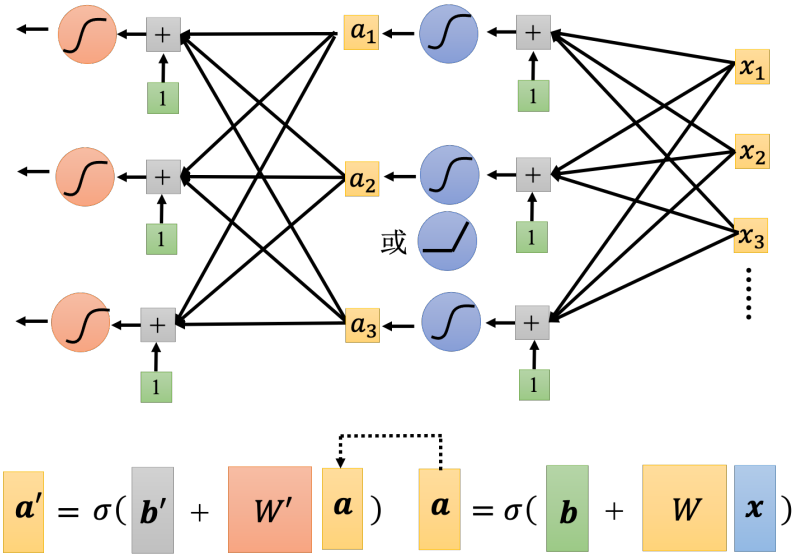


图 1.21　改进模型

看人次，尤其是在红圈标注的这一天，这一天有一个很明显的低谷，但是机器没有预测到这一天有明显的低谷，它是晚一天才预测出低谷。这天最低点就是除夕。但机器只知道看前 56 天的值，来预测下一天会发生什么事，所以它不知道那一天是除夕。

如图 1.24 所示，Sigmoid 或 ReLU 称为神经元（neuron），很多的神经元称为**神经网络 (neural network)**。人脑中就是有很多神经元，很多神经元串起来就是一个神经网络，跟人脑是一样的。人工智能就是在模拟人脑。神经网络不是新的技术，80、90 年代就已经用过了，后来为了要重振神经网络的雄风，所以需要新的名字。每一排称为一层，称为**隐藏层 (hidden layer)**，很多的隐藏层就“深”，这套技术称为深度学习。

所以人们把神经网络越叠越多越叠越深，2012 年的 AlexNet 有 8 层它的错误率是 16.4%，两年之后 VGG 有 19 层，错误率在图像识别上进步到 7.3 %。这都是在图像识别上一个基准的数据库（ImageNet）上面的结果，后来 GoogleNet 有 22 层，错误率降到 6.7%。而**残差网络（Residual Network，ResNet）**有 152 层，错误率降到 3.57%。

刚才只做到 3 层，应该要做得更深，现在网络都是叠几百层的，深度学习就要做更深。但 4 层在训练数据上，损失是 100，在没有看过 2021 年的数据上，损失是 440。在训练数据上，

	1 层	2 层	3 层	4 层
2017 – 2020	280	180	140	100
2021	430	390	380	440

图 1.22　使用 ReLU 的实验结果

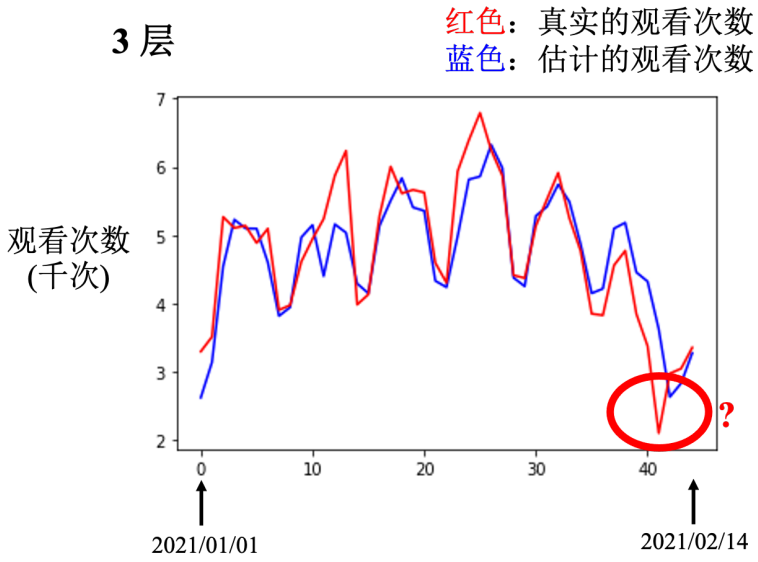


图 1.23　使用 3 次 ReLU 的实验结果

3 层比 4 层差，但是在没看过的数据上，4 层比较差，3 层比较好，如图 1.25 所示。在训练数据和测试数据上的结果是不一致的，这种情况称为**过拟合（overfitting）**。

但是做到目前为止，还没有真的发挥这个模型的力量，2021 年的数据到 2 月 14 日之前的数据是已知的。要预测未知的数据，选 3 层的网络还是 4 层的网络呢？假设今天是 2 月 26 日，今天的观看次数是未知的，如果用已经训练出来的神经网络预测今天的观看次数。要选 3 层的，虽然 4 层在训练数据上的结果比较好，但在没有看过的数据的结果更重要。应该选一个在训练的时候，没有看过的数据上表现会好的模型，所以应该选 3 层的网络。深度学习的训练会用到**反向传播（BackPropagation，BP）**，其实它就是比较有效率、算梯度的方法。

1.2.3　机器学习框架

我们会有一堆训练的数据以及测试数据如式 (1.30) 所示，测试集就是只有 x 没有 y。

$$\begin{aligned} &\text{训练数据：} \left\{\left(x^1, y^1\right), \left(x^2, y^2\right), \ldots, \left(x^N, y^N\right)\right\} \\ &\text{测试数据：} \left\{x^{N+1}, x^{N+2}, \ldots, x^{N+M}\right\} \end{aligned} \tag{1.30}$$

训练集就要拿来训练模型，训练的过程是 3 个步骤。

1. 先写出一个有未知数 $\boldsymbol{\theta}$ 的函数，$\boldsymbol{\theta}$ 代表一个模型里面所有的未知参数。$f_{\boldsymbol{\theta}}(x)$ 的意思就是函数叫 $f_{\boldsymbol{\theta}}(x)$，输入的特征为 x，;
2. 定义损失，损失是一个函数，其输入就是一组参数，去判断这一组参数的好坏;
3. 解一个优化的问题，找一个 $\boldsymbol{\theta}$，该 $\boldsymbol{\theta}$ 可以让损失的值越小越好。让损失的值最小的 $\boldsymbol{\theta}$ 为 $\boldsymbol{\theta}^*$，即

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L \tag{1.31}$$

有了 $\boldsymbol{\theta}^*$ 以后，就把它拿来用在测试集上，也就是把 $\boldsymbol{\theta}^*$ 带入这些未知的参数，本来 $f_{\boldsymbol{\theta}}(x)$ 里面有一些未知的参数，现在 $\boldsymbol{\theta}$ 用 $\boldsymbol{\theta}^*$ 来取代，输入是测试集，输出的结果存起来，上传到 Kaggle 就结束了。

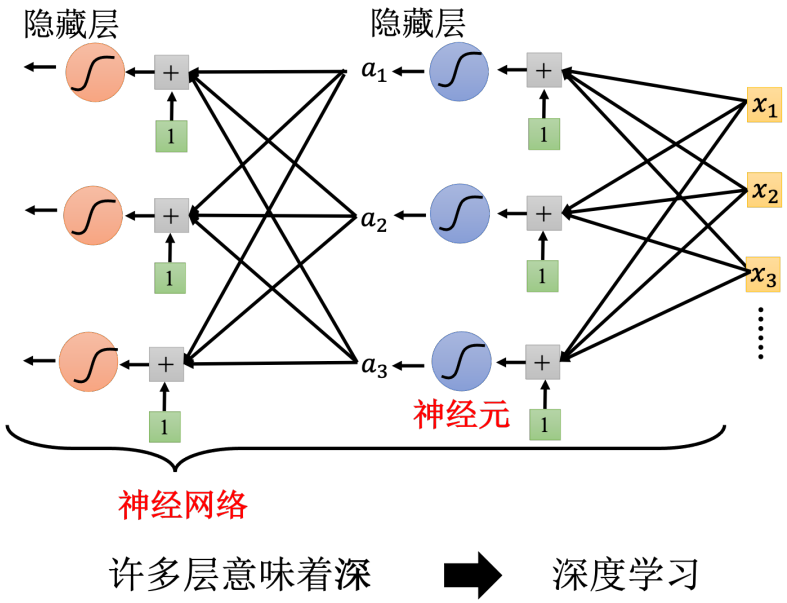


图 1.24　深度学习的结构

	1 层	2 层	3 层	4 层
2017 – 2020	280	180	140	100
2021	430	390	380	440

图 1.25　模型有过拟合问题

第 2 章 实践方法论

在应用机器学习算法时，实践方法论能够帮助我们更好地训练模型。如果在 Kaggle 上的结果不太好，虽然 Kaggle 上呈现的是测试数据的结果，但要先检查训练数据的损失。看看模型在训练数据上面，有没有学起来，再去看测试的结果，如果训练数据的损失很大，显然它在训练集上面也没有训练好。接下来再分析一下在训练集上面没有学好的原因。

2.1 模型偏差

模型偏差可能会影响模型训练。举个例子，假设模型过于简单，一个有未知参数的函数代 $\boldsymbol{\theta}_1$ 得到一个函数 $f_{\boldsymbol{\theta}_1}(\boldsymbol{x})$，同理可得到另一个函数 $f_{\boldsymbol{\theta}_2}(\boldsymbol{x})$，把所有的函数集合起来得到一个函数的集合。但是该函数的集合太小了，没有包含任何一个函数，可以让损失变低的函数不在模型可以描述的范围内。在这种情况下，就算找出了一个 $\boldsymbol{\theta}^*$，虽然它是这些蓝色的函数里面最好的一个，但损失还是不够低。这种情况就是想要在大海里面捞针（一个损失低的函数），结果针根本就不在海里。

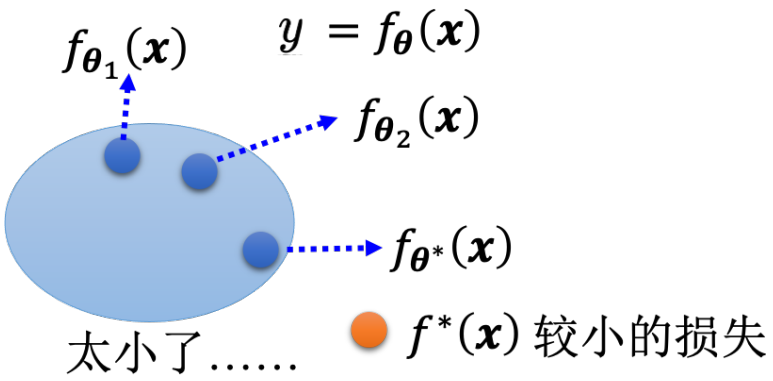


图 2.1 模型太简单的问题

这个时候重新设计一个模型，给模型更大的灵活性。以第一章的预测未来观看人数为例，可以增加输入的特征，本来输入的特征只有前一天的信息，假设要预测接下来的观看人数，用前一天的信息不够多，用 56 天前的信息，模型的灵活性就比较大了。也可以用深度学习，增加更多的灵活性。所以如果模型的灵活性不够大，可以增加更多特征，可以设一个更大的模型，可以用深度学习来增加模型的灵活性，这是第一个可以的解法。但是并不是训练的时候，损失大就代表一定是模型偏差，可能会遇到另外一个问题：优化做得不好。

2.2 优化问题

一般只会用到梯度下降进行优化，这种优化的方法很多的问题。比如可能会卡在局部最小值的地方，无法找到一个真的可以让损失很低的参数，如图 2.3(a) 所示。如图 2.3(b) 所示蓝色部分是模型可以表示的函数所形成的集合，可以把 $\boldsymbol{\theta}$ 代入不同的数值，形成不同的函数，把所有的函数通通集合在一起，得到这个蓝色的集合。这个蓝色的集合里面，确实包含了一些函数，这些函数它的损失是低的。但问题是梯度下降这一个算法无法找出损失低的函数，梯度下降是解一个优化的问题，找到 $\boldsymbol{\theta}^*$ 就结束了。但 $\boldsymbol{\theta}^*$ 的损失不够低。这个模型里面存在着某

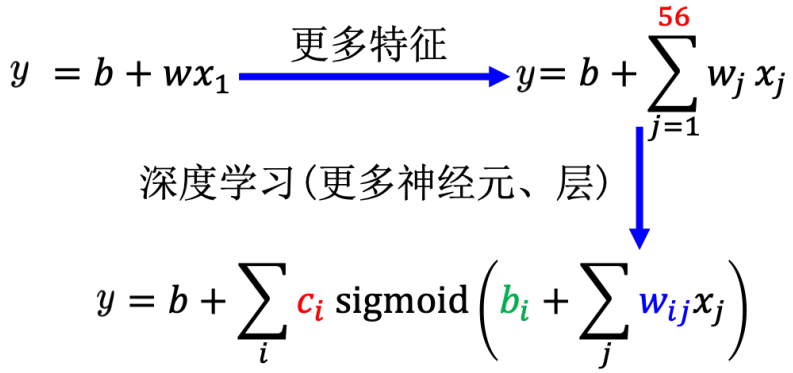
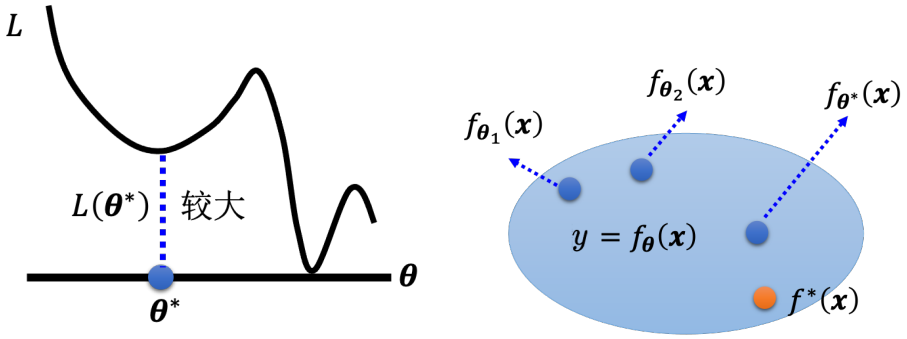


图 2.2　增加模型的灵活性

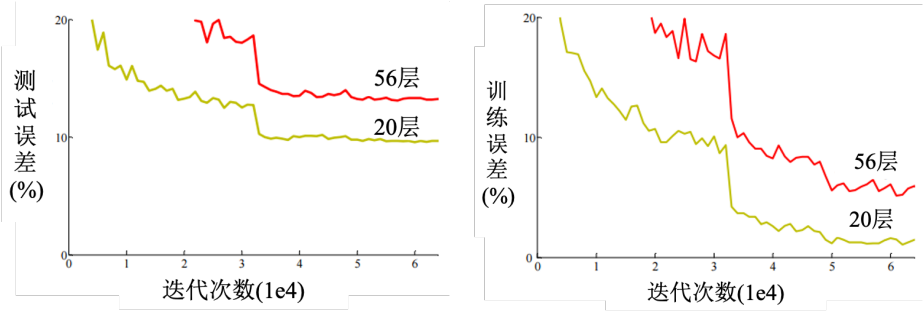
一个函数的损失是够低的，梯度下降没有给这一个函数。这就像是想大海捞针，针确实在海里，但是无法把针捞起来。训练数据的损失不够低的时候，到底是模型偏差，还是优化的问题呢。找不到一个损失低的函数，到底是因为模型的灵活性不够，海里面没有针。还是模型的灵活性已经够了，只是优化梯度下降不给力，它没办法把针捞出来 到底是哪一个。到底模型已经够大了，还是它不够大，怎么判断这件事呢?



(a) 优化过程中会卡在局部最小值　　(b) 模型的灵活性足够大

图 2.3　优化方法的问题

一个建议判断的方法，通过比较不同的模型来判断模型现在到底够不够大。举个例子，这一个实验是从残差网络的论文“Deep Residual Learning for Image Recognition”[1] 里面节录出来的。这篇论文在测试集上测试两个网络，一个网络有 20 层，一个网络有 56 层。图 2.4(a) 横轴指的是训练的过程，就是参数更新的过程，随着参数的更新，损失会越来越低，但是结果 20 层的损失比较低，56 层的损失还比较高。残差网络是比较早期的论文，2015 年的论文。很多人看到这张图认为这个代表过拟合，深度学习不奏效，56 层太深了不奏效，根本就不需要这么深。但 这个不是过拟合，并不是所有的结果不好，都叫做过拟合。在训练集上，20 层的网络损失其实是比较低的，56 层的网络损失是比较高的，如图 2.4(b) 所示，这代表 56 层的网络的优化没有做好，它的优化不给力。



(a) 测试数据 (b) 训练数据

图 2.4 残差网络的例子

Q：如何知道是 56 层的优化不给力，搞不好是模型偏差，搞不好是 56 层的网络的模型灵活性还不够大，它要 156 层才好，56 层也许灵活性还不够大?
A：但是比较 56 层跟 20 层，20 层的损失都已经可以做到这样了，56 层的灵活性一定比 20 层更大。如果 56 层的网络要做到 20 层的网络可以做到的事情，对它来说是轻而易举的。它只要前 20 层的参数，跟这个 20 层的网络一样，剩下 36 层就什么事都不做，复制前一层的输出就好了。如果优化成功，56 层的网络应该要比 20 层的网络可以得到更低的损失。但结果在训练集上面没有，这个不是过拟合，这个也不是模型偏差，因为 56 层网络灵活性是够的，这个问题是优化不给力，优化做得不够好。

这边给大家的建议是看到一个从来没有做过的问题，可以先跑一些比较小的、比较浅的网络，或甚至用一些非深度学习的方法，比如线性模型、**支持向量机(Support Vector Machine，SVM)**，SVM 可能是比较容易做优化的，它们比较不会有优化失败的问题。也就是这些模型它会竭尽全力的，在它们的能力范围之内，找出一组最好的参数，它们比较不会有失败的问题。因此可以先训练一些比较浅的模型，或者是一些比较简单的模型，先知道这些简单的模型，到底可以得到什么样的损失。

接下来还缺一个深的模型，如果深的模型跟浅的模型比起来，深的模型明明灵活性比较大，但损失却没有办法比浅的模型压得更低代表说优化有问题，梯度下降不给力，因此要有一些其它的方法来更好地进行优化。

举个观看人数预测的例子，如图 2.5 所示，在训练集上面，2017 年到 2020 年的数据是训练集，1 层的网络的损失是 280，2 层就降到 180，3 层就降到 140，4 层就降到 100。但是测 5 层的时候结果变成 340。损失很大显然不是模型偏差的问题，因为 4 层都可以做到 100 了，5 层应该可以做得更低。这个是优化的问题，优化做得不好才会导致造成这样子的问题。如果训练损失大，可以先判断是模型偏差还是优化。如果是模型偏差，就把模型变大。假设经过努力可以让训练数据的损失变小，接下来可以来看测试数据损失；如果测试数据损失也小，比这个较强的基线模型还要小，就结束了。

但如果训练数据上面的损失小，测试数据上的损失大，可能是真的过拟合。在测试上的结果不好，不一定是过拟合。要把训练数据损失记下来，先确定优化没有问题，模型够大了。接下来才看看是不是测试的问题，如果是训练损失小，测试损失大，这个有可能是过拟合。

	1 层	2 层	3 层	4 层	5 层
2017 – 2020	280	180	140	100	340

图 2.5　层数越深，损失反而变大

2.3　过拟合

为什么会有过拟合这样的情况呢？举一个极端的例子，这是训练集。假设根据这些训练集，某一个很废的机器学习的方法找出了一个一无是处的函数。这个一无是处的函数，只要输入 x 有出现在训练集里面，就把它对应的 y 当做输出。如果 x 没有出现在训练集里面，就输出一个随机的值。这个函数啥事也没有干，其是一个一无是处的函数，但它在训练数据上的损失是 0。把训练数据通通丢进这个函数里面，它的输出跟训练集的标签是一模一样的，所以在训练数据上面，这个函数的损失可是 0 呢，可是在测试数据上面，它的损失会变得很大，因为它其实什么都没有预测，这是一个比较极端的例子，在一般的情况下，也有可能发生类似的事情。

如图 2.6 所示，举例来说，假设输入的特征为 x，输出为 y，x 和 y 都是一维的。x 和 y 之间的关系是 2 次的曲线，曲线用虚线来表示，因为通常没有办法，直接观察到这条曲线。我们真正可以观察到的是训练集，训练集可以想像成从这条曲线上面，随机采样出来的几个点。模型的能力非常的强，其灵活性很大，只给它这 3 个点。在这 3 个点上面，要让损失低，所以模型的这个曲线会通过这 3 个点，但是其它没有训练集做为限制的地方，因为它的灵活性很大，它灵活性很大，所以模型可以变成各式各样的函数，没有给它数据做为训练，可以产生各式各样奇怪的结果。

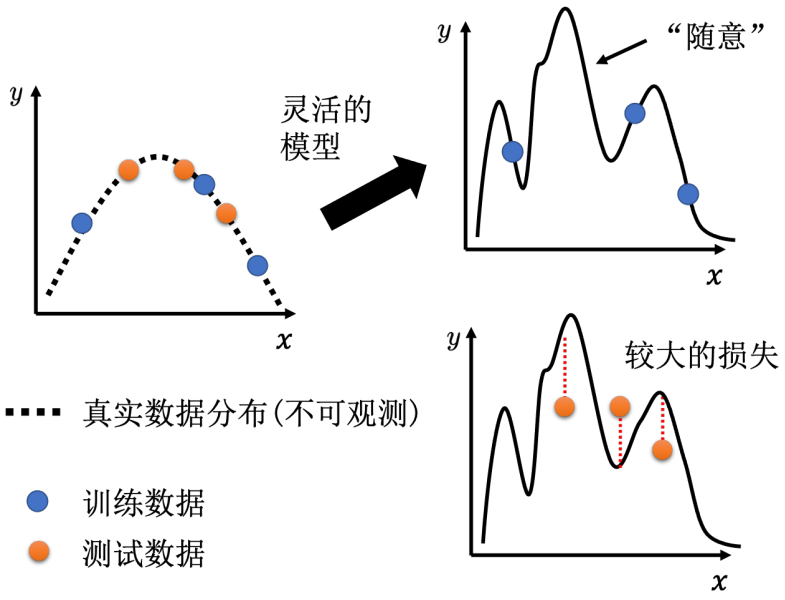


图 2.6　模型灵活导致的问题

如果再丢进测试数据，测试数据和训练数据，当然不会一模一样，它们可能是从同一个分布采样出来的，测试数据是橙色的点，训练数据是蓝色的点。用蓝色的点，找出一个函数以后，测试在橘色的点上，不一定会好。如果模型它的自由度很大的话，它可以产生非常奇怪的

曲线，导致训练集上的结果好，但是测试集上的损失很大。

怎么解决过拟合的问题呢，有两个可能的方向：

第一个方向是往往是最有效的方向，即增加训练集。因此如果训练集，蓝色的点变多了，虽然模型它的灵活性可能很大，但是因为点非常多，它就可以限制住，它看起来的形状还是会很像，产生这些数据背后的 2 次曲线，如图 2.7 所示。可以做**数据增强 (data augmentation，)**，这个方法并不算是使用了额外的数据。

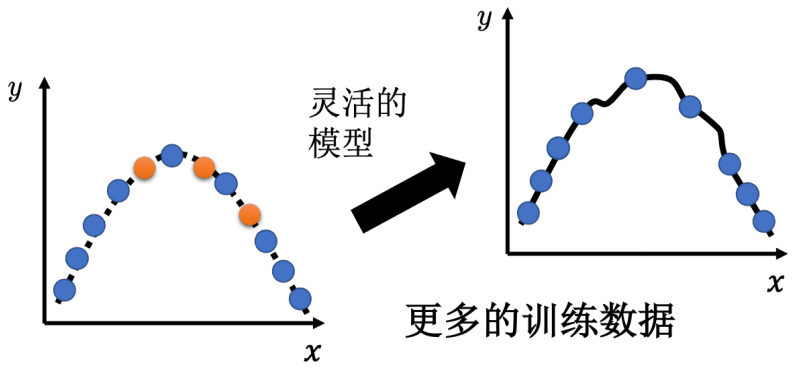


图 2.7　增加数据

数据增强就是根据问题的理解创造出新的数据。举个例子，在做图像识别的时候，常做的一个招式是，假设训练集里面有某一张图片，把它左右翻转，或者是把它其中一块截出来放大等等。对图片进行左右翻转，数据就变成两倍。但是数据增强不能够随便乱做。在图像识别里面，很少看到有人把图像上下颠倒当作增强。因为这些图片都是合理的图片，左右翻转图片，并不会影响到里面的内容。但把图像上下颠倒，可能不是一个训练集或真实世界里面会出现的图像。如果给机器根据奇怪的图像学习，它可能就会学到奇怪的东西。所以数据增强，要根据对数据的特性以及要处理的问题的理解，来选择合适的数据增强的方式。

另外一个解法是给模型一些限制，让模型不要有过大的灵活性。假设 x 跟 y 背后的关系其实就是一条 2 次曲线，只是该 2 次曲线里面的参数是未知的。如图 2.8 所示，要用多限制的模型才会好取决于对这个问题的理解。因为这种模型是自己设计的，设计出不同的模型，结果不同。假设模型是 2 次曲线，在选择函数的时候有很大的限制，因为 2 次曲线要就是这样子，来来去去就是几个形状而已。所以当训练集有限的时候，来来去去只能够选几个函数。所以虽然说只给了 3 个点，但是因为能选择的函数有限，可能就会正好选到跟真正的分布比较接近的函数，在测试集上得到比较好的结果。

解决过拟合的问题，要给模型一些限制，最好模型正好跟背后产生数据的过程，过程是一样的就有机会得到好的结果。给模型制造限制可以有如下方法：

- 给模型比较少的参数。如果是深度学习的话，就给它比较少的神经元的数量，本来每层一千个神经元，改成一百个神经元之类的，或者让模型共用参数，可以让一些参数有一样的数值。**全连接网络（fully-connected network）**其实是一个比较有灵活性的架构，而**卷积神经网络（Convolutional Neural Network，CNN）**是一个比较有限制的架构。CNN 是一种比较没有灵活性的模型，其是针对图像的特性来限制模型的灵活性。所以全连接神经网络，可以找出来的函数所形成的集合其实是比较大的，CNN 所找出来的函数，它形成的集合其实是比较小的，其实包含在全连接网络里面的，但是就是因为 CNN 给了，比较大的限制，所以 CNN 在图像上，反而会做得比较好，这个之后都还会

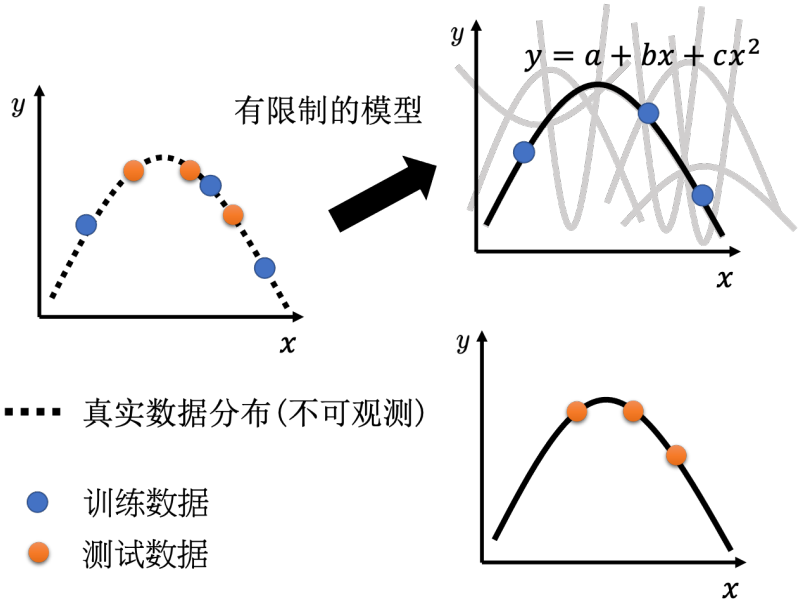


图 2.8 对模型增加限制

再提到，

- 用比较少的特征，本来给 3 天的数据，改成用给两天的数据，其实结果就好了一些。
- 还有别的方法，比如**早停(early stopping)**、**正则化(regularization)**和**丢弃法(dropout method)**。

但也不要给太多的限制。假设模型是线性的模型，图 2.9 中有 3 个点，没有任何一条直线可以同时通过这 3 个点。只能找到一条直线，这条直线跟这些点比起来，它们的距离是比较近的。这个时候模型的限制就太大了，在测试集上就不会得到好的结果。这种情况下的结果不好，并不是因为过拟合了，而是因为给模型太大的限制，大到有了模型偏差的问题。

这边产生了一个矛盾的情况，模型的复杂程度，或这样让模型的灵活性越来越大。但复杂的程度和灵活性都没有给明确的定义。比较复杂的模型包含的函数比较多，参数比较多。如图 2.10 所示，随着模型越来越复杂，训练损失可以越来越低，但测试时，当模型越来越复杂的时候，刚开始，测试损失会跟著下降，但是当复杂的程度，超过某一个程度以后，测试损失就会突然暴增了。这就是因为当模型越来越复杂的时候，复杂到某一个程度，过拟合的情况就会出现，所以在训练损失上面可以得到比较好的结果。在测试损失上面，会得到比较大的损失，可以选一个中庸的模型，不是太复杂的，也不是太简单的，刚刚好可以在训练集上损失最低，测试损失最低。

假设 3 个模型的复杂的程度不太一样，不知道要选哪一个模型才会刚刚好，在测试集上得到最好的结果。因为选太复杂的就过拟合，选太简单的有模型偏差的问题。把这 3 个模型的结果都跑出来，上传到 Kaggle 上面，损失最低的模型显然就是最好的模型，但是不建议这么做。举个极端的例子，假设有 1 到 10^{12} 个模型，这些模型学习出来的函数都是一无是处的函数。它们会做的事情就是，训练集里面有的数据就把它记下来，训练集没看过的，就直接输出随机的结果。把这 10^{12} 个模型的结果，通通上传到 Kaggle 上面，得到 10^{12} 个分数，这 10^{12} 的分数里面，结果最好的，模型也是最好的。

虽然每一个模型没看过测试数据，其输出的结果都是随机的，但不断随机，总是会找到一

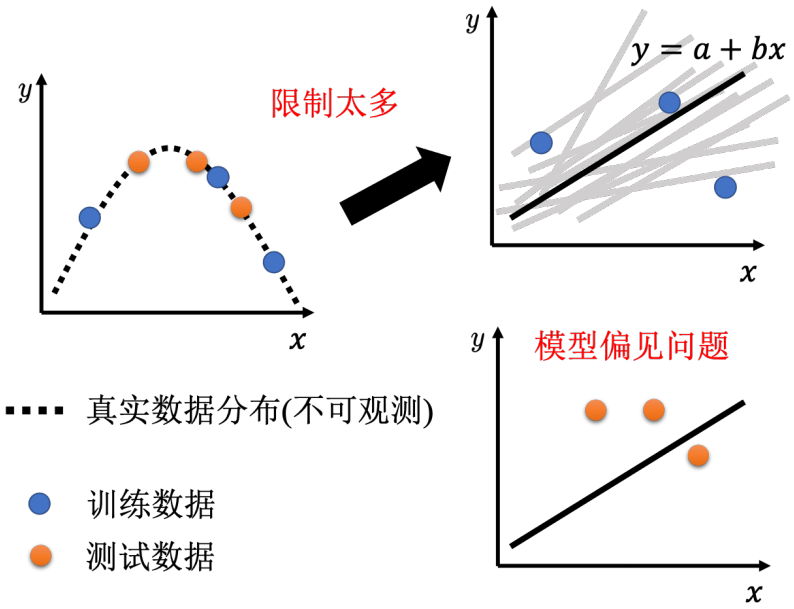


图 2.9 限制太大会导致模型偏差

个好的结果。因此也许某个模型找出来的函数，正好在测试数据上面的结果比较好，选这一个模型当作最后上传的结果，当作最后要用在私人测试集上的结果。该模型是随机的，它恰好在公开的测试数据上面得到一个好结果，但是它在私人测试集上可能仍然是随机的。测试集分成公开的数据集跟私人的数据集，公开的分数可以看到，私人的分数要截止日期以后才知道。如果根据公开数据集来选择模型，可能会出现这种情况：在公开的排行榜上面排前十，但是截止日期一结束，可能掉到 300 名之外。因为计算分数的时候，会同时考虑公开和私人的分数。

Q：为什么要把测试集分成公开和私人？
A：假设所有的数据都是公开，就算是一个一无是处的模型，它也有可能在公开的数据上面得到好的结果。如果只有公开的测试集，没有私人测试集，写一个程序不断随机产生输出就好，不断把随机的输出上传到 Kaggle，可以随机出一个好的结果。这个显然没有意义。而且如果公开的测试数据是公开的，公开的测试数据的结果是已知的，一个很废的模型也可能得到非常好的结果。不要用公开的测试集调模型，因为可能会在私人测试集上面得到很差的结果，不过因为在公开测试集上面的好的结果也有算分数。

2.4 交叉验证

比较合理选择模型的方法是把训练的数据分成两半，一部分称为**训练集（training set）**，一部分是**验证集（validation set）**。比如 90% 的数据作为训练集，有 10% 的数据作为验证集。在训练集上训练出来的模型会使用验证集来衡量它们的分数，根据验证集上面的分数去挑选结果，再把这个结果上传到 Kaggle 上面得到的公开分数。在挑分数的时候，是用验证集来挑模型，所以公开测试集分数就可以反映私人测试集的分数。但假设这个循环做太多次，根据公开测试集上的结果调整模型太多次，就又有可能在公开测试集上面过拟合，在私人测试集上面得到差的结果。不过上传的次数有限制，所以无法走太多次循环，可以避免在公开的测

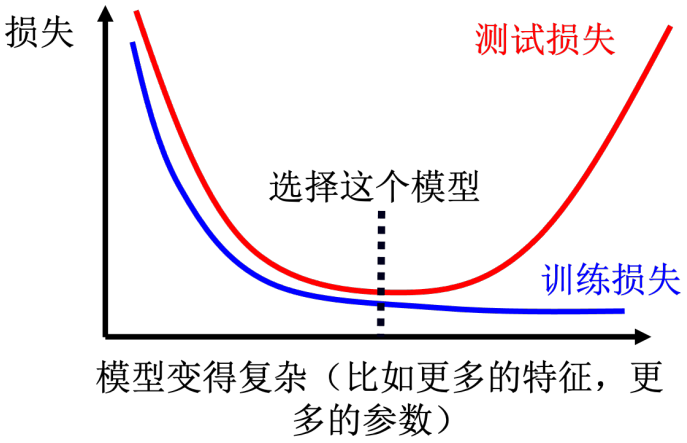


图 2.10　模型的复杂程度与损失的关系

试集上面的结果过拟合。根据过去的经验，就在公开排行榜上排前几名的，往往私人测试集很容易就不好。

其实最好的做法，就是用验证损失，最小的直接挑就好了，不要管公开测试集的结果。在实现上，不太可能这么做，因为公开数据集的结果对模型的选择，可能还是会有些影响的。理想上就用验证集挑就好，有过比较好的**基线（baseline）**算法以后，就不要再去动它了，就可以避免在测试集上面过拟合。但是这边会有一个问题，如果随机分验证集，可能会分得不好，分到很奇怪的验证集，会导致结果很差，如果有这个担心的话，可以用 k 折交叉验证（k-fold cross validation），如图 2.11 所示。k 折交叉验证就是先把训练集切成 k 等份。在这个例子，训练集被切成 3 等份，切完以后，拿其中一份当作验证集，另外两份当训练集，这件事情要重复 3 次。即第一份第 2 份当训练，第 3 份当验证；第一份第 3 份当训练，第 2 份当验证；第一份当验证，第 2 份第 3 份当训练。

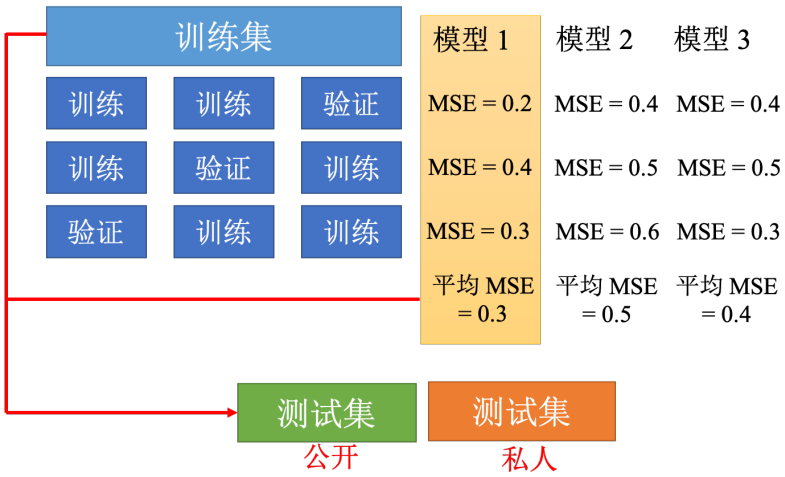


图 2.11　k 折交叉验证

接下来有 3 个模型，不知道哪一个是好的。把这 3 个模型，在这 3 个设置下，在这 3 个训练跟验证的数据集上面，通通跑过一次，把这 3 个模型，在这 3 种情况的结果都平均起来，

把每一个模型在这 3 种情况的结果，都平均起来，再看看谁的结果最好假设现在模型 1 的结果最好，3 折交叉验证得出来的结果是，模型 1 最好。再把模型 1 用在全部的训练集上，训练出来的模型再用在测试集上面。接下来也许我们要问的一个问题是，讲到预测 2 月 26 日，也就是上周五的观看人数的结果如图 2.12 所示。所以把 3 层的网络，拿来测试一下是测试的结果。

	1 层	2 层	3 层	4 层
2017 – 2020	280	180	140	100
2021	430	390	380	440

图 2.12　3 层网络的结果最好

2.5　不匹配

图 2.13 中横轴就是从 2021 年的 1 月 1 号开始一直往下，红色的线是真实的数字，蓝色的线是预测的结果。2 月 26 日是 2021 年观看人数最高的一天了，机器的预测差距非常的大，差距有 2580，所以这一天是 2021 年观看人数最多的一天。跑了一层 2 层跟四层的看看，所有的模型的结果都不好，两层跟 3 层的错误率都是 2 千多，其实四层跟一层比较好，都是 1800 左右，但是这四个模型不约而同的，觉得 2 月 26 日应该是个低点，但实际上 2 月 26 日是一个峰值，模型其实会觉得它是一个低点，也不能怪它，因为根据过去的数据，周五晚上大家都出去玩了。但是 2 月 26 日出现了反常的情况。这种情况应该算是另外一种错误的形式，这种错误的形式称为不匹配（mismatch）。

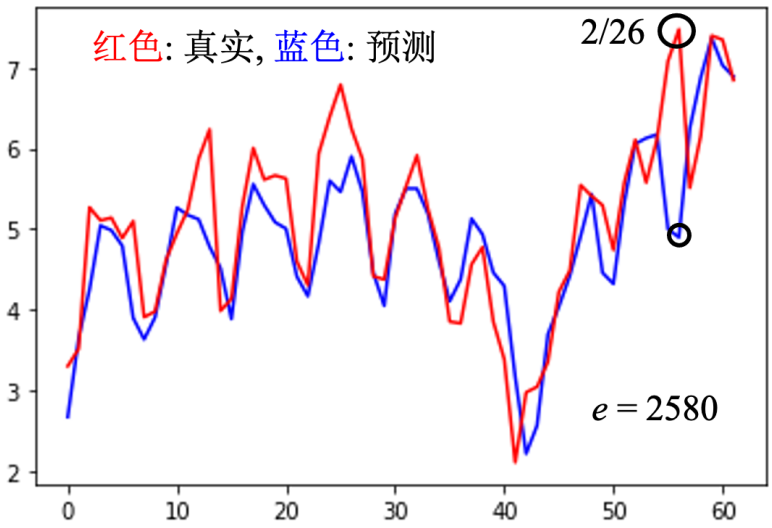


图 2.13　另一种错误形式：不匹配

不匹配跟过拟合其实不同，一般的过拟合可以用搜集更多的数据来克服，但是不匹配是指训练集跟测试集的分布不同，训练集再增加其实也没有帮助了。假设数据在分训练集跟测试集的时候，使用 2020 年的数据作为训练集，使用 2021 年的数据作为测试集，不匹配的问题可能就很严重。如果今天用 2020 年当训练集，2021 年当测试集，根本预测不准。因为 2020

年的数据跟 2021 年的数据背后的分布不同。图 2.14 是图像分类中的不匹配问题。增加数据也不能让模型做得更好，所以这种问题要怎么解决，匹不匹配要看对数据本身的理解了，我们可能要对训练集跟测试集的产生方式有一些理解，才能判断它是不是遇到了不匹配的情况。

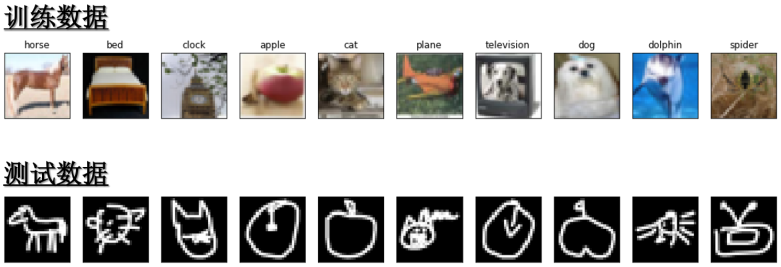


图 2.14　图像分类的不匹配问题

参考文献

[1] 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.

第 3 章 深度学习基础

本章介绍了深度学习常见的概念，理解这些概念能够帮助我们从不同角度来更好地优化神经网络。要想更好地优化神经网络，首先，要理解为什么优化会失败，收敛在局部极限值与鞍点会导致优化失败。其次，可以对学习率进行调整，使用自适应学习率和学习率调度。最后，批量归一化可以改变误差表面，这对优化也有帮助。

3.1 局部极小值与鞍点

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）、线性模型和浅层网络（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？

3.1.1 临界点及其种类

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 3.1 所示。图 3.1 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，损失不再下降了。

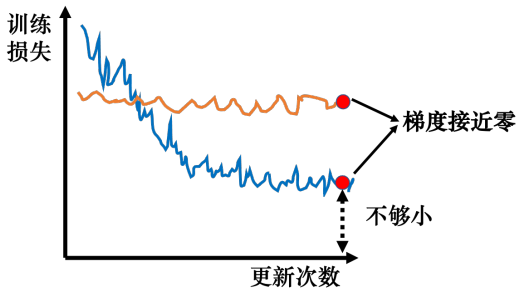
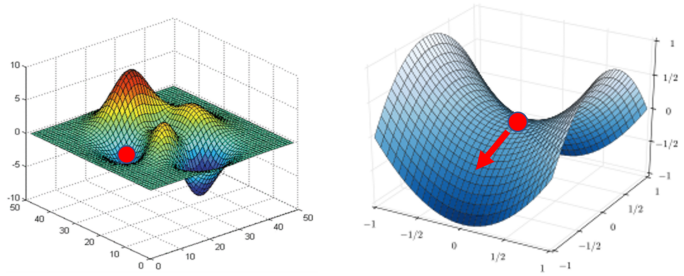


图 3.1 梯度下降失效的情况

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 3.2a 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 3.2b 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。

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。



(a) 局部极小值 (b) 鞍点

图 3.2 局部极小值与鞍点

3.1.2 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。虽然无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 $\boldsymbol{\theta}'$，在 $\boldsymbol{\theta}'$ 附近的损失函数是有办法写出来的——虽然 $L(\boldsymbol{\theta})$ 完整的样子写不出来。$\boldsymbol{\theta}'$ 附近的 $L(\boldsymbol{\theta})$ 可近似为

$$L(\boldsymbol{\theta}) \approx L\left(\boldsymbol{\theta}'\right) + \left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{g} + \frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right). \tag{3.1}$$

式 (3.1) 是泰勒级数近似（Tayler series appoximation）。其中，第一项 $L(\boldsymbol{\theta})'$ 告诉我们，当 $\boldsymbol{\theta}$ 跟 $\boldsymbol{\theta}'$ 很近的时候，$L(\boldsymbol{\theta})$ 应该跟 $L(\boldsymbol{\theta}')$ 还蛮靠近的；第二项 $\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{g}$ 中，$\boldsymbol{g}$ 代表梯度，它是一个向量，可以弥补 $L(\boldsymbol{\theta}')$ 跟 $L(\boldsymbol{\theta})$ 之间的差距。有时候梯度 $\boldsymbol{g}$ 会写成 $\nabla L(\boldsymbol{\theta}')$。$g_i$ 是向量 $\boldsymbol{g}$ 的第 i 个元素，就是 L 关于 $\boldsymbol{\theta}$ 的第 i 个元素的微分，即

$$g_i = \frac{\partial L\left(\boldsymbol{\theta}'\right)}{\partial \theta_i}. \tag{3.2}$$

光看 $\boldsymbol{g}$ 还是没有办法完整地描述 $L(\boldsymbol{\theta})$，还要看式 (3.1) 的第三项 $\frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)$。第三项跟**海森矩阵（Hessian matrix）** $\boldsymbol{H}$ 有关。$\boldsymbol{H}$ 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 $\boldsymbol{\theta}$ 的第 i 个元素对 $L\left(\boldsymbol{\theta}'\right)$ 作微分，再把 $\boldsymbol{\theta}$ 的第 j 个元素对 $\frac{\partial L\left(\boldsymbol{\theta}'\right)}{\partial \theta_i}$ 作微分后的结果，即

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L\left(\boldsymbol{\theta}'\right). \tag{3.3}$$

总结一下，损失函数 $L(\boldsymbol{\theta})$ 在 $\boldsymbol{\theta}'$ 附近可近似为式 (3.1)，式 (3.1) 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 $\boldsymbol{g}$ 为零，因此 $\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{g}$ 为零。所以在临界点的附近，损失函数可被近似为

$$L(\boldsymbol{\theta}) \approx L\left(\boldsymbol{\theta}'\right) + \frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right); \tag{3.4}$$

我们可以根据 $\frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)$ 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 $\boldsymbol{v}$ 来表示 $\boldsymbol{\theta}-\boldsymbol{\theta}'$，$\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}} \boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)$ 可改写为 $\boldsymbol{v}^{\mathrm{T}} \boldsymbol{H} \boldsymbol{v}$，有如下三种情况。

（1）如果对所有 $\boldsymbol{v}$，$\boldsymbol{v}^{\mathrm{T}} \boldsymbol{H} \boldsymbol{v} > 0$. 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 $\boldsymbol{v}$，$\boldsymbol{v}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{v}<0$. 这意味着对任意 $\boldsymbol{\theta}$，$L(\boldsymbol{\theta})<L(\boldsymbol{\theta}')$，$\boldsymbol{\theta}'$ 是附近最高的一个点，$L(\boldsymbol{\theta}')$ 是局部极大值。

（3）如果对于 $\boldsymbol{v}$，$\boldsymbol{v}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{v}$ 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 $\frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}}\boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)$ 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 $\boldsymbol{v}$ 都拿来试试看，所以有一个更简便的方法来判断 $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{v}$ 的正负。算出一个海森矩阵后，不需要把它跟所有的 $\boldsymbol{v}$ 都乘乘看，只要看 $\boldsymbol{H}$ 的特征值。若 $\boldsymbol{H}$ 的所有特征值都是正的，$\boldsymbol{H}$ 为正定矩阵，则 $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{v}>0$，临界点是局部极小值。若 $\boldsymbol{H}$ 的所有特征值都是负的，$\boldsymbol{H}$ 为负定矩阵，则 $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{v}<0$，临界点是局部极大值。若 $\boldsymbol{H}$ 的特征值有正有负，临界点是鞍点。

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

$$y=w_1w_2x. \tag{3.5}$$

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 3.3 所示。

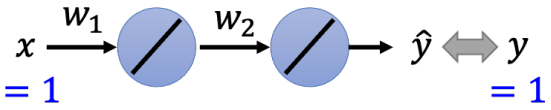


图 3.3　简单的神经网络

可以直接画出这个神经网络的误差表面，如图 3.4 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。

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 3.3 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 $\hat{y}=w_1w_2x$ 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

$$L=(y-w_1w_2x)^2=(1-w_1w_2)^2. \tag{3.6}$$

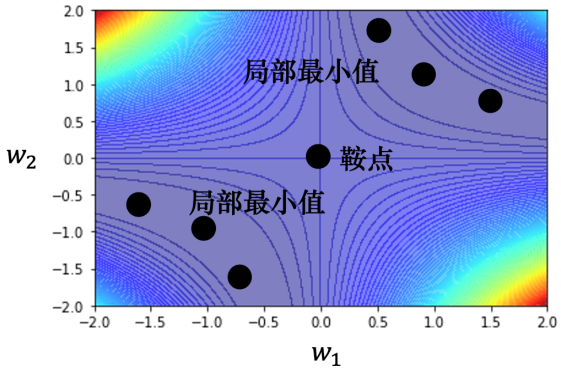


图 3.4 误差表面

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

$$\begin{cases}\frac{\partial L}{\partial w_1} &= 2\left(1-w_1w_2\right)\left(-w_2\right);\\ \frac{\partial L}{\partial w_2} &= 2\left(1-w_1w_2\right)\left(-w_1\right).\end{cases} \tag{3.7}$$

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 $\boldsymbol{H}$ 收集了 L 的二次微分：

$$\begin{cases}H_{1,1} &= \frac{\partial^2 L}{\partial w_1^2}=2\left(-w_2\right)\left(-w_2\right);\\ H_{1,2} &= \frac{\partial^2 L}{\partial w_1\partial w_2}=-2+4w_1w_2;\\ H_{2,1} &= \frac{\partial^2 L}{\partial w_2\partial w_1}=-2+4w_1w_2;\\ H_{2,2} &= \frac{\partial^2 L}{\partial w_2^2}=2\left(-w_1\right)\left(-w_1\right).\end{cases} \tag{3.8}$$

对于原点，只要把 $w_1=0, w_2=0$ 代进去，有海森矩阵

$$\boldsymbol{H}=\begin{bmatrix}0 & -2\\ -2 & 0\end{bmatrix}. \tag{3.9}$$

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。$\boldsymbol{H}$ 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 $\boldsymbol{g}$，但是我们走到某个地方以后发现 $\boldsymbol{g}$ 变成 $\boldsymbol{0}$ 了，就不能再看 $\boldsymbol{g}$ 了，$\boldsymbol{g}$ 不见了。但如果临界点是一个鞍点，还可以再看 $\boldsymbol{H}$，怎么再看 $\boldsymbol{H}$ 呢，$\boldsymbol{H}$ 怎么告诉我们怎么更新参数呢？

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

$$\boldsymbol{u}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{u}=\boldsymbol{u}^{\mathrm{T}}(\lambda\boldsymbol{u})=\lambda\|\boldsymbol{u}\|^2. \tag{3.10}$$

若 $\lambda<0$，则 $\lambda\|\boldsymbol{u}\|^2<0$。所以 $\frac{1}{2}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)^{\mathrm{T}}\boldsymbol{H}\left(\boldsymbol{\theta}-\boldsymbol{\theta}'\right)<0$。此时，$L(\boldsymbol{\theta})<L(\boldsymbol{\theta}')$，且

$$\boldsymbol{\theta}=\boldsymbol{\theta}'+\boldsymbol{u}. \tag{3.11}$$

沿着 $\boldsymbol{u}$ 的方向更新 $\boldsymbol{\theta}$，损失就会变小。因为根据式 (3.10) 和式 (3.11)，只要 $\boldsymbol{\theta} = \boldsymbol{\theta}' + \boldsymbol{u}$，沿着特征向量 $\boldsymbol{u}$ 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。

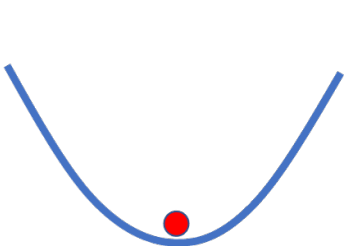
在前面的例子中，原点是一个临界点，此时的海森矩阵如式 (3.9) 所示，该海森矩阵有一个负的特征值：-2，特征值 -2 对应的特征向量有无穷多个。不妨取 $\boldsymbol{u} = [1, 1]^{\mathrm{T}}$，作为 -2 对应的特征向量。我们其实只要顺着 $\boldsymbol{u}$ 的方向去更新参数，就可以找到一个比鞍点的损失还要更低的点。以这个例子来看，原点是鞍点，其梯度为零，所以梯度不会告诉我们要怎么更新参数。但海森矩阵的特征向量告诉我们只要往 $[1, 1]^{\mathrm{T}}$ 的方向更新。损失就会变得更小，就可以逃离鞍点。

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，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。

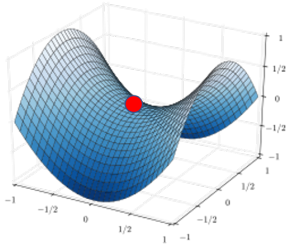
3.1.3 逃离鞍点的方法

讲到这边会有一个问题：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。

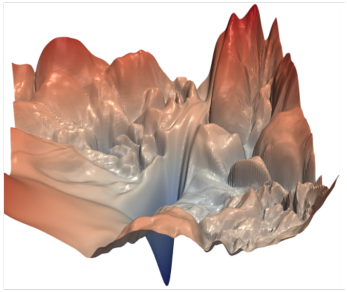
如图 3.5(a) 所示的一维空间中的误差表面，有一个局部极小值。但是在二维空间（如图 3.5(b) 所示），这个点就可能只是一个鞍点。常常会有人画类似图 3.5(c) 这样的图来告诉我们深度学习的训练是非常复杂的。如果我们移动某两个参数，误差表面的变化非常的复杂，有非常多局部极小值。低维度空间中的局部极小值点，在更高维的空间中，实际是鞍点。同样地，如果在二维的空间中没有路可以走，会不会在更高维的空间中，其实有路可以走？更高的维度难以视化它，但我们在训练一个网络的时候，参数数量动辄达百万千万级，所以误差表面其实有非常高的维度—— 参数的数量代表了误差表面的维度。既然维度这么高，会不会其实就有非常多的路可以走呢？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 3.6 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 3.6 中，横轴代表最小值比例（minimum ratio），最小



(a) 一维误差表面



(b) 二维误差表面



(c) 复杂误差表面

图 3.5　误差表面

值比例定义为

$$\text{最小值比例} = \frac{\text{正特征值数量}}{\text{总特征值数量}}. \tag{3.12}$$

实际上，我们几乎找不到所有特征值都为正的临界点。在图 3.6 所示的例子中，最小值比例最大也不过处于 0.5 ~ 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，往往只是遇到了鞍点。

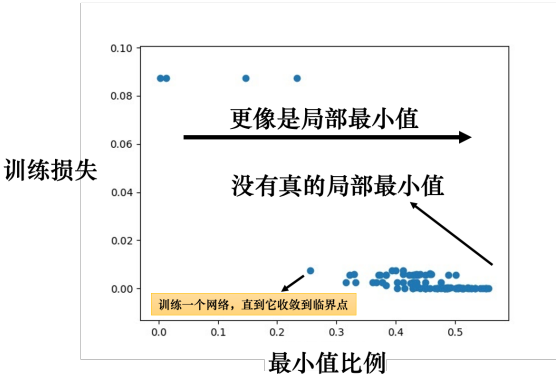


图 3.6　训练不同神经网络的结果

3.2 批量和动量

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 3.7 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重新划分批量，也就是说，每个回合的批量的数据都不一样。

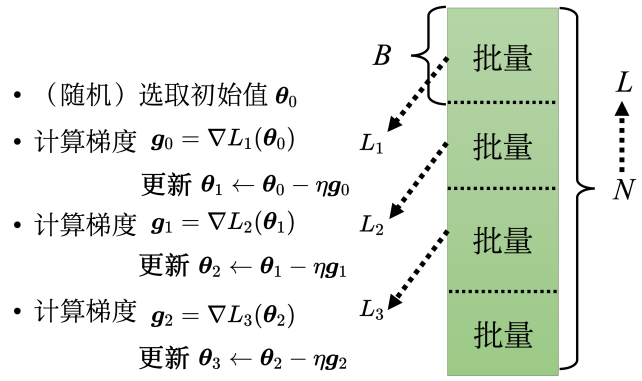


图 3.7 使用批量优化

3.2.1 批量大小对梯度下降法的影响

假设现在我们有 20 笔训练数据，先看下两个最极端的情况，如图 3.8 所示。

- 图 3.8 （a）的情况是没有用批量，批量大小为训练数据的大小，这种使用全批量（full batch)的数据来更新参数的方法即**批量梯度下降法(Batch Gradient Descent，BGD)**。此时模型必须把 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 3.8（b）中，批量大小等于 1，此时使用的方法即**随机梯度下降法 (Stochastic Gradient Descent，SGD)**，也称为增量梯度下降法。批量大小等于 1 意味着只要取出一笔数据即可计算损失、更新一次参数。如果总共有 20 笔数据，那么在每一个回合里面，参数会更新 20 次。用一笔数据算出来的损失相对带有更多噪声，因此其更新的方向如图 3.8 所示，是曲曲折折的 。

实际上，批量梯度下降并没有“划分批量”：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。但相比随机梯度下降，批量梯度下降每次更新更稳定、更准确。

> 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，批量梯度下降花费的时间不一定更长；对于比较大的批量，计算损失和梯度花费的时间不一定比使用小批量的计算时间长 。使用 Tesla V100 GPU 在 MNIST 数据集得到的实验结果如图 3.9 所示。图 3.9 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

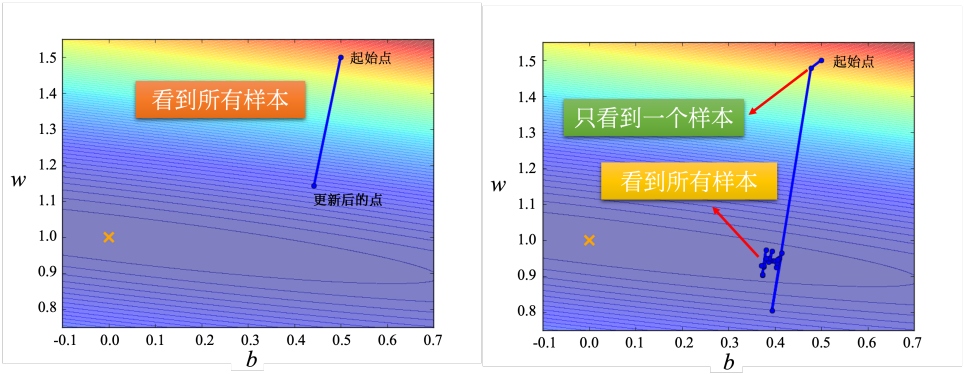


图 3.8　批量梯度下降法与随机梯度下降法

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 28×28 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”。

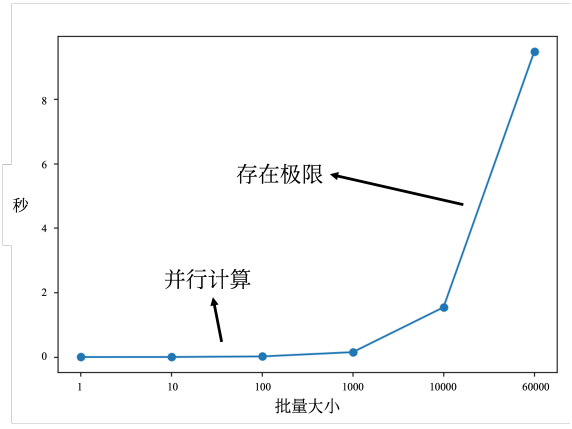
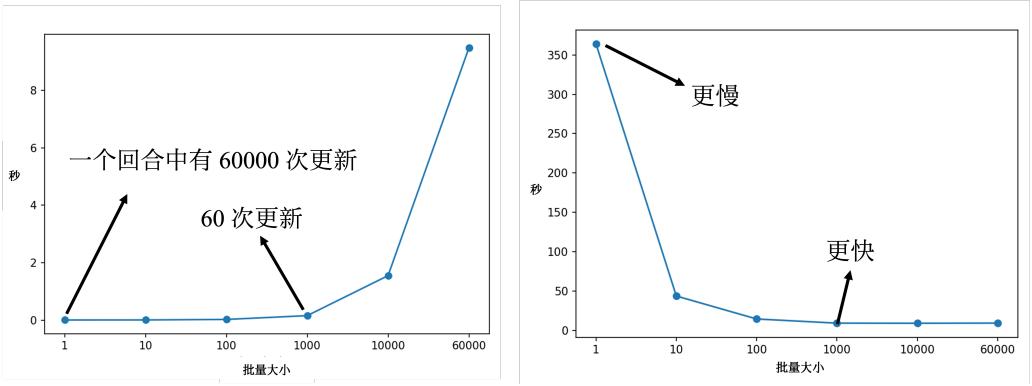


图 3.9　批量大小与计算时间的关系

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

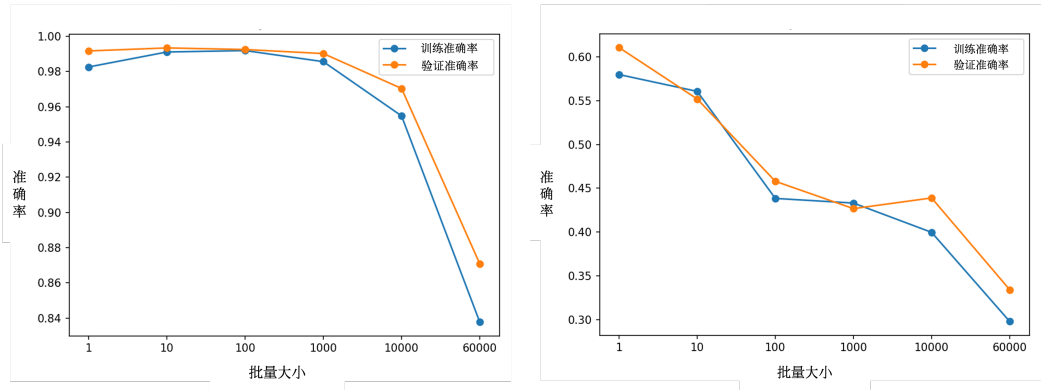
60000 次更新跟 60 次更新比起来，其时间的差距量就非常大了。图 3.10(a) 是用一个批量计算梯度并更新一次参数所需的时间。假设批量大小为 1，“跑”完一个回合，要更新 60000 次参数，其时间是非常大的。但假设批量大小是 1000，更新 60 次参数就会“跑”完一个回合。图 3.10(b) 是“跑”完一个完整的回合需要花的时间。如果批量大小为 1000 或 60000，其时间比批量大小设 1 还要短 。图 3.10(a) 和图 3.10(b) 的趋势正好是相反的。因此实际上，在有考虑并行计算的时候，大的批量大小反而是较有效率的，一个回合大的批量花的时间反而是比较少的。



(a) 1 次更新的时间　　(b) 1 个回合的时间

图 3.10　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 3.11 所示，横轴是批量大小，纵轴是正确率。图 3.11(a) 是 MNIST 数据集上的结果，图 3.11(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。



(a) MNIST　　(b) CIFAR-10

图 3.11　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 3.12 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，如果不看海森矩阵，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。

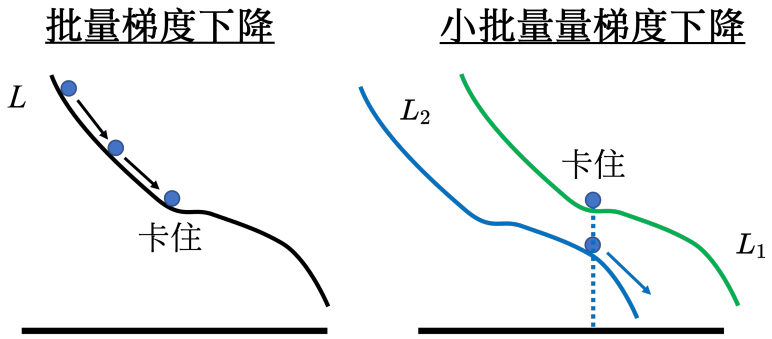


图 3.12　小批量梯度下降更好的原因

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[1]。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。大的批量跟小的批量的训练**准确率（accuracy）**差不多，但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。

这篇论文给出了一个解释，如图 3.13 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，训练跟测试采样到的数据可能不一样，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。但对在右边在“峡谷”里面的最小值，一差就可以天差地远 。虽然它在训练集上的损失很低，但训练跟测试之间的损失函数不一样，因此测试时，损失函数一变，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 3.1 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

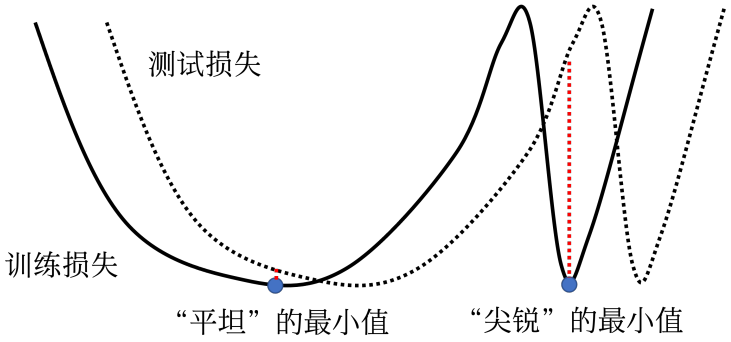


图 3.13 小批量优化容易跳出局部最小值的原因

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[2-3]。比如 76 分钟训练 BERT[4]，15 分钟训练 ResNet[5]，一分钟训练 ImageNet[6] 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 3.1 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

3.2.2 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 3.14 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。

一般的梯度下降（vanilla gradient descent）如图 3.15 所示。初始参数为 $\boldsymbol{\theta}_0$，计算一下梯度，计算完梯度后，往梯度的反方向去更新参数 $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 - \eta \boldsymbol{g}_0$。有了新的参数 $\boldsymbol{\theta}_1$ 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。

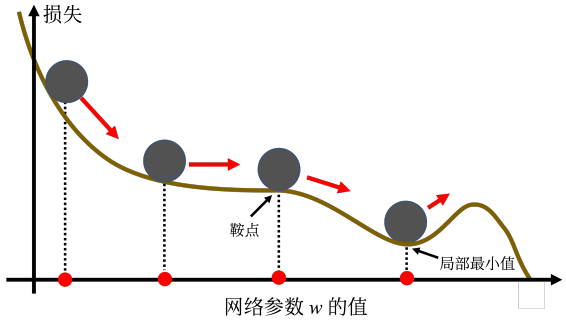


图 3.14 物理世界中的惯性

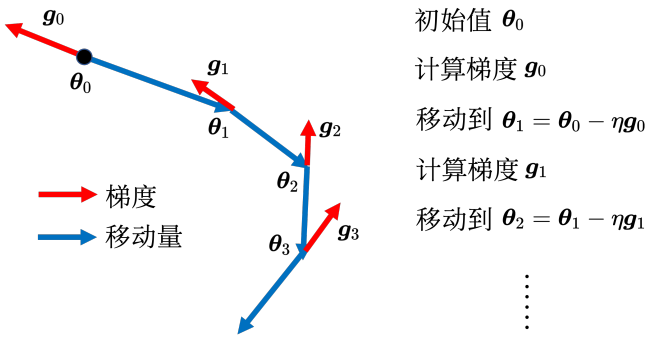


图 3.15 一般梯度下降

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 3.16 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 3.16 所示，初始的参数值为 $\boldsymbol{\theta}_0 = 0$，前一步的参数的更新量为 $\boldsymbol{m}_0 = 0$。接下来在 $\boldsymbol{\theta}_0$ 的地方，计算梯度的方向 $\boldsymbol{g}_0$。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 $\boldsymbol{g}_1$，接下来更新的方向为 $\boldsymbol{m}_2 = \lambda \boldsymbol{m}_1 - \eta \boldsymbol{g}_1$，参数更新为 $\boldsymbol{\theta}_2$ ，接下来就反复进行同样的过程。

每一步的移动都用 $\boldsymbol{m}$ 来表示。$\boldsymbol{m}$ 其实可以写成之前所有计算的梯度的加权和，如式 (3.13) 所示。其中 η 是学习率，λ 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。

$$\begin{aligned} \boldsymbol{m}_0 &= 0 \\ \boldsymbol{m}_1 &= -\eta \boldsymbol{g}_0 \\ \boldsymbol{m}_2 &= -\lambda \eta \boldsymbol{g}_0 - \eta \boldsymbol{g}_1 \\ &\vdots \end{aligned} \tag{3.13}$$

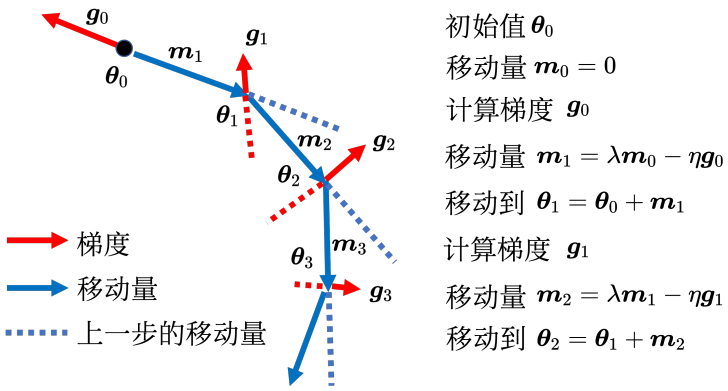


图 3.16　动量法

动量的简单例子如图 3.17 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 。

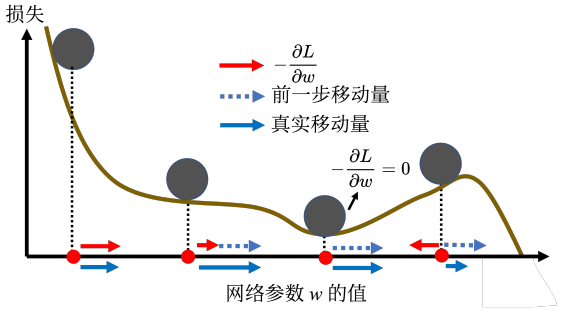


图 3.17　动量的好处

3.3　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 3.18 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 3.19 给出了示例。图 3.19 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 3.20 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

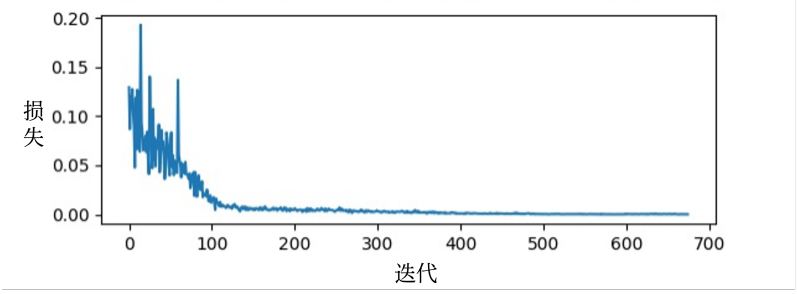


图 3.18　训练网络时损失变化

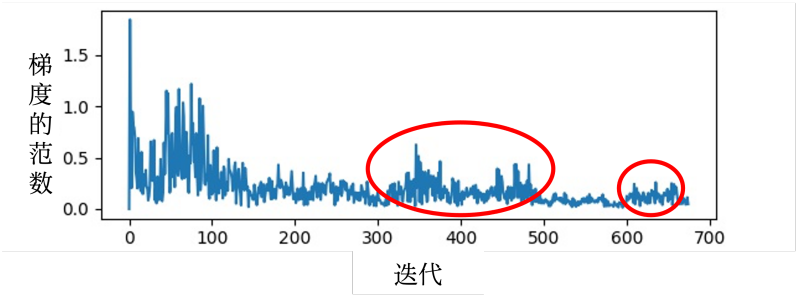


图 3.19　训练网络时梯度范数变化

在局部最小值或鞍点，只是单纯的损失无法再下降。

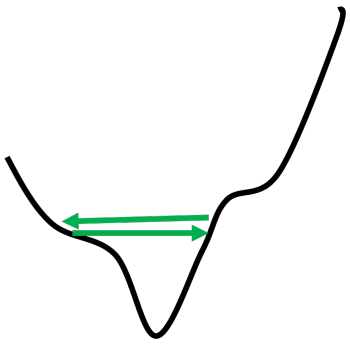


图 3.20　梯度来回“震荡”

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。一般的梯度下降，其实是做不到的。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。

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 3.21 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 3.22(a) 所示。参数在峡谷的两端，参数在山壁的两端不断

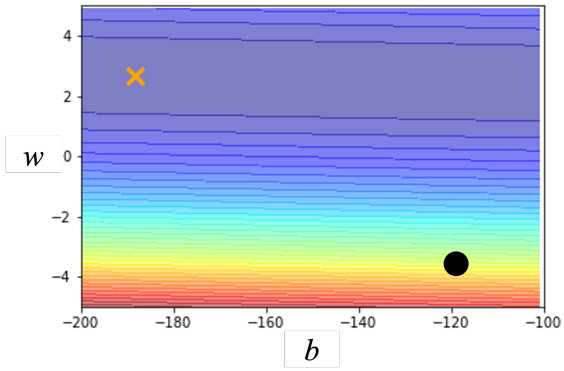
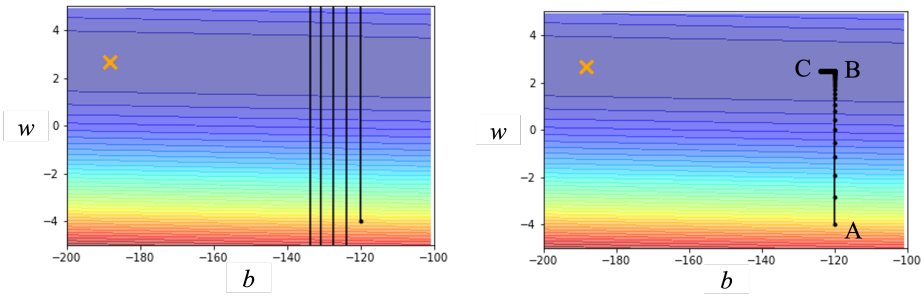


图 3.21　凸误差表面

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 3.22(b) 所示，参数不再“震荡”了。参数会滑到山谷底后左转，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。



(a) 学习率为 η^{-2}　　(b) 学习率为 η^{-7}

图 3.22　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 3.23 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。

3.3.1　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。

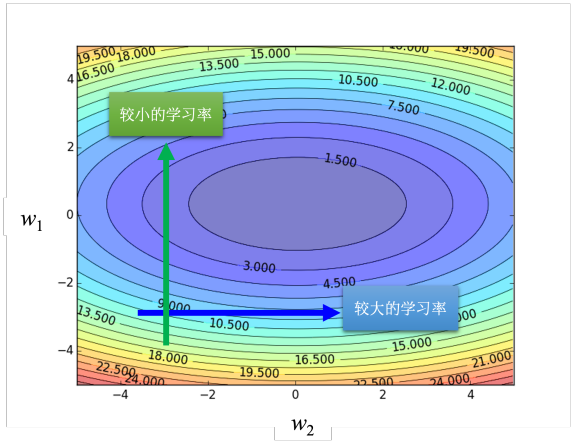


图 3.23 不同参数需要不同的学习率

梯度下降更新某个参数 $\boldsymbol{\theta}_t^i$ 的过程为

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \eta \boldsymbol{g}_t^i \tag{3.14}$$

$\boldsymbol{\theta}_t^i$ 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

$$\boldsymbol{g}_t^i = \left.\frac{\partial L}{\partial \boldsymbol{\theta}^i}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \tag{3.15}$$

$\boldsymbol{g}_t^i$ 代表在第 t 个迭代，即 $\boldsymbol{\theta} = \boldsymbol{\theta}_t$ 时，参数 $\boldsymbol{\theta}^i$ 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 η 变成 $\frac{\eta}{\sigma_t^i}$

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \frac{\eta}{\sigma_t^i}\boldsymbol{g}_t^i \tag{3.16}$$

σ_t^i 的上标为 i，这代表参数 σ 与 i 相关，不同的参数的 σ 不同。σ_t^i 的下标为 t，这代表参数 σ 与迭代相关，不同的迭代也会有不同的 σ。学习率从 η 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

$$\boldsymbol{\theta}_1^i \leftarrow \boldsymbol{\theta}_0^i - \frac{\eta}{\sigma_0^i}\boldsymbol{g}_0^i \tag{3.17}$$

其中 $\boldsymbol{\theta}_0^i$ 是初始化参数。而 σ_0^i 的计算过程为

$$\sigma_0^i = \sqrt{\left(\boldsymbol{g}_0^i\right)^2} = \left|\boldsymbol{g}_0^i\right| \tag{3.18}$$

其中 $\boldsymbol{g}_0^i$ 是梯度。将 σ_0^i 的值代入更新的公式可知 $\frac{\boldsymbol{g}_0^i}{\sigma_0^i}$ 的值是 $+1$ 或 -1。第一次在更新参数，从 $\boldsymbol{\theta}_0^i$ 更新到 $\boldsymbol{\theta}_1^i$ 的时候，要么是加上 η，要么是减掉 η，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

$$\boldsymbol{\theta}_2^i \leftarrow \boldsymbol{\theta}_1^i - \frac{\eta}{\sigma_1^i}\boldsymbol{g}_1^i \tag{3.19}$$

其中 σ_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 (3.20) 所示。

$$\sigma_1^i = \sqrt{\frac{1}{2}\left[\left(\boldsymbol{g}_0^i\right)^2 + \left(\boldsymbol{g}_1^i\right)^2\right]} \tag{3.20}$$

同样的操作反复继续下去，如式 (3.21) 所示。

$$\begin{aligned} \boldsymbol{\theta}_3^i &\leftarrow \boldsymbol{\theta}_2^i - \frac{\eta}{\sigma_2^i}\boldsymbol{g}_2^i \quad \sigma_2^i = \sqrt{\frac{1}{3}\left[\left(\boldsymbol{g}_0^i\right)^2 + \left(\boldsymbol{g}_1^i\right)^2 + \left(\boldsymbol{g}_2^i\right)^2\right]} \\ &\vdots \end{aligned} \tag{3.21}$$

第 $t+1$ 次更新参数的时候，即

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \frac{\eta}{\sigma_t^i}\boldsymbol{g}_t^i \quad \sigma_t^i = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}\left(\boldsymbol{g}_t^i\right)^2} \tag{3.22}$$

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

图 3.24 中有两个参数：$\boldsymbol{\theta}^1$ 和 $\boldsymbol{\theta}^2$。$\boldsymbol{\theta}^1$ 坡度小，$\boldsymbol{\theta}^2$ 坡度大。因为 $\boldsymbol{\theta}^1$ 坡度小，根据式 (3.22)，$\boldsymbol{\theta}_1^i$ 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 σ_t^i 就小，σ_t^i 小学习率就大。反过来，$\boldsymbol{\theta}^1$ 坡度大，所以计算出的梯度都比较大，σ_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 σ_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。

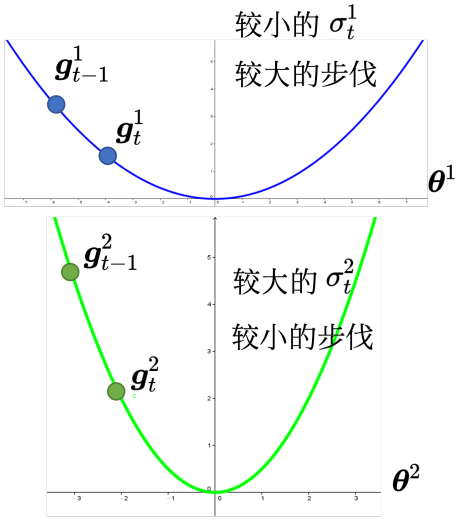


图 3.24　自动调整学习率示例

3.3.2　RMSProp

同一个参数需要的学习率，也会随着时间而改变。在图 3.25 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（Root Mean Squared propagation）**。

RMSprop 没有论文，Geoffrey 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 Adagrad 的方法是相同的，即

$$\sigma_0^i = \sqrt{\left(g_0^i\right)^2} = \left|\boldsymbol{g}_0^i\right| \tag{3.23}$$

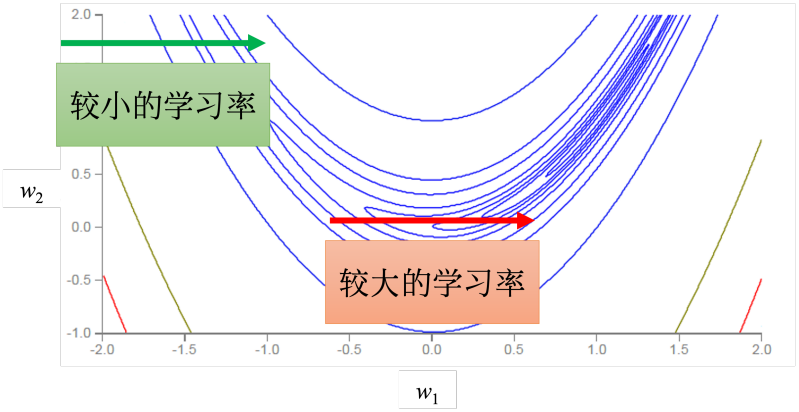


图 3.25 AdaGrad 的问题

第二步更新过程为

$$\boldsymbol{\theta}_2^i \leftarrow \boldsymbol{\theta}_1^i - \frac{\eta}{\sigma_1^i}\boldsymbol{g}_1^i \quad \sigma_1^i = \sqrt{\alpha\left(\sigma_0^i\right)^2 + (1-\alpha)\left(\boldsymbol{g}_1^i\right)^2} \tag{3.24}$$

其中 $0 < \alpha < 1$，其是一个可以调整的超参数。计算 $\boldsymbol{\theta}_1^i$ 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 α 设很小趋近于 0，代表 $\boldsymbol{g}_1^i$ 相较于之前算出来的梯度而言，比较重要；如果 α 设很大趋近于 1，代表 $\boldsymbol{g}_1^i$ 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 (3.25) 所示。

$$\begin{aligned}
\boldsymbol{\theta}_3^i &\leftarrow \boldsymbol{\theta}_2^i - \frac{\eta}{\sigma_2^i}\boldsymbol{g}_2^i \quad \sigma_2^i = \sqrt{\alpha\left(\sigma_1^i\right)^2 + (1-\alpha)\left(\boldsymbol{g}_2^i\right)^2} \\
&\vdots \\
\boldsymbol{\theta}_{t+1}^i &\leftarrow \boldsymbol{\theta}_t^i - \frac{\eta}{\sigma_t^i}\boldsymbol{g}_t^i \quad \sigma_t^i = \sqrt{\alpha\left(\sigma_{t-1}^i\right)^2 + (1-\alpha)\left(\boldsymbol{g}_t^i\right)^2}
\end{aligned} \tag{3.25}$$

RMSProp 通过 α 可以决定，$\boldsymbol{g}_t^i$ 相较于之前存在 σ_{t-1}^i 里面的 $\boldsymbol{g}_1^i, \boldsymbol{g}_2^i, \cdots\cdots, \boldsymbol{g}_{t-1}^i$ 的重要性有多大。如果使用 RMSprop，就可以动态调整 σ_t^i 这一项。图 3.26 中黑线是误差表面，球就从 A 走到 B，AB 段的路很平坦，$\boldsymbol{g}$ 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 α 设小一点，让新的、刚看到的梯度的影响比较大，很快地让 σ_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 α，让其比较看重最近算出来的梯度，梯度一变小，σ_t^i 的值就变小了，走的步伐就变大了。

3.3.3 Adam

最常用的优化的策略或者**优化器（optimizer）**是**Adam（Adaptive moment estimation）**[7]。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。

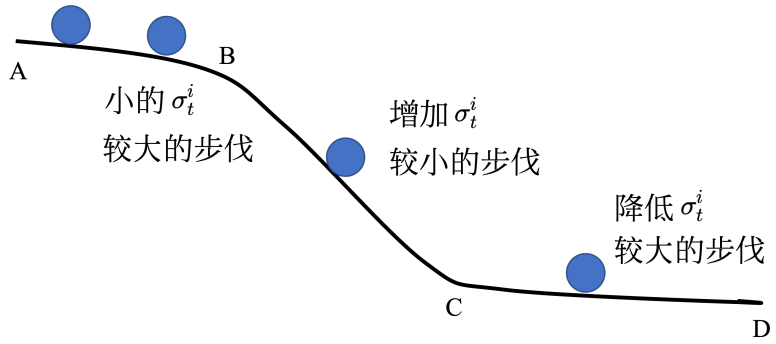


图 3.26 RMSprop 示例

3.4 学习率调度

如图 3.22 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 3.27 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 3.27 中红圈的地方，快走到终点的时候突然“爆炸”了。σ_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 σ_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，σ_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。

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \frac{\eta}{\sigma_t^i}\boldsymbol{g}_t^i$$

$$\sigma_t^i = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}\left(\boldsymbol{g}_t^i\right)^2}$$

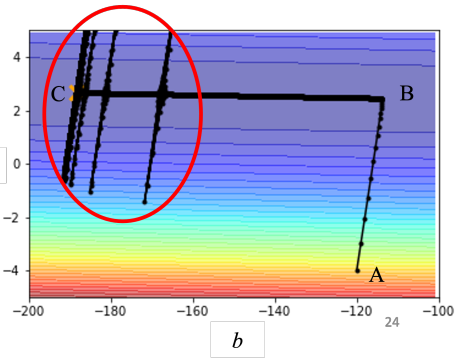


图 3.27 AdaGrad 优化的问题

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 η 是一个固定的值，而在学习率调度中 η 跟时间有关，如式 (3.26) 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 η 越来越小，如图 3.28 所示。图 3.22b 的情况，如果加上学习率下降，可以很平顺地走到终点，如图 3.29 所示。在图 3.22b 红圈的地方，虽然步伐很大，但 η 变得非常小，步伐乘上 η 就变小了，就可以慢慢地走到终点。

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \frac{\eta_t}{\sigma_t^i}\boldsymbol{g}_t^i \tag{3.26}$$

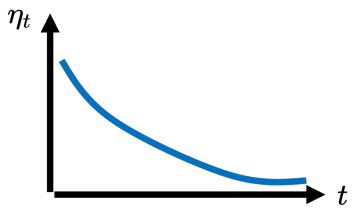


图 3.28 学习率衰减

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。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。残差网络[8] 里面是有预热的，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。

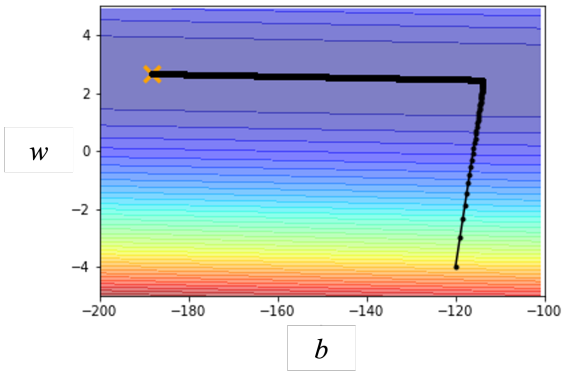


图 3.29 学习率衰减的优化效果

Q：为什么需要预热?
A：当我们使用 Adam、RMSprop 或 AdaGrad 时，需要计算 σ。而 σ 是一个统计的结果。从 σ 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 σ 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 σ 的统计数据，等 σ 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[9]。

3.5 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 (3.27) 所示。

$$\boldsymbol{\theta}_{t+1}^i \leftarrow \boldsymbol{\theta}_t^i - \frac{\eta_t}{\sigma_t^i}\boldsymbol{m}_t^i \tag{3.27}$$

其中 $\boldsymbol{m}_t^i$ 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 $\frac{\boldsymbol{m}_t^i}{\sigma_t^i}$ 。最后通过 η_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 $\boldsymbol{m}_t^i$ 或 σ_t^i，或者是使用不同的学习率调度的方式。

Q：动量 $\boldsymbol{m}_t^i$ 考虑了过去所有的梯度，均方根 σ_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：$\boldsymbol{m}_t^i$ 和 σ_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 σ_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 σ_t^i 计算出来的结果并不会互相抵消。

3.6 分类

分类与回归是深度学习最常见的两种问题，第一章的观看次数预测属于回归问题，本节将介绍分类问题。

3.6.1 分类与回归的关系

回归是输入一个向量 $\boldsymbol{x}$，输出 $\hat{y}$，我们希望 $\hat{y}$ 跟某一个标签 y 越接近越好，y 是要学习的目标。而分类可当作回归来看，输入 $\boldsymbol{x}$ 后，输出仍然是一个标量 $\hat{y}$，要让它跟正确答案的那个类越接近越好。$\hat{y}$ 是一个数字，我们可以把类也变成数字。如图 3.30 所示，类 1 是编号 1，类 2 是编号 2，类 3 是编号 3，$\hat{y}$ 跟类的编号越接近越好。但该方法在某些状况下会有问题，假设类 1、2、3 有某种关系。比如根据一个人的身高跟体重，预测他的年级，一年级、二年级还是三年级。一年级跟二年级关系比较近，一年级跟三年级关系比较远。用数字来表示类会预设 1 和 2 有比较近的关系，1 和 3 有比较远的关系。但假设三个类本身没有特定的关系，类 1 是 1，类 2 是 2 类 3 是 3。这种情况，需要引入独热向量来表示类。实际上，在做分类的问题的时候，比较常见的做法也是用独热向量表示类。

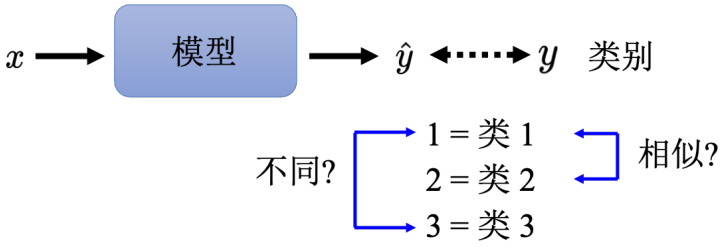


图 3.30 用数字表示类的问题

如果有三个类，标签 y 就是一个三维的向量，比如类 1 是 $[1,0,0]^{\mathrm{T}}$，类 2 是 $[0,1,0]^{\mathrm{T}}$，类 3 是 $[0,0,1]^{\mathrm{T}}$。如果每个类都用一个独热向量来表示，就没有类 1 跟类 2 比较接近，类 1 跟类 3 比较远的问题。如果用独热向量计算距离的话，类两两之间的距离都是一样的。

如果目标 y 是一个向量，比如 y 是有三个元素的向量，网络也要输出三个数字才行。如图 3.31 所示，输出三个数值就是把本来输出一个数值的方法，重复三次。把 a_1、a_2 和 a_3 乘

上三个不同的权重，加上偏置，得到 $\hat{y}_1$；再把 a_1、a_2 和 a_3 乘上另外三个权重，再加上另外一个偏置得到 $\hat{y}_2$；把 a_1、a_2 和 a_3 再乘上另外一组权重，再加上另外一个偏置得到 $\hat{y}_3$。输入一个特征向量，产生 $\hat{y}_1$、$\hat{y}_2$、$\hat{y}_3$，希望 $\hat{y}_1$、$\hat{y}_2$、$\hat{y}_3$ 跟目标越接近越好。

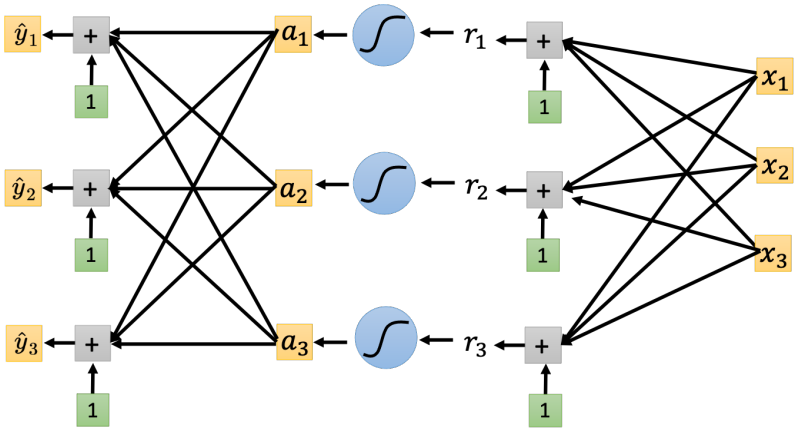


图 3.31　网络多个输出示例

3.6.2　带有 softmax 的分类

按照上述的设定，分类实际过程是：输入 x，乘上 W，加上 b，通过激活函数 σ，乘上 W'，再加上 b' 得到向量 $\hat{\boldsymbol{y}}$。但实际做分类的时候，往往会把 $\hat{y}$ 通过 softmax 函数得到 y'，才去计算 y' 跟 $\hat{y}$ 之间的距离。

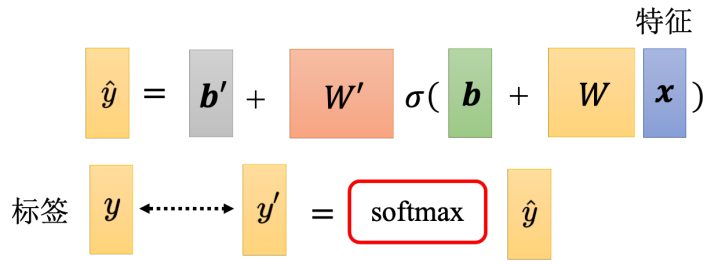


图 3.32　带有 softmax 的分类

> Q：为什么分类过程中要加上 softmax 函数？
> A：一个比较简单的解释是，y 是独热向量，所以其里面的值只有 0 跟 1，但是 $\hat{y}$ 里面有任何值。既然目标只有 0 跟 1，但 $\hat{y}$ 有任何值，可以先把它归一化到 0 到 1 之间，这样才能跟标签的计算相似度。

softmax 的计算如式 (3.28) 所示，先把所有的 y 取一个指数（负数取指数后也会变成正的），再对其做归一化（除掉所有 y 的指数值的和）得到 y'。图 3.33 是 softmax 的块（block），输入 y_1、y_2 和 y_3，产生 y'_1、y'_2 和 y'_3。比如 $y_1 = 3$，$y_2 = 1$，$y_3 = -3$，取完指数的时候，$\exp(3) = 20$、$\exp(1) = 2.7$ 和 $\exp(-3) = 0.05$，做完归一化后，就变成 0.88、0.12 跟 0。-3

取完指数，再做归一化以后，会变成趋近于 0 的值。所以 softmax 除了归一化，让 y_1'、y_2' 和 y_3'，变成 0 到 1 之间，和为 1 以外，它还会让大的值跟小的值的差距更大。

$$y_i' = \frac{\exp(y_i)}{\sum_j \exp(y_i)} \tag{3.28}$$

其中，$1 > y_i' > 0$，$\sum_i y_i' = 1$。

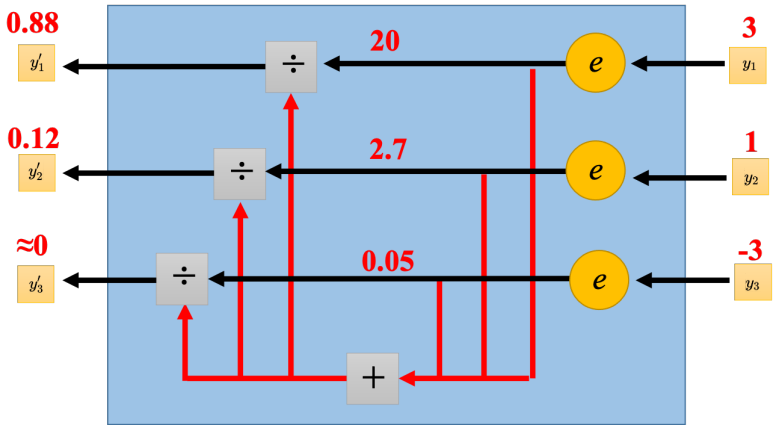


图 3.33　softmax 示例

图 3.33 考虑了三个类的状况，两个类也可以直接套 softmax 函数。但一般有两个类的时候，我们不套 softmax，而是直接取 sigmoid。当只有两个类的时候，sigmoid 和 softmax 是等价的。

3.6.3　分类损失

当我们把 x 输入到一个网络里面产生 $\hat{y}$ 后，通过 softmax 得到 y'，再去计算 y' 跟 y 之间的距离 e，如图 3.34 所示。

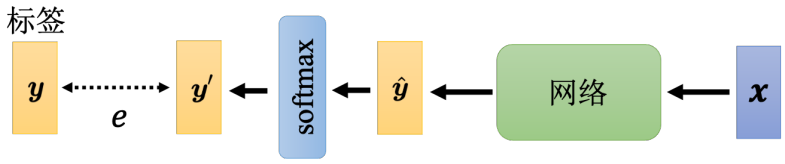


图 3.34　分类损失

计算 y' 跟 y 之间的距离不只一种做法，可以是如式 (3.29) 所示的均方误差，即把 y 里面每一个元素拿出来，计算它们的平方和当作误差。

$$e = \sum_i (y_i - y_i')^2 \tag{3.29}$$

但如式 (3.30) 所示的交叉熵更常用，当 $\hat{y}$ 跟 y' 相同时，可以最小化交叉熵的值，此时均方误差也是最小的。最小化交叉熵其实就是最大化似然（maximize likelihood）。

$$e = -\sum_i y_i \ln y_i' \tag{3.30}$$

接下来从优化的角度来说明相较于均方误差，交叉熵是被更常用在分类上。如图 3.35 所示，有一个三类的分类，网络先输出 y_1、y_2 和 y_3，在通过 softmax 以后，产生 y_1'、y_2' 和 y_3'。假设正确答案是 $[1,0,0]^{\mathrm{T}}$，要计算 $[1,0,0]^{\mathrm{T}}$ 跟 y_1'、y_2' 和 y_3' 之间的距离 e，e 可以是均方误差或交叉熵。假设 y_1 的变化是从-10 到 10，y_2 的变化也是从-10 到 10，y_3 就固定设成-1000。因为 y_3 的值很小，通过 softmax 以后，y_3' 非常趋近于 0，它跟正确答案非常接近，且它对结果影响很少。总之，我们假设 y_3 设一个定值，只看 y_1 跟 y_2 有变化的时候，对损失 e 的影响。

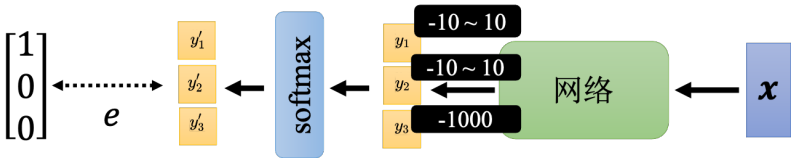
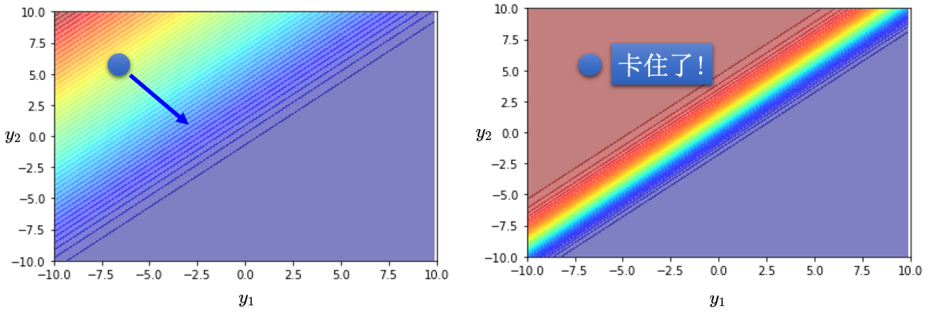


图 3.35　使用 softmax 的好处

图 3.36 是分别在 e 为均方误差和交叉熵时，y_1、y_2 的变化对损失的影响，对误差表面的影响，红色代表损失大，蓝色代表损失小。如果 y_1 很大，y_2 很小，代表 y_1' 会很接近 1，y_2' 会很接近 0。所以不管 e 取均方误差或交叉熵，如果 y_1 大、y_2 小，损失都是小的；如果 y_1 小，y_2 大，y_1' 是 0，y_2' 是 1，这个时候损失会比较大。

图 3.36 中左上角损失大，右下角损失小，所以期待最后在训练的时候，参数可以“走”到右下角的地方。假设参数优化开始的时候，对应的损失都是左上角。如果选择交叉熵，如图 3.36(a) 所示，左上角圆圈所在的点有斜率的，所以可以通过梯度，一路往右下的地方“走”；如果选均方误差，如图 3.36(b) 所示，左上角圆圈就卡住了，均方误差在这种损失很大的地方，它是非常平坦的，其梯度是非常小趋近于 0 的。如果初始时在圆圈的位置，离目标非常远，其梯度又很小，无法用梯度下降顺利地“走”到右下角。

因此做分类时，选均方误差的时候，如果没有好的优化器，有非常大的可能性会训练不起来。如果用 Adam，虽然图 3.36(b) 中圆圈的梯度很小，但 Adam 会自动调大学习率，还有机会走到右下角，不过训练的过程比较困难。总之，改变损失函数可以改变优化的难度。



(a) 交叉熵的损失　　(b) 均方误差的损失

图 3.36　均方误差、交叉熵优化对比

3.7　批量归一化

如果误差表面很崎岖，它比较难训练。能不能直接改误差表面的地貌，“把山铲平”，让它变得比较好训练呢？**批量归一化（Batch Normalization，BN）**就是其中一个“把山铲平”的

想法。不要小看优化这个问题，有时候就算误差表面是凸（convex）的，它就是一个碗的形状，都不一定很好训练。如图 3.37 所示，假设两个参数对损失的斜率差别非常大，在 w_1 这个方向上面，斜率变化很小，在 w_2 这个方向上面斜率变化很大。

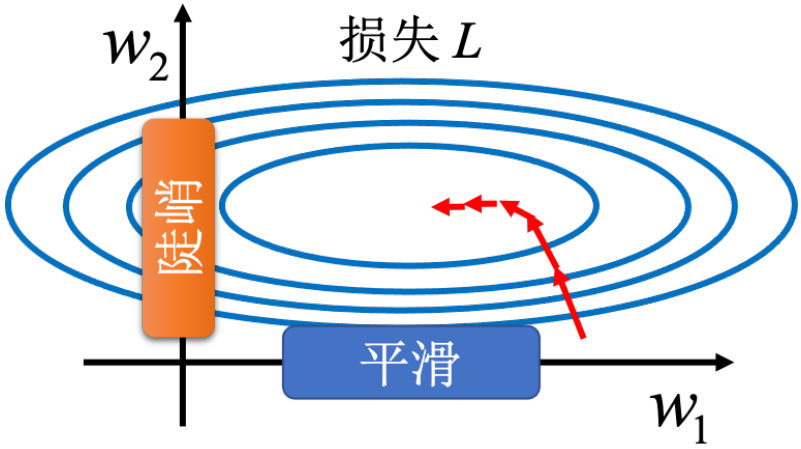


图 3.37　训练的问题

如果是固定的学习率，可能很难得到好的结果，所以我们才需要自适应的学习率、Adam 等比较进阶的优化的方法，才能够得到好的结果。从另外一个方向想，直接把难做的误差表面把它改掉，看能不能够改得好做一点。在做这件事之前，第一个要问的问题就是：w_1 跟 w_2 斜率差很多的这种状况，到底是从什么地方来的。

图 3.38 是一个非常简单的模型，其输入是 x_1 跟 x_2，对应的参数为 w_1 跟 w_2，它是一个线性的模型，没有激活函数。w_1 乘 x_1，w_2 乘 x_2 加上 b 以后就得到 $\hat{y}$，然后会计算 $\hat{y}$ 跟 y 之间的差距当做 e，把所有训练数据 e 加起来就是损失，然后去最小化损失。

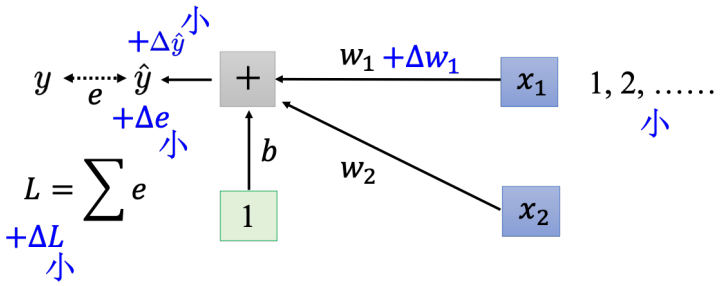


图 3.38　简单的线性模型

什么样的状况会产生像上面这样子，比较不好训练的误差表面呢？对 w_1 有一个小小的改变，比如加上 Δw_1 的时候，L 也会有一个改变，那这个 w_1 呢，是通过 w_1 改变的时候，就改变了 y，y 改变的时候就改变了 e，接下来就改变了 L。

什么时候 w_1 的改变会对 L 的影响很小呢，也就是它在误差表面上的斜率会很小呢？一

个可能性是当输入很小的时候，假设 x_1 的值在不同的训练样本里面，它的值都很小，那因为 x_1 是直接乘上 w_1，如果 x_1 的值都很小，w_1 有一个变化的时候，它得到的，它对 y 的影响也是小的，对 e 的影响也是小的，它对 L 的影响就会是小的。反之，如图 3.39 所示，如果是 x_2 的话，假设 x_2 的值都很大，当 w_2 有一个小小的变化的时候，虽然 w_2 这个变化可能很小，但是因为它乘上了 x_2，x_2 的值很大，那 y 的变化就很大，e 的变化就很大，L 的变化就会很大，就会导致我们在 w 这个方向上，做变化的时候，我们把 w 改变一点点，误差表面就会有很大的变化。所以既然在这个线性的的模型里面，当输入的特征，每一个维度的值，它的范围差距很大的时候，我们就可能产生像这样子的误差表面，就可能产生不同方向，斜率非常不同，坡度非常不同的误差表面所以怎么办呢，有没有可能给特征里面不同的维度，让它有同样的数值的范围。如果我们可以给不同的维度，同样的数值范围的话，那我们可能就可以制造比较好的误差表面，让训练变得比较容易一点其实有很多不同的方法，这些不同的方法往往就合起来统称为**特征归一化（feature normalization）**。

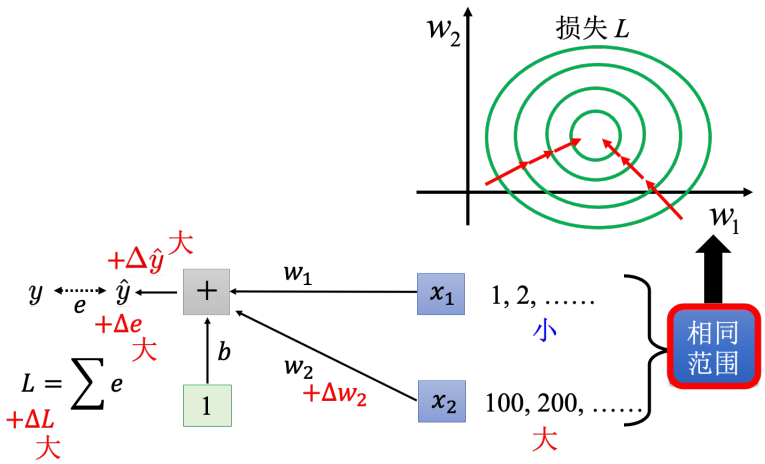


图 3.39　需要特征归一化的原因

以下所讲的方法只是特征归一化的一种可能性，即 Z 值归一化（Z-score normalization），也称为标准化（standardization）。它并不是特征归一化的全部，假设 x^1 到 x^R，是我们所有的训练数据的特征向量。我们把所有训练数据的特征向量，统统都集合起来。向量 x_1 里面就 x_1^1 代表 x_1 的第一个元素，x_1^2 代表 x_2 的第一个元素，以此类推。我们把不同笔数据即不同特征向量，同一个维度里面的数值，把它取出来，对于每个维度 i，计算其**平均值（mean）** m_i 和**标准差（standard deviation）** σ_i。接下来我们就可以做一种归一化。

$$\tilde{x}_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i} \tag{3.31}$$

我们就是把这边的某一个数值 x，减掉这一个维度算出来的平均值，再除掉这个维度，算出来的标准差，得到新的数值 $\tilde{x}$。得到新的数值以后，再把新的数值把它塞回去。

归一化有个好处，做完归一化以后，这个维度上面的数值就会平均是 0，其方差是 1，所以这一排数值的分布就都会在 0 上下；对每一个维度都做一样的归一化，所有特征不同维度

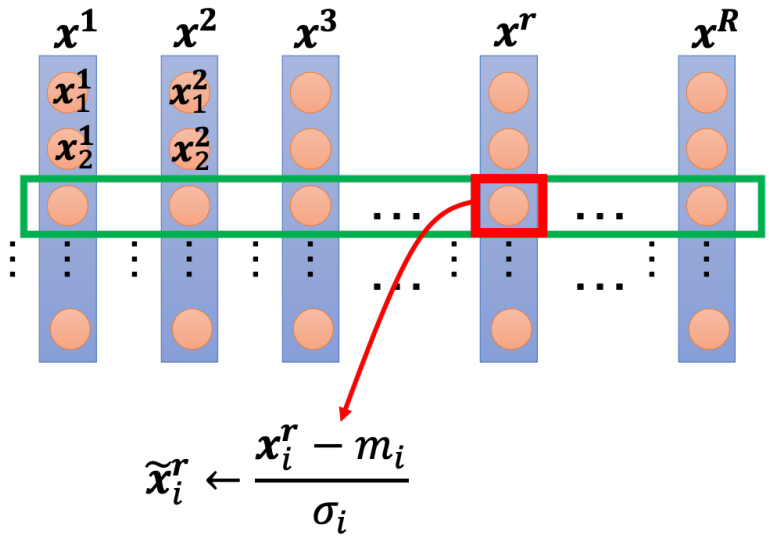


图 3.40 Z 值归一化

的数值都在 0 上下，可能就可以制造一个比较好的误差表面。所以像这样子的特征归一化方式往往对训练有帮助，它可以让在做梯度下降的时候，损失收敛更快一点，训练更顺利一点。

3.7.1 考虑深度学习

$\tilde{\boldsymbol{x}}$ 代表归一化的特征，把它丢到深度网络里面，去做接下来的计算和训练。如图 3.41 所示，$\tilde{\boldsymbol{x}}_1$ 通过第一层得到 $\boldsymbol{z}^1$，有可能通过激活函数，不管是选 sigmoid 或者 ReLU 都可以，再得到 $\boldsymbol{a}^1$，接着再通过下一层等等。对每个 $\boldsymbol{x}$ 都做类似的事情。

虽然 $\tilde{\boldsymbol{x}}$ 已经做归一化了，但是通过 W_1 以后，没有做归一化。如果 $\tilde{\boldsymbol{x}}$ 通过 W_1 得到 $\boldsymbol{z}^1$，而 $\boldsymbol{z}^1$ 不同的维度间，它的数值的分布仍然有很大的差异，训练 W_2 第二层的参数也会有困难。对于 W_2，$\boldsymbol{a}$ 或 $\boldsymbol{z}$ 其实也是一种特征，也应该要对这些特征做归一化。如果选择 sigmoid，比较推荐对 $\boldsymbol{z}$ 做特征归一化，因为 sigmoid 是一个 s 的形状，其在 0 附近斜率比较大，如果对 $\boldsymbol{z}$ 做特征归一化，把所有的值都挪到 0 附近，到时候算梯度的时候，算出来的值会比较大。如果使用别的激活函数，可能对 $\boldsymbol{a}$ 归一化也会有好的结果。一般而言，特征归一化，要放在激活函数之前，之后都是可以的，在实现上，没有太大的差别。

如何对 $\boldsymbol{z}$ 做特征归一化？$\boldsymbol{z}$ 可以看成另外一种特征。首先计算下 $\boldsymbol{z}^1, \boldsymbol{z}^2, \boldsymbol{z}^3$ 的平均值，即

$$\boldsymbol{\mu} = \frac{1}{3}\sum_{i=1}^{3} \boldsymbol{z}^i \tag{3.32}$$

接下来计算标准差

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{3}\sum_{i=1}^{3} (\boldsymbol{z}^i - \boldsymbol{\mu})^2} \tag{3.33}$$

注意，式 (3.33) 中的平方就是指对每一个元素都去做平方，开根号指的是对向量里面的每一个元素开根号。

最后，根据计算出的 $\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$ 进行归一化：

$$\tilde{\boldsymbol{z}}^i = \frac{\boldsymbol{z}^i - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{3.34}$$

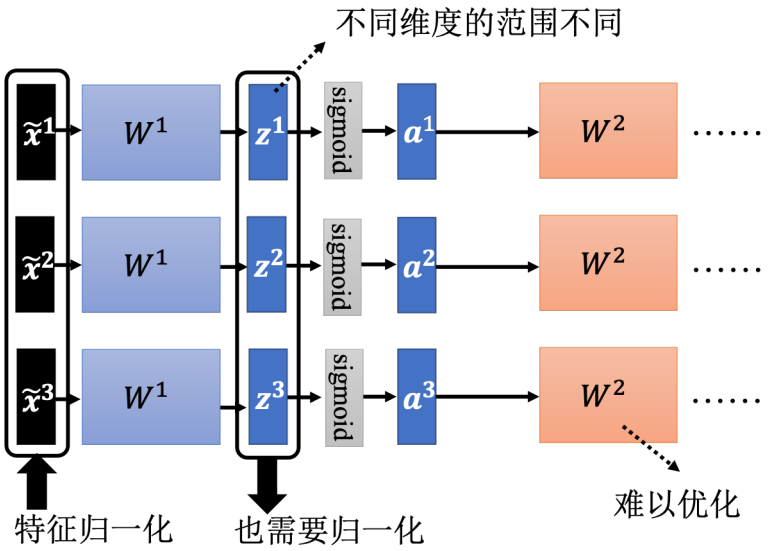


图 3.41 深度学习的归一化

其中，除号代表逐元素的除，即分子分母两个向量对应元素相除。

归一化的过程如图 3.42 所示。

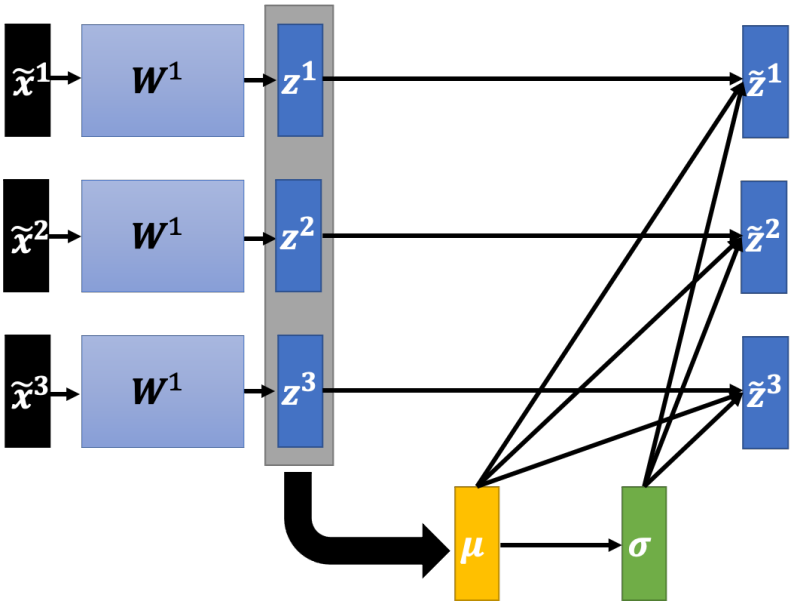


图 3.42 深度学习中间层的特征归一化

如图 3.43 所示，接下来可以通过激活函数得到其他向量，$\boldsymbol{\mu}$ 跟 $\boldsymbol{\sigma}$ 都是根据 $\boldsymbol{z}^1, \boldsymbol{z}^2, \boldsymbol{z}^3$ 计算出来的。改变了 $\boldsymbol{z}^1$ 的值，$\boldsymbol{a}^1$ 的值也会改变，$\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$ 也会改变。$\boldsymbol{\mu}, \boldsymbol{\sigma}$ 改后，$\boldsymbol{z}^2, \boldsymbol{a}^2, \boldsymbol{z}^3, \boldsymbol{a}^3$ 的值也会改变。之前的 $\tilde{\boldsymbol{x}}_1, \tilde{\boldsymbol{x}}_2$ $\tilde{\boldsymbol{x}}_3$ 是独立分开处理的，但是在做特征归一化以后，这三个样本变得彼此关联了。所以有做特征归一化的时候，可以把整个过程当做是网络的一部分。即有一个比较大的网络，该网络吃一堆输入，用这堆输入在这个网络里面计算出 $\boldsymbol{\mu}, \boldsymbol{\sigma}$，接下来产生一堆输出。这边就会有一个问题了，因为训练数据非常多，现在一个数据集可能有上百万笔

数据，GPU 的显存无法把它整个数据集的数据都加载进去。因此，在实现的时候，我们不会让这一个网络考虑整个训练数据里面的所有样本，而是只会考虑一个批量里面的样本。比如批量设 64，这个网络就是把 64 笔数据读进去，计算这 64 笔数据的 $\boldsymbol{\mu}, \boldsymbol{\sigma}$，对这 64 笔数据做归一化。因为实际实现的时候，只对一个批量里面的数据做归一化，所以技巧称为批量归一化。一定要有一个够大的批量，才算得出 $\boldsymbol{\mu}, \boldsymbol{\sigma}$。所以批量归一化适用于批量大小比较大的时候，批量大小如果比较大，也许这个批量大小里面的数据就足以表示整个数据集的分布。这个时候就不需要对整个数据集做特征归一化，而改成只在一个批量上做特征归一化作为近似。

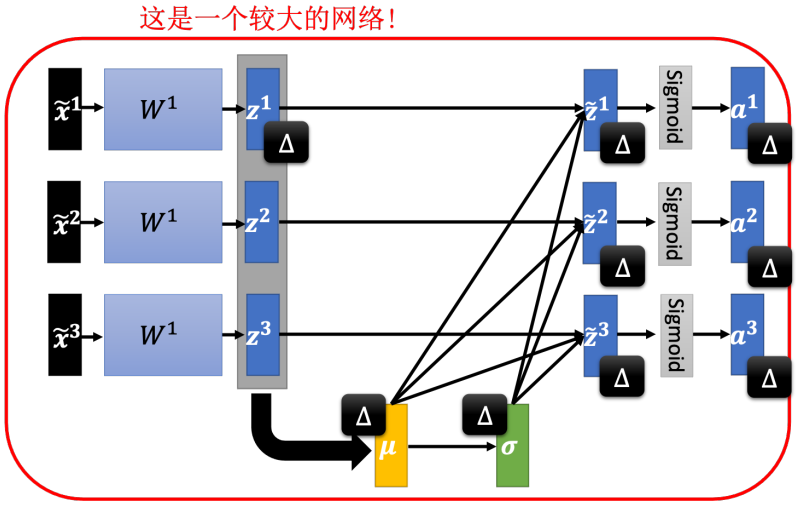


图 3.43　批量归一化可以理解为网络的一部分

在做批量归一化的时候，如图 3.44 所示，往往还会做如下操作：

$$\hat{\boldsymbol{z}}^i = \boldsymbol{\gamma} \odot \tilde{\boldsymbol{z}}^i + \boldsymbol{\beta} \tag{3.35}$$

其中，$\odot$ 代表逐元素的相乘。$\boldsymbol{\beta}, \boldsymbol{\gamma}$ 可以想成是网络的参数，需要另外再被学习出来。

Q：为什么要加上 $\boldsymbol{\beta}$ 跟 $\boldsymbol{\gamma}$ 呢?
A：如果做归一化以后，$\tilde{\boldsymbol{z}}$ 的平均值一定是 0，如果平均值是 0 的话，这会给网络一些限制，这个限制可能会带来负面的影响，所以需要把 $\boldsymbol{\beta}, \boldsymbol{\gamma}$ 加回去，让网络隐藏层的输出平均值不是 0。让网络学习 $\boldsymbol{\beta}, \boldsymbol{\gamma}$ 来调整一下输出的分布，从而来调整 $\hat{\boldsymbol{z}}$ 的分布。

Q：批量归一化是为了要让每一个不同的维度的范围相同，如果把 $\boldsymbol{\gamma}$ 跟 $\boldsymbol{\beta}$ 加进去，这样不同维度的分布，其范围不会又都不一样了吗?
A：有可能，但是实际上在训练的时候，$\boldsymbol{\gamma}$ 的初始值都设为 1，所以 $\boldsymbol{\gamma}$ 值都为 1 的向量。$\boldsymbol{\beta}$ 是值全部都是 0 的向量，即零向量。所以让网络在一开始训练的时候，每一个维度的分布，是比较接近的，也许训练到后来，已经训练够长的一段时间，已经找到一个比较好的误差表面，走到一个比较好的地方以后，再把 $\boldsymbol{\gamma}, \boldsymbol{\beta}$ 慢慢地加进去，所以加了 $\boldsymbol{\gamma}, \boldsymbol{\beta}$ 的批量归一化，往往对训练是有帮助的。

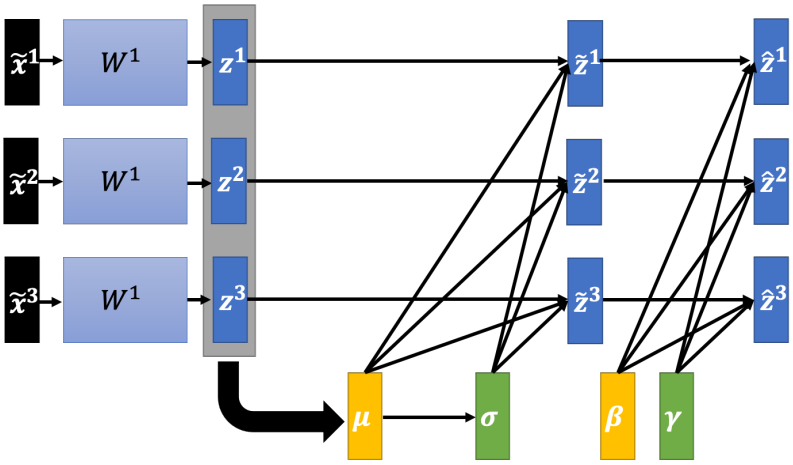


图 3.44　加 γ, β 的批量归一化

3.7.2　测试时的批量归一化

以上说的都是训练的部分，测试有时候又称为**推断（inference）**。批量归一化在测试的时候，会有什么样的问题呢？在测试的时候，我们一次会得到所有的测试数据，确实也可以在测试的数据上面，制造一个一个批量。但是假设系统上线，做一个真正的线上的应用，比如批量大小设 64，我一定要等 64 笔数据都进来，才做一次做运算，这显然是不行的。

但是在做批量归一化的时候，$\boldsymbol{\mu}, \boldsymbol{\sigma}$ 是用一个批量的数据算出来的。但如果在测试的时候，根本就没有批量，如何算 $\boldsymbol{\mu}, \boldsymbol{\sigma}$ 呢？所以真正的实现上的解法是这个样子的。批量归一化在测试的时候，并不需要做什么特别的处理，PyTorch 已经处理好了。在训练的时候，如果有在做批量归一化，每一个批量计算出来的 $\boldsymbol{\mu}, \boldsymbol{\sigma}$，都会拿出来算**移动平均（moving average）**。假设现在有各个批量计算出来的 $\boldsymbol{\mu}^1, \boldsymbol{\mu}^2, \boldsymbol{\mu}^3, \cdots\cdots, \boldsymbol{\mu}^t$，则可以计算移动平均

$$\bar{\boldsymbol{\mu}} \leftarrow p\bar{\boldsymbol{\mu}} + (1-p)\boldsymbol{\mu}^t \tag{3.36}$$

其中，$\bar{\boldsymbol{\mu}}$ 是 $\boldsymbol{\mu}$ 的个平均值，p 是因子，这也是一个常数，这也是一个超参数，也是需要调的那种。在 PyTorch 里面，p 设 0.1。计算滑动平均来更新 $\boldsymbol{\mu}$ 的平均值。最后在测试的时候，就不用算批量里面的 $\boldsymbol{\mu}$ 跟 $\boldsymbol{\sigma}$ 了。因为测试的时候，在真正应用上也没有批量，就可以就直接拿 $\bar{\mu}$ 跟 $\bar{\sigma}$ ，也就是 $\boldsymbol{\mu}, \boldsymbol{\sigma}$ 在训练的时候，得到的移动平均来取代原来的 $\boldsymbol{\mu}$ 跟 $\boldsymbol{\sigma}$，如图 3.45 所示，这就是批量归一化在测试的时候的运作方式。

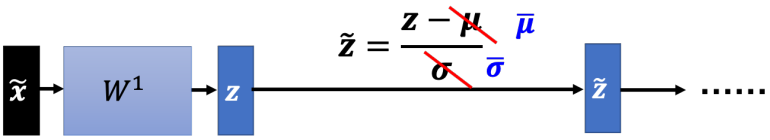


图 3.45　测试时的批量归一化

图 3.46 是从批量归一化原始文献的实验结果，横轴代表的是训练的过程，纵轴代表的是验证集上的准确率。黑色的虚线是没有做批量归一化的结果，它用的是 inception 的网络（一种网络以 CNN 为基础的网络架构）。如果有做批量归一化，则是红色的这一条虚线。红色虚线的训练速度显然比黑色的虚线还要快很多。虽然只要给模型足够的训练的时间，最后会收

敛都差不多的准确率。但是红色虚线可以在比较短的时间内跑到一样的准确率。蓝色的菱形代表说几个点的准确率是一样的。粉红色的线是 sigmoid 函数，一般的认知，但一般都会选择 ReLU，而不是用 sigmoid 函数，因为 sigmoid 函数的训练是比较困难的。但是这边想要强调的点是，就算是 sigmoid 比较难搞的加批量归一化，还是可以训练的，这边没有 sigmoid，没有做批量归一化的结果。因为在这个实验上，sigmoid 不加批量归一化，根本连训练都训练不起来。蓝色的实线跟这个蓝色的虚线呢是把学习率设比较大一点，×5 就是学习率变原来的 5 倍；×30 就是学习率变原来的 30 倍。因为如果做批量归一化，误差表面会比较平滑，比较容易训练，所以就可以把学习率设大一点。这边有个不好解释的地方，学习率设 30 倍的时候比 5 倍差，作者也没有解释。

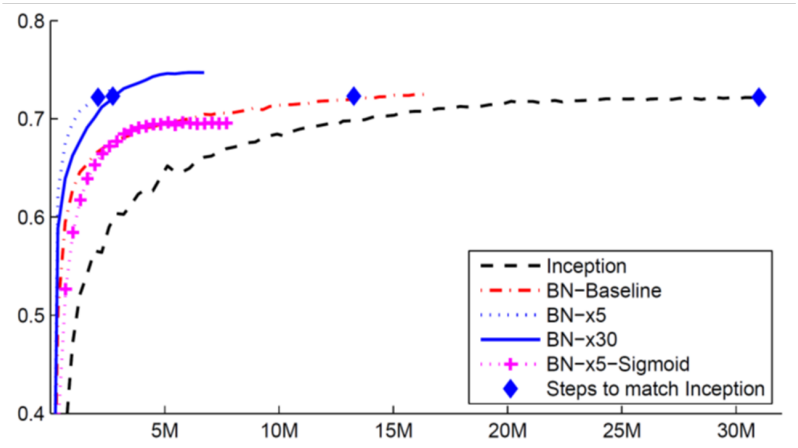


图 3.46　批量归一化实验结果[10]

3.7.3　内部协变量偏移

接下来的问题就是批量归一化为什么会有帮助呢？原始的批量归一化论文里面提出**内部协变量偏移（internal covariate shift）**概念。如图 3.47 所示，假设网络有很多层，$\boldsymbol{x}$ 通过第一层后得到 $\boldsymbol{a}$，$\boldsymbol{a}$ 通过第二层以后得到 $\boldsymbol{b}$；计算出梯度以后，把 $\boldsymbol{A}$ 更新成 $\boldsymbol{A}'$，把 $\boldsymbol{B}$ 这一层的参数更新成 $\boldsymbol{B}'$。但是作者认为说，我们在计算 $\boldsymbol{B}$ 更新到 $\boldsymbol{B}'$ 的梯度的时候，这个时候前一层的参数是 $\boldsymbol{A}$，或者是前一层的输出是 $\boldsymbol{a}$。那当前一层从 $\boldsymbol{A}$ 变成 $\boldsymbol{A}'$ 的时候，其输出就从 $\boldsymbol{a}$ 变成 $\boldsymbol{a}'$ 。但是我们计算这个梯度的时候，是根据 $\boldsymbol{a}$ 算出来，所以这个更新的方向也许它适合用在 $\boldsymbol{a}$ 上，但不适合用在 $\boldsymbol{a}'$ 上面。因为我们每次都有做批量归一化，就会让 $\boldsymbol{a}$ 和 $\boldsymbol{a}'$ 的分布比较接近，也许这样就会对训练有帮助。但是论文“How Does Batch Normalization Help Optimization?”[11] 认为内部协变量偏移有问题。这篇论文从不同的角度来说明内部协变量偏移不一定是训练网络的时候的一个问题。批量归一化会比较好，可能不一定是因为它解决了内部协变量偏移。这篇论文里面做了很多实验，比如其比较了训练的时候 $\boldsymbol{a}$ 的分布的变化，发现不管有没有做批量归一化，其变化都不大。就算是变化很大，对训练也没有太大的伤害。不管是根据 $\boldsymbol{a}$ 算出来的梯度，还是根据 $\boldsymbol{a}'$ 算出来的梯度，方向居然都差不多。内部协变量偏移可能不是训练网络的时候，最主要的问题，它可能也不是批量归一化会好的一个的关键。

协变量偏移（covariate shift），训练集和预测集样本分布不一致的问题就叫做协变量偏移现象，这个词汇是原来就有的，内部协变量偏移是批量归一化的作者自己发明的。

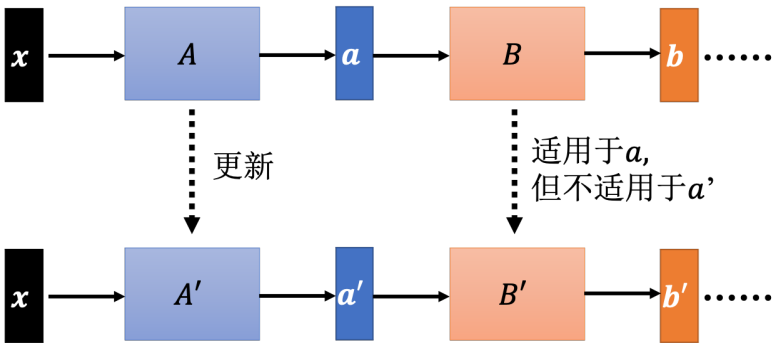


图 3.47　内部协变量偏移示例

为什么批量归一化会比较好呢，那在这篇“How Does Batch Normalization Help Optimization?”这篇论文从实验和理论上，至少支持批量归一化可以改变误差表面，让误差表面比较不崎岖这个观点。所以这个观点是有理论的支持，也有实验的佐证的。如果要让网络误差表面变得比较不崎岖，其实不一定要做批量归一化，还有很多其他的方法都可以让误差表面变得不崎岖，这篇论文就试了一些其他的方法，发现跟批量归一化表现也差不多，甚至还稍微好一点，这篇论文的作者也觉得批量归一化是一种偶然的发现，但无论如何，其是一个有用的方法。其实批量归一化不是唯一的归一化，还有很多归一化方法，比如批量重归一化（batch renormalization）[12]、层归一化（layer normalization）[13]、实例归一化（instance normalization）[14]、组归一化（group normalization）[15]、权重归一化（weight normalization）[16] 和谱归一化（spectrum normalization）[17]。

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.

[7] KINGMA D P, BA J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[8] 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.

[9] 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.

[10] IOFFE S, SZEGEDY C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C]//International conference on machine learning. pmlr, 2015: 448-456.

[11] SANTURKAR S, TSIPRAS D, ILYAS A, et al. How does batch normalization help optimization?[J]. Advances in neural information processing systems, 2018, 31.

[12] IOFFE S. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models[J]. Advances in neural information processing systems, 2017, 30.

[13] BA J L, KIROS J R, HINTON G E. Layer normalization[J]. arXiv preprint arXiv:1607.06450, 2016.

[14] ULYANOV D, VEDALDI A, LEMPITSKY V. Instance normalization: The missing ingredient for fast stylization[J]. arXiv preprint arXiv:1607.08022, 2016.

[15] WU Y, HE K. Group normalization[C]//Proceedings of the European conference on computer vision (ECCV). 2018: 3-19.

[16] SALIMANS T, KINGMA D P. Weight normalization: A simple reparameterization to accelerate training of deep neural networks[J]. Advances in neural information processing systems, 2016, 29.

[17] YOSHIDA Y, MIYATO T. Spectral norm regularization for improving the generalizability of deep learning[J]. arXiv preprint arXiv:1705.10941, 2017.

第 4 章　卷积神经网络

我们从卷积神经网络开始，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 4.1 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。

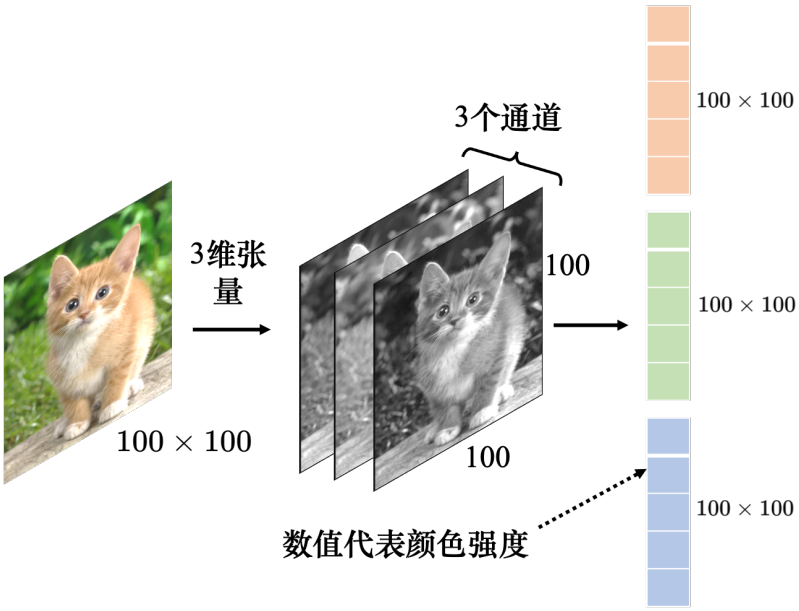


图 4.1　把图像作为模型输入

> 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 × 100像素。

如图 4.2 所示，如果把向量当做全连接网络的输入，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。

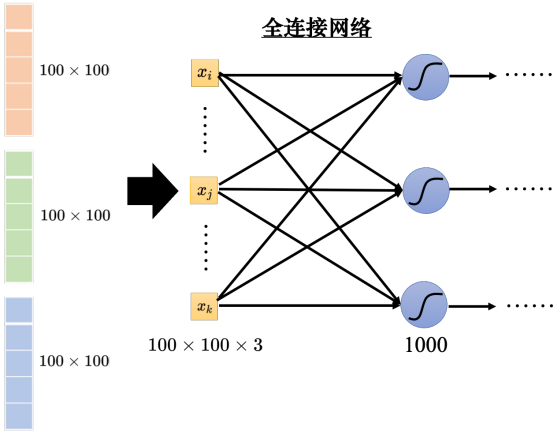


图 4.2　全连接网络

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 4.3 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 $\hat{y}$ 的交叉熵越小越好。

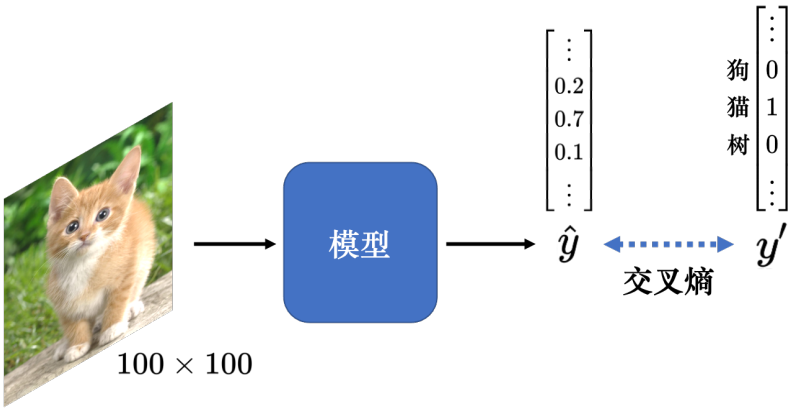


图 4.3　图像分类

4.1 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体的。比如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 4.4 所示。

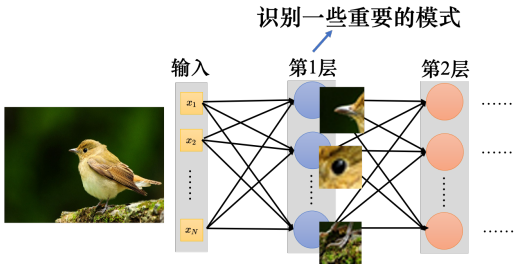


图 4.4 使用神经网络来检测模式

人在判断一个物体的时候，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体。对于机器，也许这是一个有效的判断图像中物体的方法。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 4.5 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。

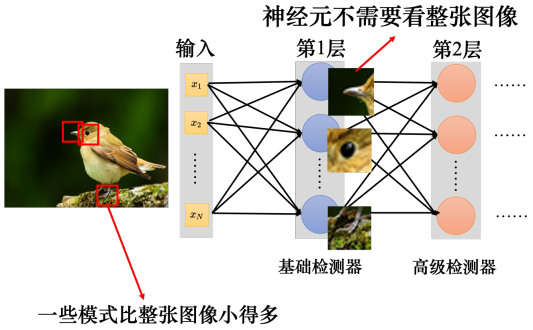


图 4.5 检测模式不需要整张图像

4.2 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，即**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。比如在图 4.6 中，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。

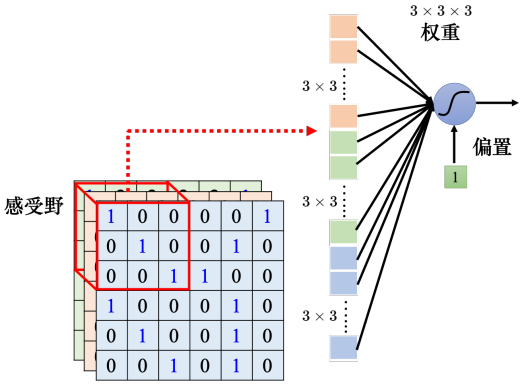


图 4.6　感受野

如图 4.7 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 4.7 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。

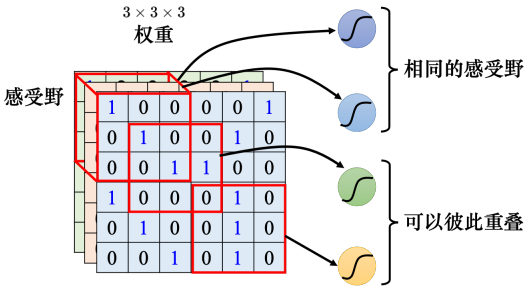


图 4.7　感受野彼此重叠

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 3×3 的范围内就可以被检测出来，有的模式也许要 11×11 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 3×3、11×11，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的位置。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。图 4.8 中的核大小就是 3×3。在图像识别里面，一般核大小不会设太大，3×3 的核大小就足够了，7×7、9×9 算是蛮大的核大小。如果核大小都是 3×3，意味着我们觉得在做图像识别的时候，重要的模式都只在 3×3 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 3×3 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 3×3。

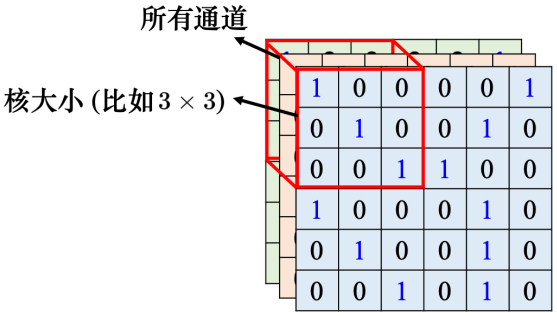


图 4.8 卷积核

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。移动的量称为**步幅（stride）**，图 4.9 中的这个例子里面，步幅就等于 2。步幅是一个超参数，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。

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。如令步幅 $= 2$，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 4.10 所示，超出范围就做**填充（padding）**，填充就是补

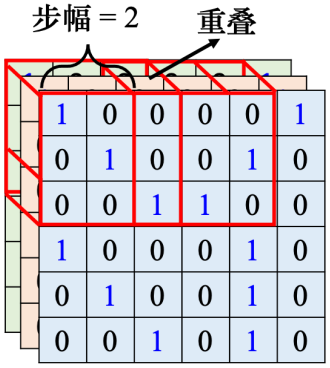


图 4.9　步幅

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。

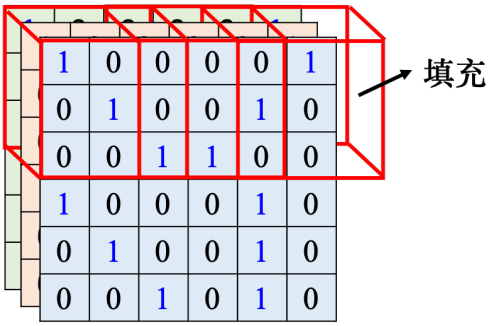


图 4.10　填充

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 4.11 所示。我们就按照这个方式扫过整张图像，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。

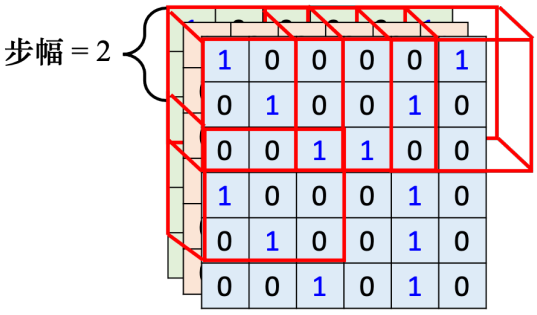


图 4.11　垂直移动

4.3 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 4.12 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。

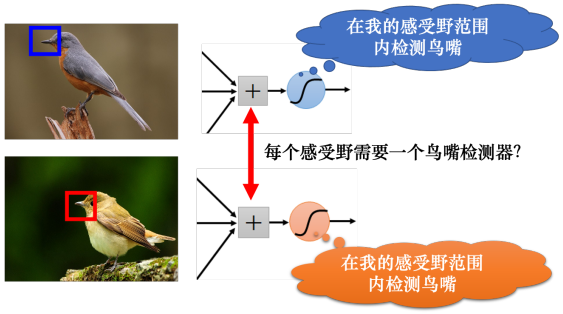


图 4.12 每个感受野都放一个鸟嘴检测器

4.4 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 4.13 所示。所谓参数共享就是两个神经元的权重完全是一样的。

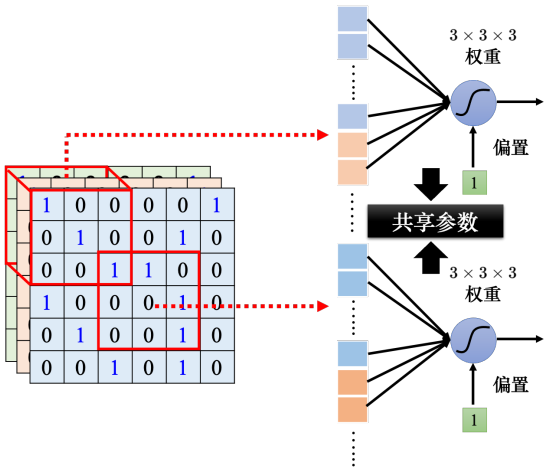


图 4.13 共享参数

如图 4.14 所示，颜色相同，权重完全是一样的，比如上面神经元的第 1 个权重是 w_1，下面神经元的第 1 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 $x_1, x_2, \cdots\cdots$，下面神经元的输入是 $x_1', x_2', \cdots\cdots$。上面神经元的输出为

$$\sigma\left(w_1 x_1 + w_2 x_2 + \cdots\cdots + 1\right) \tag{4.1}$$

下面神经元的输出为

$$\sigma\left(w_1 x_1' + w_2 x_2' + \cdots\cdots + 1\right) \tag{4.2}$$

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。

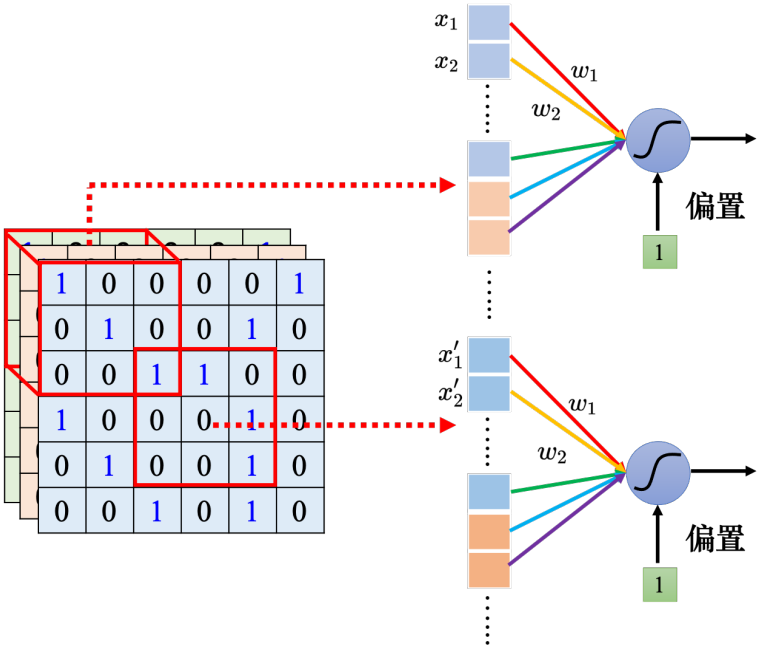


图 4.14　两个神经元共享参数

如图 4.15 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 4.16 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。

4.5　简化 1 和 2 的总结

目前已经讲了两个简化的方法，我们来总结下。如图 4.17 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

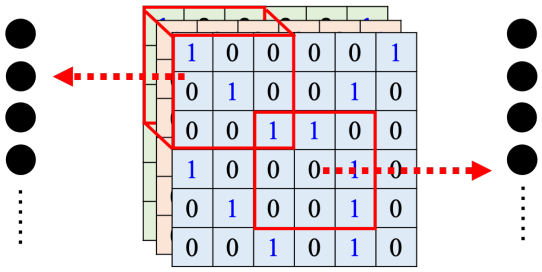


图 4.15 守备感受野的神经元

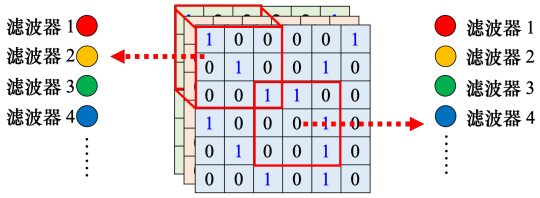


图 4.16 多个神经元共享参数

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，模型的灵活性较低时，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。

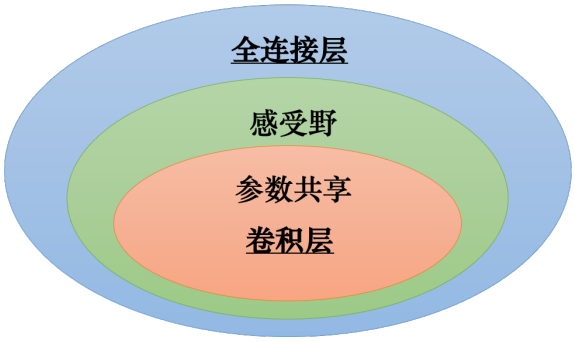


图 4.17 卷积层与全连接层的关系

接下来通过第 2 个版本的故事来说明卷积神经网络。如图 4.18 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 4.19 所示。如图 4.20 所示，这是一个 6×6 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 4.20 中的步幅为 1。使用滤波器 1 检测模式时，如果出现图像 3×3 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。

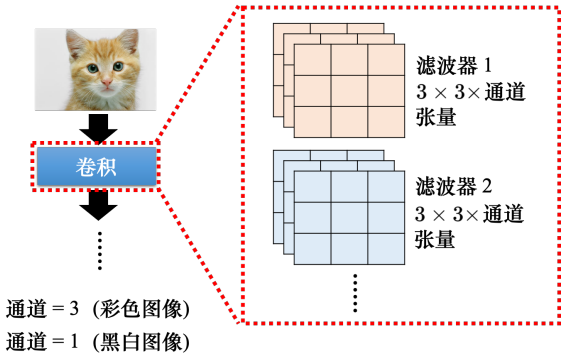


图 4.18　卷积层中的滤波器

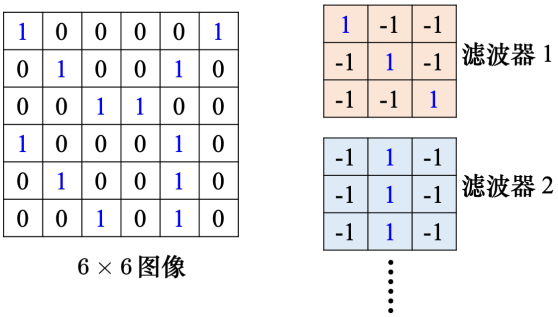


图 4.19　滤波器示例

如图 4.21 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 3×3 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 4×4，即第 1 个滤波器产生 4×4 个数字，第 2 个滤波器也产生 4×4 个数字，第 3 个也产生 4×4 个数字，64 个滤波器都产生 4×4 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

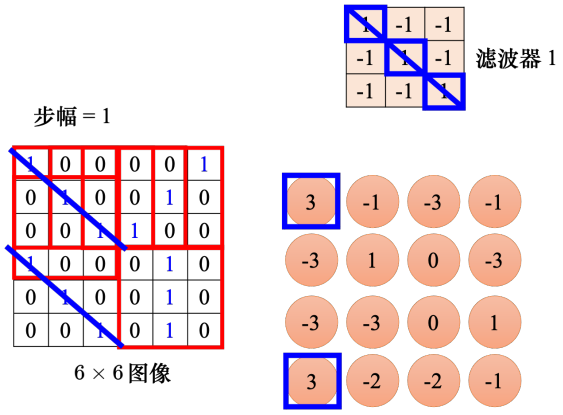


图 4.20　使用滤波器检测模式

张新的有 64 个通道图像。

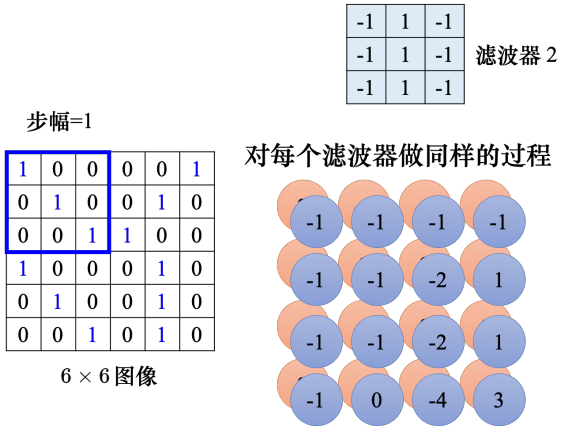


图 4.21　使用多个滤波器检测模式

卷积层是可以叠很多层的，如图 4.22 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 × 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。

Q：如果滤波器的大小一直设 3 × 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 4.23 所示，如果在第 2 层卷积层滤波器的大小一样设 3 × 3，当我们看第 1 个卷积层输出的特征映射的 3 × 3 的范围的时候，在原来的图像上是考虑了一个 5 × 5 的范围。虽然滤波器只有 3 × 3，但它在图像上考虑的范围是比较大的是 5 × 5。因此网络叠得越深，同样是 3 × 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。

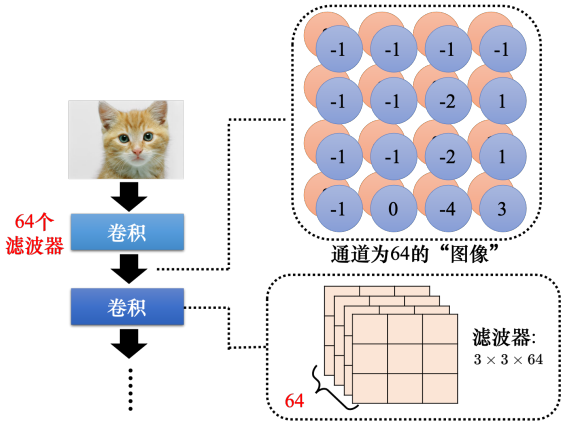


图 4.22　对图像进行卷积

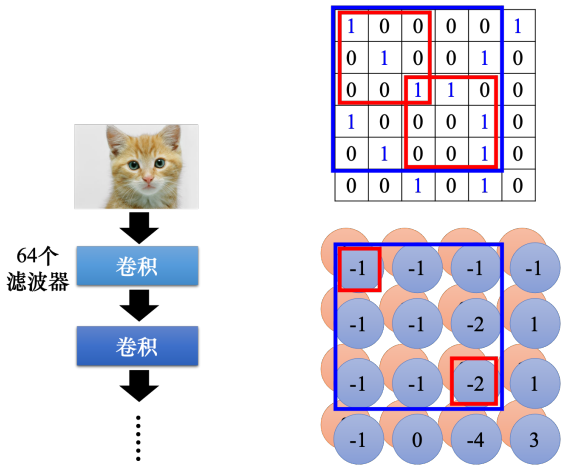


图 4.23　网络越深，可以检测的模式越大

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 4.24 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 4.25 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。

4.6　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 (downsampling)**，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。如图 4.26 所示，把一张大的鸟的图像缩小，这张小的图像还是一只鸟。

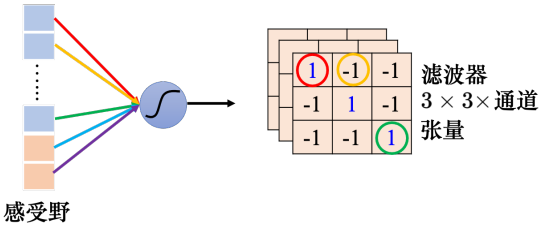


图 4.24　共享参数示例

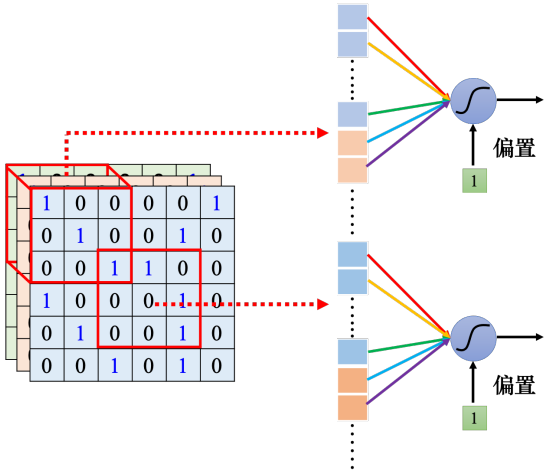


图 4.25　从不同的角度理解参数共享

4.7　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。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 2×2 个一组，3×3、4×4 也可以，这个是我们自己决定的，图 4.27 中的例子是 2×2 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一个代表，选的代表就是最大的一个，如图 4.28 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。

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 4.29 所示，

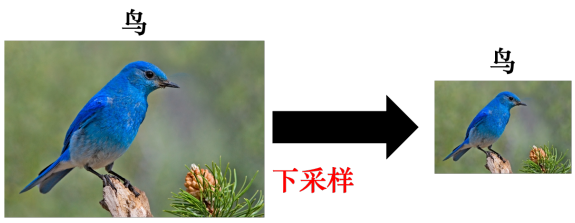


图 4.26　下采样示意

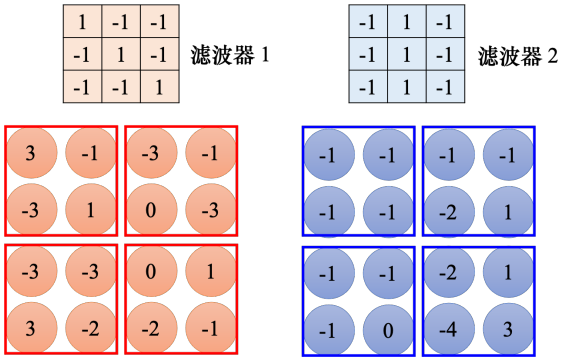


图 4.27　最大汇聚示例

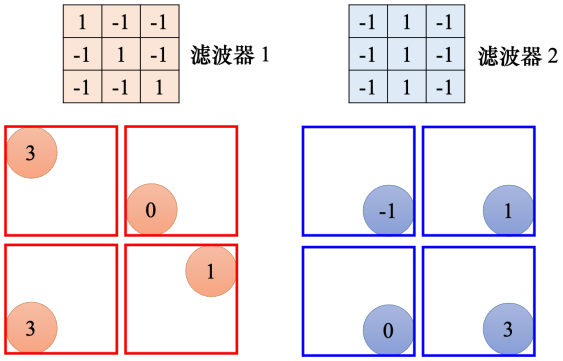


图 4.28　最大汇聚结果

在刚才的例子里面，本来 4×4 的图像，如果把这个输出的数值 2×2 个一组，4×4 的图像就会变成 2×2 的图像，这就是汇聚所做的事情。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。

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 4.30 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。

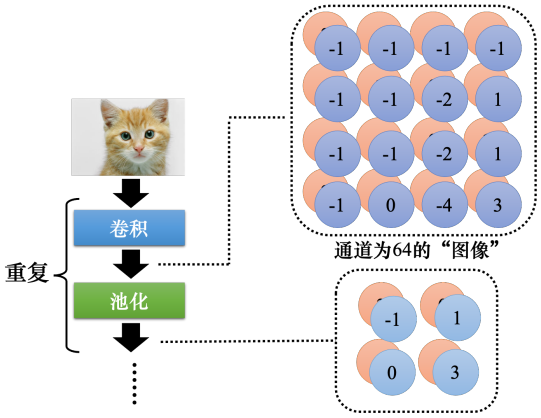


图 4.29　重复使用卷积和池化

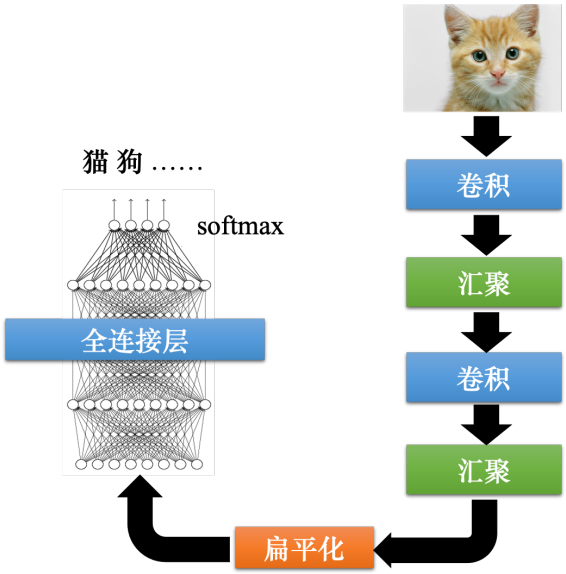


图 4.30　经典的图像识别的网络

4.8　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 19×19 个位置，可以把一个棋盘表示成一个 19×19 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 19×19 个类别的分类问题，网络会输出 19×19 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。

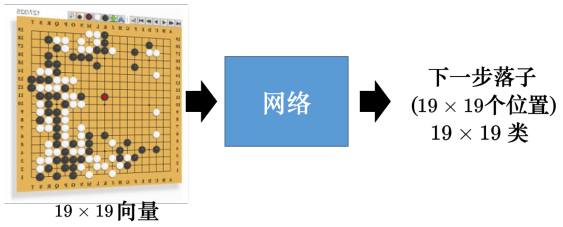


图 4.31　使用卷积神经网络下围棋

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为 19×19 的图像。一般图像很大，100×100 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 19×19。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[1]。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 × 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 4.32 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 × 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 × 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 4.33 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。

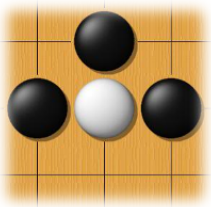


图 4.32　围棋的模式

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体的判断。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，而是在附件中介绍了这个细节。AlphaGo 把一个棋盘看作 19 × 19 × 48 大小的图像。接下来它有做零

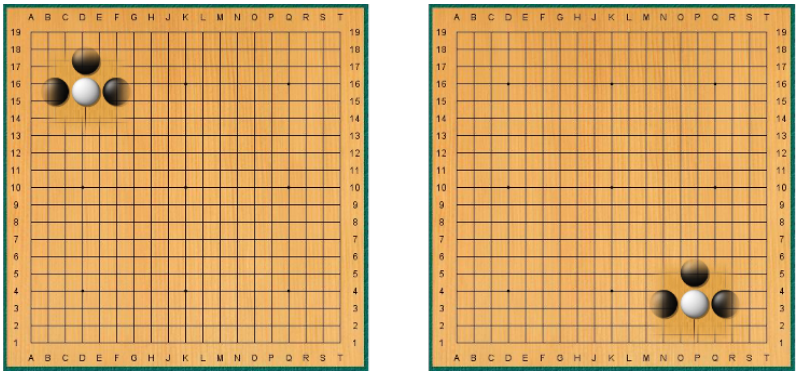


图 4.33 吃的模式

填充。它的滤波器的大小是 5×5，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 3×3，一样是 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”[2] 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”[3] 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，要怎么设计合适的感受野。

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。假设图像里面的物体都是比较小的，当卷积神经网络在某种大小的图像上面学会做图像识别，我们把物体放大，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。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，让它看过某一个物体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 5 章 循环神经网络

循环神经网络（Recurrent Neural Network）是深度学习领域中一种非常经典的网络结构，在现实生活中有着广泛的应用。以槽填充（slot filling）为例，如图 5.1 所示，假设订票系统听到用户说：“我想在 6 月 1 日抵达上海。”，系统有一些槽（slot）：目的地和到达时间，系统要自动知道这边的每一个单词是属于哪一个槽，比如“上海”属于目的地槽，“6 月 1 号”属于到达时间槽。

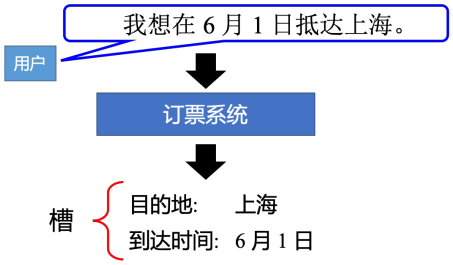


图 5.1 槽填充示例

这个问题可以使用一个前馈神经网络（feedforward neural network）来解，如图 5.2 所示，输入是一个单词，把“上海”变成一个向量，“丢”到这个神经网络里面。要把一个单词丢到一个神经网络里面去，就必须把它变成一个向量。

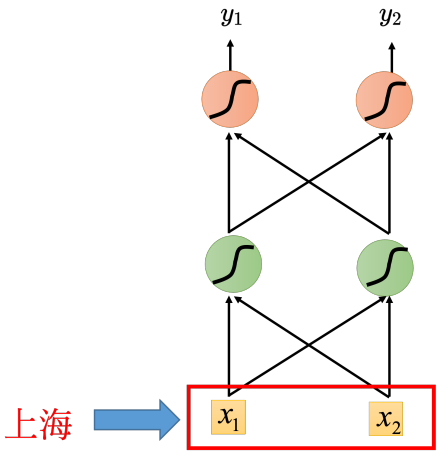


图 5.2 使用神经网络解决槽填充问题

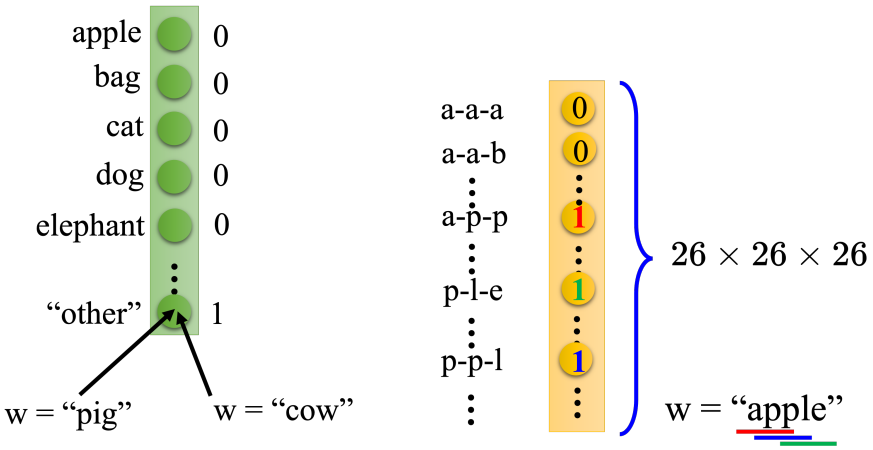
以下是把单词用向量来表示的方法。

5.1 独热编码

假设词典中有 5 个单词：apple，bag，cat，dog，elephant，如式 (5.1)。向量的大小是词典大小。每一个维度对应词典的一个单词。对应单词的维度为 1，其他为 0。

$$\begin{aligned} \text{apple} &= [1,0,0,0,0] \\ \text{bag} &= [0,1,0,0,0] \\ \text{cat} &= [0,0,1,0,0] \\ \text{dog} &= [0,0,0,1,0] \\ \text{elephant} &= [0,0,0,0,1] \end{aligned} \tag{5.1}$$

如果只是用独热编码来描述一个单词，会有一些问题：因为很多单词可能都没有见过，所以需要在独热编码里面多加维度，用一个维度代表 other，如图 5.3(a) 所示。如果不是在词表中，有的单词就归类到 other 里面去（Pig，Cow 归类到 other 里面去）。我们可以用每一个单词的字母来表示它的向量，比如单词是 apple，apple 里面有出现 app、ple、ppl，在这个向量里面对应到 app、ple、ppl 的维度就是 1，其他都为 0，如图 5.3(b) 所示。



(a) 用 other 表示非词表中的单词　　(b) 词哈希

图 5.3　另一种编码方法

假设把单词表示为向量，把这个向量丢到前馈神经网络里面去，在该任务里面，输出是一个概率分布，该概率分布代表着输入单词属于每一个槽的概率，比如“上海”属于目的地的概率和“上海”属于出发地的概率，如图 5.4 所示。但是前馈网络会有问题，如图 5.5 所示，假设用户 1 说：“在 6 月 1 号抵达上海”。用户 2 说：“在 6 月 1 号离开上海”，这时候“上海”就变成了出发地。但是对于神经网络，输入一样的东西，输出就应该是一样的东西。在例子中，输入“上海”，输出要么让目的地概率最高，要么让出发地概率最高。不能一会让出发地的概率最高，一会让目的地概率最高。在这种情况下，如果神经网络有记忆力的，它记得它看过“抵达”，在看到“上海”之前；或者它记得它已经看过“离开”，在看到“上海”之前。通过记忆力，它可以根据上下文产生不同的输出。如果让神经网络是有记忆力，其就可以解决输入不同的单词，输出不同的问题。

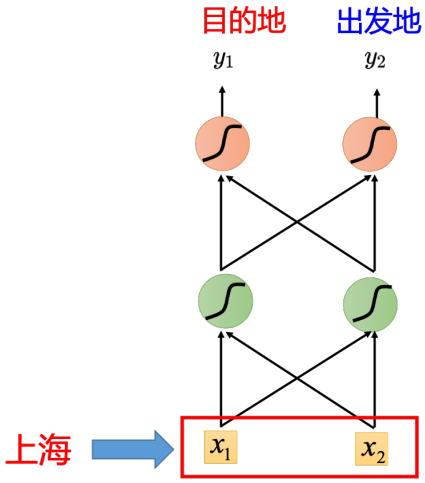


图 5.4 使用前馈神经网络预测概率分布

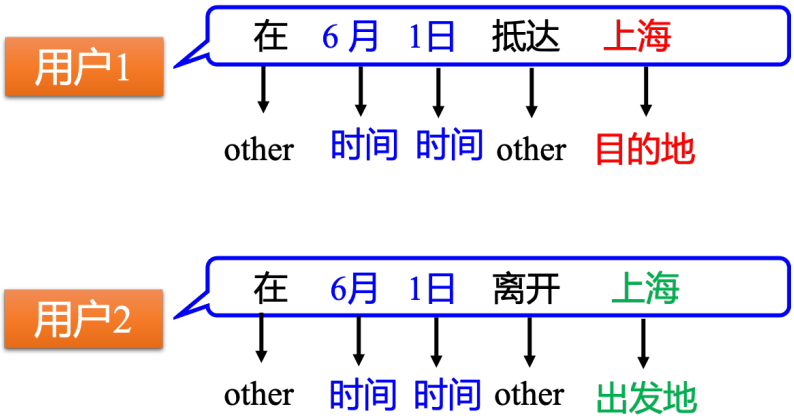


图 5.5 前馈神经网络的问题

5.2 什么是 RNN?

在 RNN 里面，每一次隐藏层的神经元产生输出的时候，该输出会被存到记忆元（memory cell），图 5.6(a) 中的蓝色方块表示记忆元。下一次有输入时，这些神经元不仅会考虑输入 x_1, x_2，还会考虑存到记忆元里的值。除了 x_1, x_2，存在记忆元里的值 a_1, a_2 也会影响神经网络的输出。

记忆元可简称为单元（cell），记忆元的值也可称为隐状态（hidden state）。

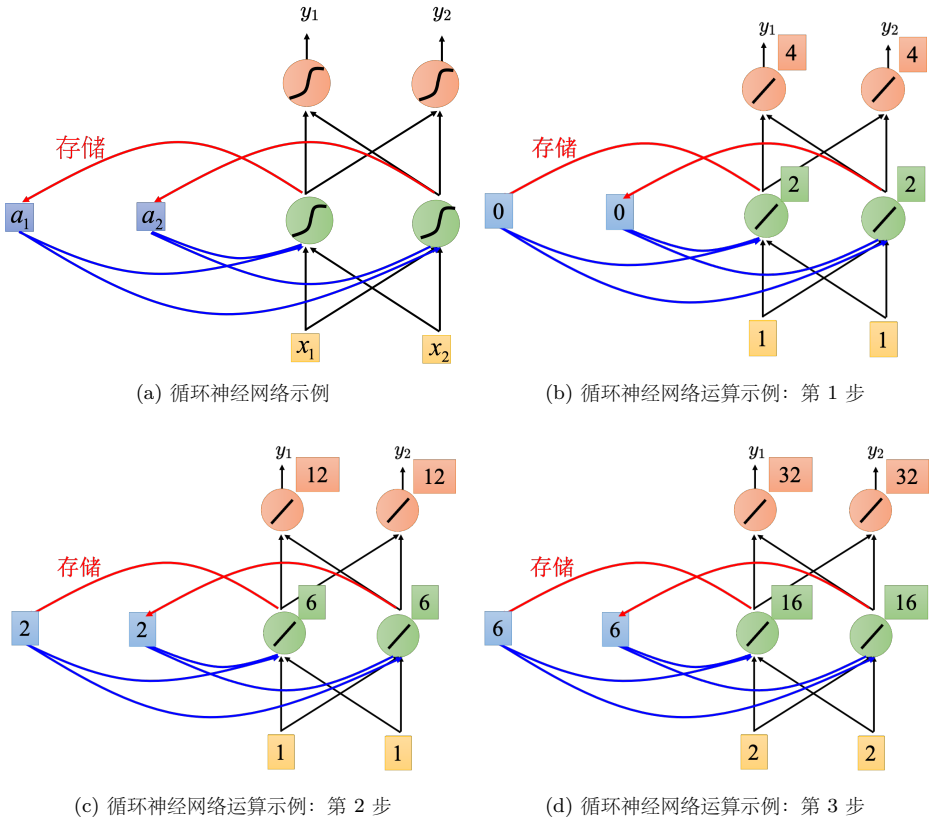
举个例子，假设图 5.6(b) 中的神经网络所有的权重都是 1，所有的神经元没有任何的偏置（bias）。为了便于计算，假设所有的激活函数都是线性的，输入是序列 $[1,1]^{\mathrm{T}}, [1,1]^{\mathrm{T}}, [2,2]^{\mathrm{T}}, \cdots$，所有的权重都是 1。首先设置记忆元的初始值都为 0，接着输入第一个 $[1,1]^{\mathrm{T}}$，对于左边的神经元（第一个隐藏层），其除了接到输入的 $[1,1]^{\mathrm{T}}$，还接到了记忆元（0 跟 0），输出就是 2。同理，右边神经元的输出为 2，第二层隐藏层输出为 4。

接下来循环神经网络会将绿色神经元的输出存在记忆元里去，所以记忆元里面的值被更

新为 2。如图 5.6(c) 所示，接下来再输入 $[1,1]^{\mathrm{T}}$，接下来绿色的神经元输入为 $[1,1]^{\mathrm{T}}$、$[2,2]^{\mathrm{T}}$，输出为 $[6,6]^{\mathrm{T}}$，第二层的神经元输出为 $[12,12]^{\mathrm{T}}$。所以因为循环神经网络有记忆元，就算输入相同，输出也可能不一样。

如图 5.6(d) 所示，$[6,6]^{\mathrm{T}}$ 存到记忆元里去，接下来输入是 $[2,2]^{\mathrm{T}}$，输出为 $[16,16]^{\mathrm{T}}$；第二层隐藏层为 $[32,32]^{\mathrm{T}}$。在做循环神经网络时，它会考虑序列的顺序，输入序列调换顺序之后输出不同。

因为当前时刻的隐状态使用与上一时刻隐状态相同的定义，所以隐状态的计算是循环的（recurrent），基于循环计算的隐状态神经网络被称为循环神经网络。



(a) 循环神经网络示例　(b) 循环神经网络运算示例：第 1 步

(c) 循环神经网络运算示例：第 2 步　(d) 循环神经网络运算示例：第 3 步

图 5.6　循环神经网络运算示例

5.3　RNN 架构

使用循环神经网络处理槽填充这件事，如图 5.7 所示。用户说：“我想在 6 月 1 日抵达上海”，“抵达”就变成了一个向量“丢”到神经网络里面去，神经网络的隐藏层的输出为向量 $\boldsymbol{a}_1$，$\boldsymbol{a}_1$ 产生“抵达”属于每一个槽填充的概率 $\boldsymbol{y}_1$。接下来 $\boldsymbol{a}_1$ 会被存到记忆元里面去，“上海”会变为输入，这个隐藏层会同时考虑“上海”这个输入和存在记忆元里面的 $\boldsymbol{a}_1$，得到 $\boldsymbol{a}_2$。根据 $\boldsymbol{a}_2$ 得到 $\boldsymbol{y}_2$，$\boldsymbol{y}_2$ 是属于每一个槽填充的概率。

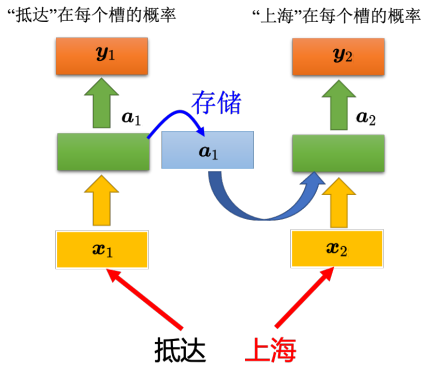


图 5.7　使用循环神经网络处理槽填充

这个不是三个网络，这是同一个网络在三个不同的时间点被使用了三次，用同样的权重用同样的颜色表示。

有了记忆元以后，输入同一个单词，希望输出不同的问题就有可能被解决。如图 5.8 所示，同样是输入“上海”这个单词，但是因为红色“上海”前接了“离开”，绿色“上海”前接了“抵达”，“离开”和“抵达”的向量不一样，隐藏层的输出会不同，所以存在记忆元里面的值会不同。虽然 $\boldsymbol{x}_2$ 的值是一样的，因为存在记忆元里面的值不同，所以隐藏层的输出会不同，所以最后的输出也就会不一样。

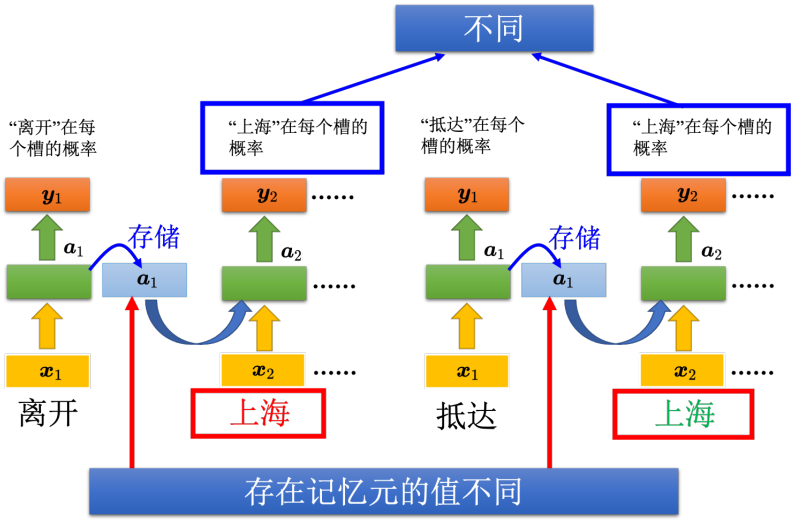


图 5.8　输入相同，输出不同示例

5.4　其他 RNN

循环神经网络的架构是可以任意设计的，之前提到的 RNN 只有一个隐藏层，但 RNN 也可以是深层的。比如把 $\boldsymbol{x}_t$ 丢进去之后，它可以通过一个隐藏层，再通过第二个隐藏层，以此类推（通过很多的隐藏层）才得到最后的输出。每一个隐藏层的输出都会被存在记忆元里面，

在下一个时间点的时候，每一个隐藏层会把前一个时间点存的值再读出来，以此类推最后得到输出，这个过程会一直持续下去。

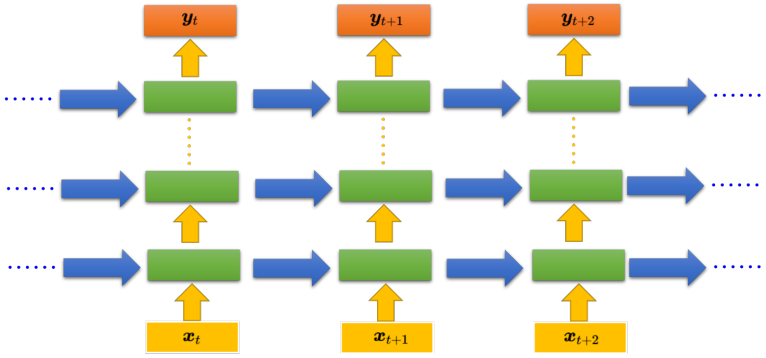


图 5.9　深层循环神经网络

5.4.1　Elman 网络 &Jordan 网络

循环神经网络会有不同的变形，如图 5.10 所示，刚才讲的是简单循环网络（Simple Recurrent Network，SRN），即把隐藏层的值存起来，在下一个时间点在读出来。还有另外一种叫做 Jordan 网络，Jordan 网络存的是整个网络输出的值，它把输出值在下一个时间点在读进来，把输出存到记忆元里。Elman 网络没有目标，很难控制说它能学到什么隐藏层信息（学到什么放到记忆元里），但是 Jordan 网络是有目标，比较很清楚记忆元存储的东西。

简单循环网络也称为 Elman 网络。

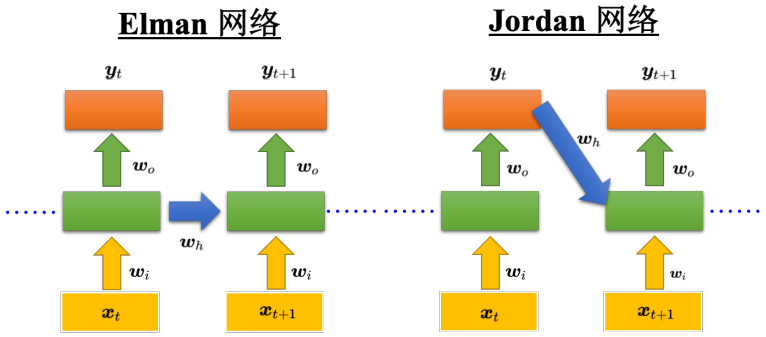


图 5.10　Elman 网络和 Jordan 网络

5.4.2　双向循环神经网络

循环神经网络还可以是双向。刚才 RNN 输入一个句子，它就是从句首一直读到句尾。如图 5.11 所示，假设句子里的每一个单词用 $\boldsymbol{x}_t$ 表示，其是先读 $\boldsymbol{x}_t$，再读 $\boldsymbol{x}_{t+1}$、$\boldsymbol{x}_{t+2}$。但其读取方向也可以是反过来的，它可以先读 $\boldsymbol{x}_{t+2}$，再读 $\boldsymbol{x}_{t+1}$、$\boldsymbol{x}_t$。我们可以同时训练一个正向的循环神经网络，又可以训练一个逆向的循环神经网络，然后把这两个循环神经网络的隐藏层拿出来，都接给一个输出层得到最后的 $\boldsymbol{y}_t$。所以把正向的网络在输入 $\boldsymbol{x}_t$ 的时候跟逆向的

网络在输入 $\boldsymbol{x}_t$ 时，都丢到输出层产生 $\boldsymbol{y}_t$，产生 $\boldsymbol{y}_{t+1}, \boldsymbol{y}_{t+2}$，以此类推。双向循环神经网络（Bidirectional Recurrent Neural Network，Bi-RNN）的好处是，神经元产生输出的时候，它看的范围是比较广的。如果只有正向的网络，再产生 $\boldsymbol{y}_t$、$\boldsymbol{y}_{t+1}$ 的时候，神经元只看过 $\boldsymbol{x}_1$ 到 $\boldsymbol{x}_{t+1}$ 的输入。但双向循环神经网络产生 $\boldsymbol{y}_{t+1}$ 的时候，网络不只是看过 $\boldsymbol{x}_1$，到 $\boldsymbol{x}_{t+1}$ 所有的输入，它也看了从句尾到 $\boldsymbol{x}_{t+1}$ 的输入。网络就等于整个输入的序列。假设考虑的是槽填充，网络就等于看了整个句子后，才决定每一个单词的槽，这样会比看句子的一半还要得到更好的性能。

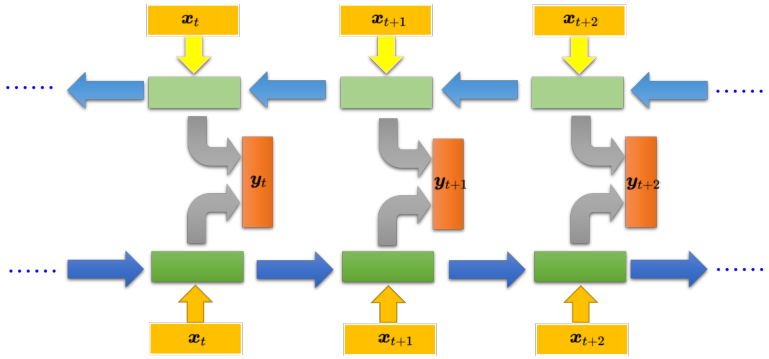


图 5.11　双向循环神经网络

5.4.3　长短期记忆网络

之前提到的记忆元是最单纯的，可以随时把值存到记忆元去，也可以把值读出来。但最常用的记忆元是**长短期记忆网络（Long Short-Term Memory network，LSTM）**，长时间的短期记忆。LSTM 是比较复杂的。LSTM 有三个**门（gate）**，当外界某个神经元的输出想要被写到记忆元里面的时候，必须通过一个**输入门（input gate）**，输入门要被打开的时候，才能把值写到记忆元里面。如果把这个关起来的话，就没有办法把值写进去。至于输入门的开关是神经网络自己学的，其可以自己学什么时候要把输入门打开，什么时候要把输入门关起来。输出的地方也有一个**输出门（output gate）**，输出门会决定外界其他的神经元能否从这个记忆元里面把值读出来。把输出门关闭的时候是没有办法把值读出来，输出门打开的时候才可以把值读出来。跟输入门一样，输出门什么时候打开什么时候关闭，网络是自己学到的。第三个门称为**遗忘门（forget gate）**，遗忘门决定什么时候记忆元要把过去记得的东西忘掉。这个遗忘门什么时候会把存在记忆元的值忘掉，什么时候会把存在记忆元里面的值继续保留下来，这也是网络自己学到的。整个 LSTM 可以看成有 4 个输入、1 个输出。在这 4 个输入中，一个是想要被存在记忆元的值，但不一定能存进去，还有操控输入门的信号、操控输出门的信号、操控遗忘门的信号，有着四个输入但它只会得到一个输出。

“-”应该在 short-term 中间，是长时间的短期记忆。之前的循环神经网络，它的记忆元在每一个时间点都会被洗掉，只要有新的输入进来，每一个时间点都会把记忆元洗掉，所以的短期是非常短的，但如果是长时间的短期记忆元，它记得会比较久一点，只要遗忘门不要决定要忘记，它的值就会被存起来。

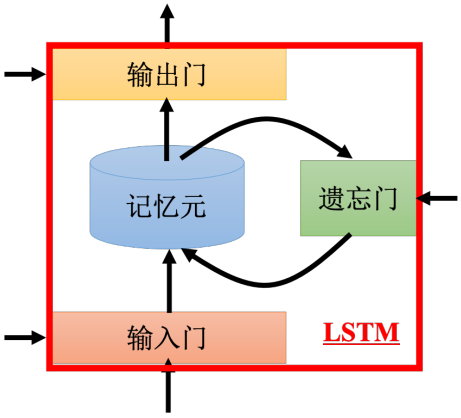


图 5.12 LSTM 结构

记忆元对应的计算公式为

$$c' = g(z)f(z_i) + cf(z_f) \tag{5.2}$$

如图 5.13 所示，底下这个是外界传入单元的输入，还有输入门、遗忘门和输出门。假设要被存到单元的输入叫做 z，操控输入门的信号为 z_i，操控遗忘门的信号为 z_f，操控输出门为 z_o，综合这些东西会得到一个输出记为 a。假设单元里面有这四个输入之前，它里面已经存了值 c。输出 a 会长什么样子，把 z 通过激活函数得到 $g(z)$，z_i 通过另外一个激活函数得到 $f(z_i)$（激活函数通常会选择 sigmoid 函数），因为其值介在 0 到 1 之间的，这个 0 到 1 之间的值代表了这个门被打开的程度。（如果 f 的输出是 1，表示为被打开的状态，反之代表这个门是关起来的）。

接下来，把 $g(z)$ 乘以 $f(z_i)$ 得到 $g(z)f(z_i)$，对于遗忘门的 z_f，也通过 sigmoid 的函数得到 $f(z_f)$ 接下来把存到记忆元里面的值 c 乘以 $f(z_f)$ 得到 $cf(z_f)$，加起来 $c^{'} = g(z)f(z_i)+cf(z_f)$，那么 $c^{'}$ 就是重新存到记忆元里面的值。所以根据目前的运算，这个 $f(z_i)$ 控制这个 $g(z)$。假设输入 $f(z_i)0$，那 $g(z)f(z_i)$ 就等于 0，就好像是没有输入一样，如果 $f(z_i)$ 等于 1 就等于是把 $g(z)$ 当做输入。那这个 $f(z_f)$ 决定是否要把存在记忆元的值洗掉，假设 $f(z_f)$ 为 1，遗忘门开启的时候，这时候 c 会直接通过，把之前的值还会记得。如果 $f(z_f)$ 等于 0(遗忘门关闭的时候)$cf(z_f)$ 等于 0。然后把这个两个值加起来 $(c^{'} = g(z)f(z_i)+cf(z_f))$ 写到记忆元里面得到 $c^{'}$。这个遗忘门的开关是跟直觉是相反的，遗忘门打开的时候代表的是记得，关闭的时候代表的是遗忘。那这个 $c^{'}$ 通过 $h(c^{'})$，将 $h(c^{'})$ 乘以 $f(z_o)$ 得到 $a = f(c^{'}f(z_o)$。输出门受 $f(z_o)$ 所操控，$f(z_o)$ 等于 1 的话，就说明 $h(c^{'})$ 能通过，$f(z_o)$ 等于 0 的话，说明记忆元里面存在的值没有办法通过输出门被读取出来。

5.4.4 LSTM 举例

如图 5.14 所示，网络里面只有一个 LSTM 的单元，输入都是三维的向量，输出都是一维的输出。这三维的向量跟输出还有记忆元的关系是这样的。假设 x_2 的值是 1 时，x_1 的值就会被写到记忆元里；假设 x_2 的值是-1 时，就会重置这个记忆元；假设 x_3 的值为 1 时，才会把输出打开，才能看到输出，看到记忆元的数字。

假设原来存到记忆元里面的值是 0，当第二个维度 x_2 的值是 1 时，3 会被存到记忆元里面去。第四个维度的 x_2 等于，所以 4 会被存到记忆元里面去，所以会得到 7。第六个维度的

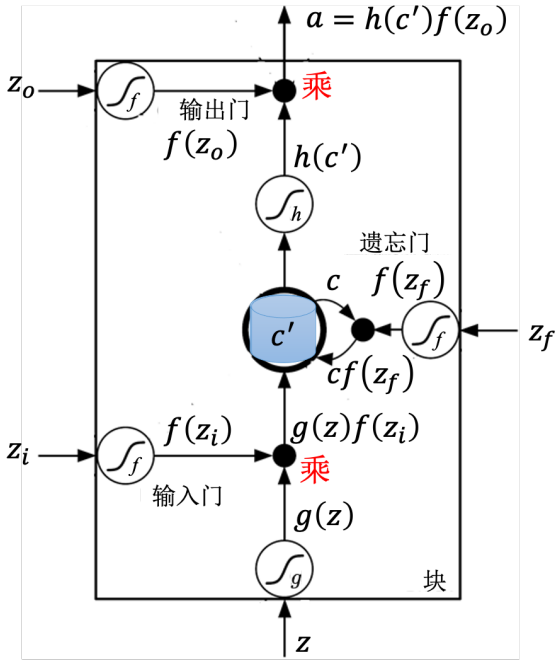


图 5.13　LSTM 记忆元示例

x_3 等于 1，这时候 7 会被输出。第七个维度的 x_2 的值为-1，记忆元里面的值会被洗掉变为 0。第八个维度的 x_2 的值为 1，所以把 6 存进去，因为 x_3 的值为 1，所以把 6 输出。

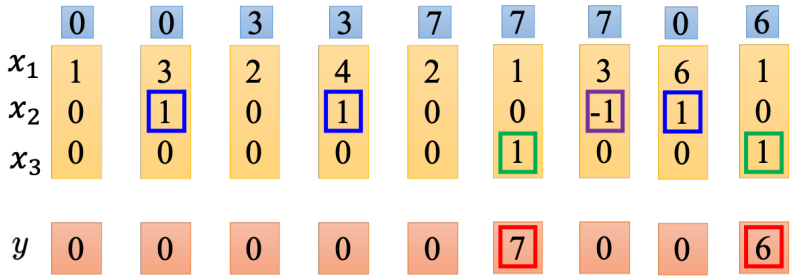


图 5.14　LSTM 示例

5.4.5　LSTM 运算示例

图 5.15 给出了 LSTM 实际的运算的例子。记忆元的四个输入标量是这样来的：输入的三维向量乘以线性变换（linear transform）后所得到的结果，x_1, x_2, x_3 乘以权重再加上偏置。假设这些值是已知的，在实际运算之前，先根据它的输入，分析下可能会得到的结果。底下这个外界传入的单元，x_1 乘以 1，其他的向量乘以 0，所以就直接把 x_1 当做输入。在输入门时，x_2 乘以 100，偏置乘以 -10。假设 x_2 是没有值的话，通常输入门是关闭的（偏置等于 -10）。因为 -10 通过 sigmoid 函数之后会接近 0，所以就代表是关闭的，若 x_2 的值大于 1 的话，结果会是一个正值，代表输入门会被打开。遗忘门通常会被打开的，因为其偏置等于 10，它平常会一直记得东西，只有当 x_2 的值为一个很大的负值时，才会把遗忘门关起来。输出门平常是被关闭的，因为偏置是一个很大的负值，若 x_3 有一个很大的正值的话，压过偏置把输出打

开。

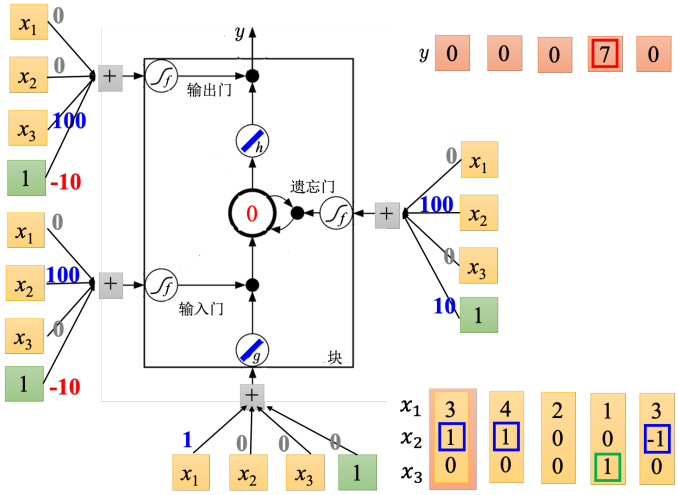


图 5.15　LSTM 运算示例

接下来，实际的输入一下看看。为了简化计算，假设 g 和 h 都是线性的。假设存到记忆元里面的初始值是 0，如图 5.17 所示，输入第一个向量 $[3,1,0]^{\mathrm{T}}$，输入这边 3*1=3，这边输入的是的值为 3。输入门这边 $(1*100-10\approx 1)$ 是被打开 (输入门约等于 1)。$(g(z)*f(z_i)=3)$。遗忘门 $(1*100+10\approx 1)$ 是被打开的 (遗忘门约等于 1)。0 *1+3=3$(c^{'}=g(z)f(z_i)+cf(z_f))$，所以存到记忆元里面的为 3。输出门 (-10) 是被关起来的，所以 3 无关通过，所以输出值为 0。

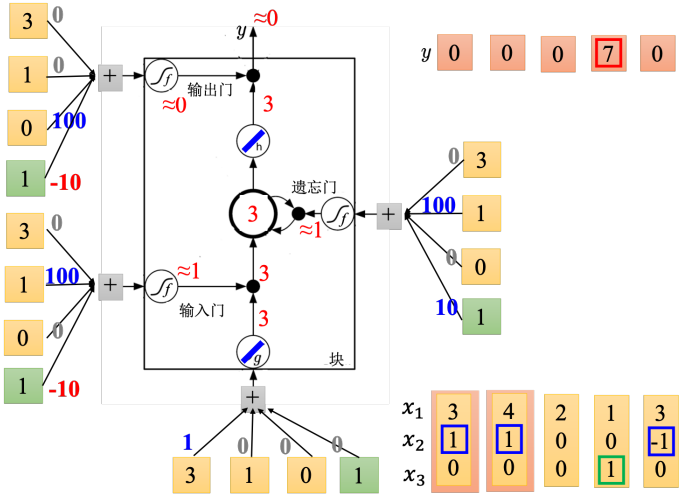


图 5.16　LSTM 运算示例：第 1 步

接下来输入 $[4,1,0]^{\mathrm{T}}$，如图 5.17 所示，传入输入的值为 4，输入门会被打开，遗忘门也会被打开，所以记忆元里面存的值等于 7（$3+4=7$），输出门仍然会被关闭的，所以 7 没有办法被输出，所以整个记忆元的输出为 0。

接下来输入 $[2,0,0]^{\mathrm{T}}$，如图 5.18 所示，传入输入的值为 2，输入门关闭（≈ 0），输入被输入门给挡住了（$0\times 2=0$），遗忘门打开（10）。原来记忆元里面的值还是 7（$1\times 7+0=7$）。输出门仍然为 0，所以没有办法输出，所以整个输出还是 0。

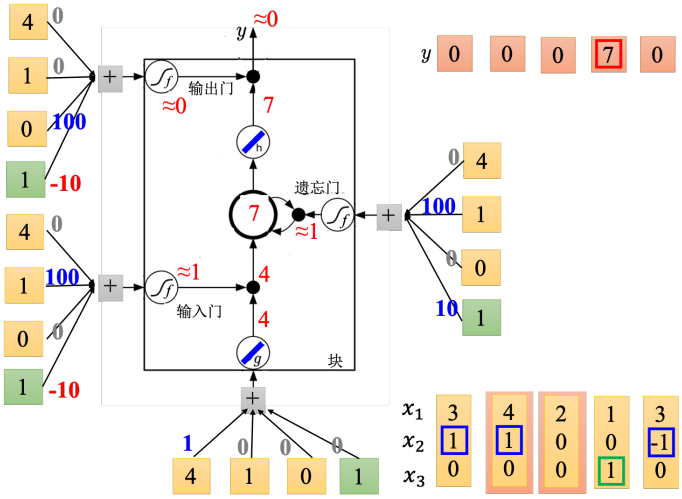


图 5.17　LSTM 运算示例：第 2 步

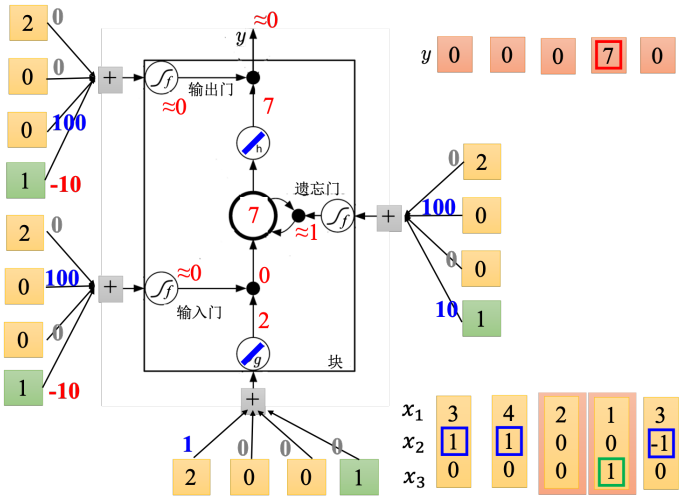


图 5.18　LSTM 运算示例：第 3 步

接下来输入 $[1,0,1]^{\mathrm{T}}$，如图 5.19 所示，传入输入的值为 1，输入门是关闭的，遗忘门是打开的，记忆元里面存的值不变，输出门被打开，整个输出为 7，记忆元里面存的 7 会被读取出来。

最后输入 $[3,-1,0]^{\mathrm{T}}$，如图 5.20 所示，传入输入的值为 3，输入门关闭，遗忘门关闭，记忆元里面的值会被洗掉变为 0，输出门关闭，所以整个输出为 0。

5.5　LSTM 原理

在原来的神经网络里面会有很多的神经元，我们会把输入乘以不同的权重当做不同神经元的输入，每一个神经元都是一个函数，输入一个值然后输出一个值。但是如果是 LSTM 的话，只要把 LSTM 想成是一个神经元。所以要用一个 LSTM 的神经元，其实就是原来简单的神经元换成 LSTM。

如图 5.22 所示，为了简化，假设隐藏层只有两个神经元，输入 x_1, x_2 会乘以不同的权重

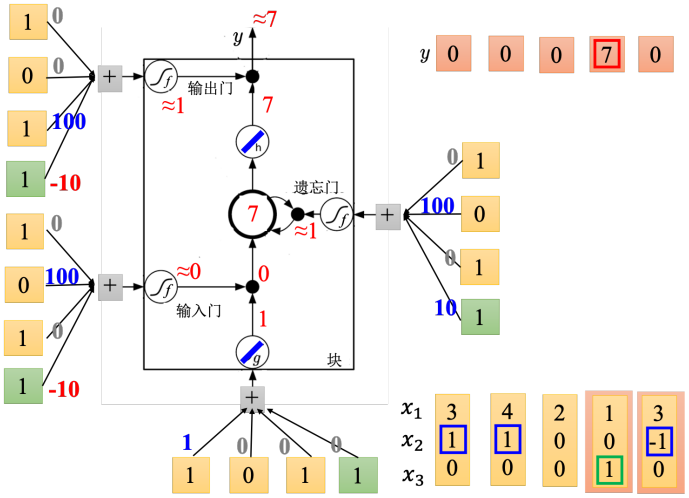


图 5.19　LSTM 运算示例：第 4 步

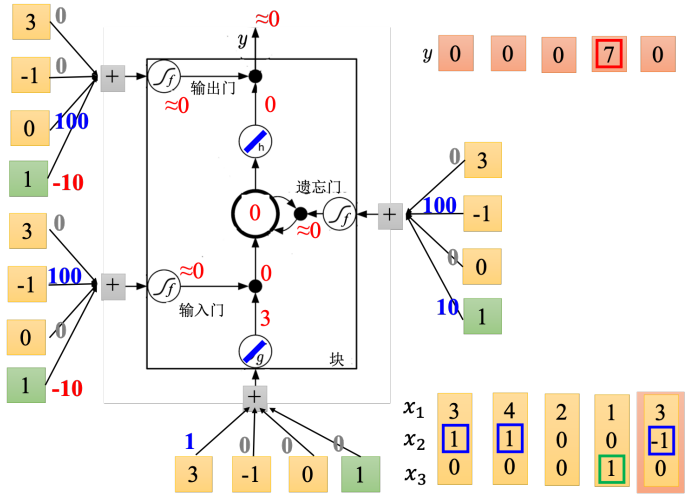


图 5.20　LSTM 运算示例：第 5 步

当做 LSTM 不同的输入。输入 (x_1, x_2) 会乘以不同的权重会去操控输出门，乘以不同的权重操控输入门，乘以不同的权重当做底下的输入，乘以不同的权重当做遗忘门。第二个 LSTM 也是一样的。所以 LSTM 是有四个输入跟一个输出，对于 LSTM 来说，这四个输入是不一样的。在原来的神经网络里是一个输入一个输出。在 LSTM 里面它需要四个输入，它才能产生一个输出。LSTM 因为需要四个输入，四个输入都是不一样，原来的一个神经元就只有一个输入和输出。假设用的神经元的数量跟 LSTM 是一样的，则 LSTM 需要的参数量是一般神经网络的四倍。

如图 5.23 所示，假设有一整排的 LSTM，这些 LSTM 里面的记忆元都存了一个值，把所有的值接起来就变成了向量，写为 c^{t-1}（一个值就代表一个维度）。现在在时间点 t，输入向量 $\boldsymbol{x}_t$，这个向量首先会乘上一矩阵（线性变换）变成一个向量 $\boldsymbol{z}$，向量 $\boldsymbol{z}$ 的维度就代表了操控每一个 LSTM 的输入。$\boldsymbol{z}$ 这个维度正好就是 LSTM 记忆元的数量。$\boldsymbol{z}$ 的第一维就丢给第一个单元。这个 $\boldsymbol{x}_t$ 会乘上另外的一个变换得到 $\boldsymbol{z}^i$，然后这个 $\boldsymbol{z}^i$ 的维度也跟单元的数量一

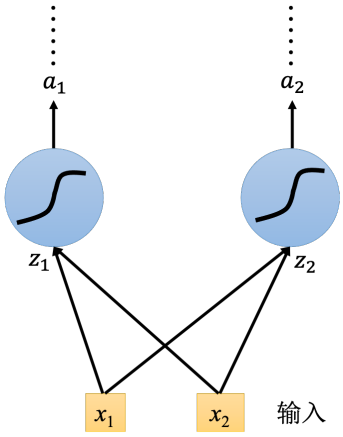


图 5.21　把 LSTM 想成一个神经元

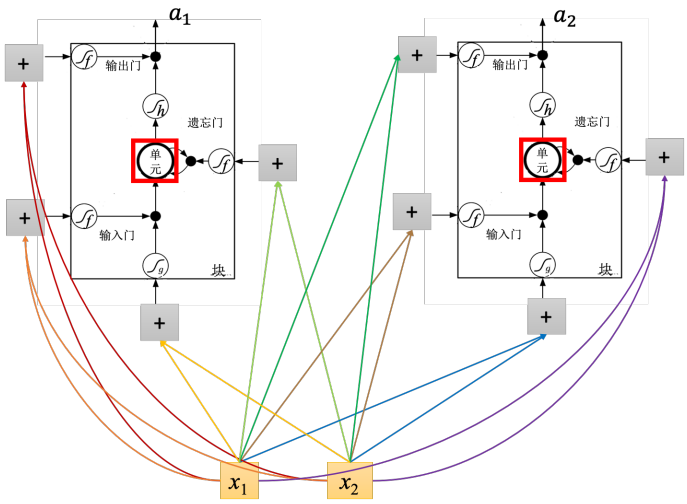


图 5.22　LSTM 需要 4 个输入

样，$\boldsymbol{z}^i$ 的每一个维度都会去操控输入门。遗忘门跟输出门也都是一样，不再赘述。所以我们把 $\boldsymbol{x}_t$ 乘以四个不同的变换得到四个不同的向量，四个向量的维度跟单元的数量一样，这四个向量合起来就会去操控这些记忆元运作。

如图 5.24 所示，输入分别就是 $\boldsymbol{z}, \boldsymbol{z}^i, \boldsymbol{z}^o, \boldsymbol{z}^f$(都是向量)，丢到单元里面的值其实是向量的一个维度，因为每一个单元输入的维度都是不一样的，所以每一个单元输入的值都会是不一样。所以单元是可以共同一起被运算的。$\boldsymbol{z}^i$ 通过激活函数跟 $\boldsymbol{z}$ 相乘，$\boldsymbol{z}^f$ 通过激活函数跟之前存在单元里面的值相乘，然后将 $\boldsymbol{z}$ 跟 $\boldsymbol{z}^i$ 相乘的值加上 $\boldsymbol{z}^f$ 跟 $\boldsymbol{c}^{t-1}$ 相乘的值，$\boldsymbol{z}^o$ 通过激活函数的结果输出，跟之前相加的结果再相乘，最后就得到了输出 $\boldsymbol{y}_t$。

之前那个相加以后的结果就是记忆元里面存放的值 $\boldsymbol{c}^t$，这个过程反复的进行，在下一个时间点输入 $\boldsymbol{x}_{t+1}$，把 $\boldsymbol{z}$ 跟输入门相乘，把遗忘门跟存在记忆元里面的值相乘，将前面两个值再相加起来，在乘上输出门的值，得到下一个时间点的输出 $\boldsymbol{y}_{t+1}$ 但这还不是 LSTM 的最终形态，真正的 LSTM 会把上一个时间的输出接进来，当做下一个时间的输入，即下一个时间点操控这些门的值不是只看那个时间点的输入 x_t ，还看前一个时间点的输出 $\boldsymbol{h}^t$ 。其实还不止这样，还会添加 peephole 连接，如图 5.25 所示。peephole 就是把存在记忆元里面的值也拉

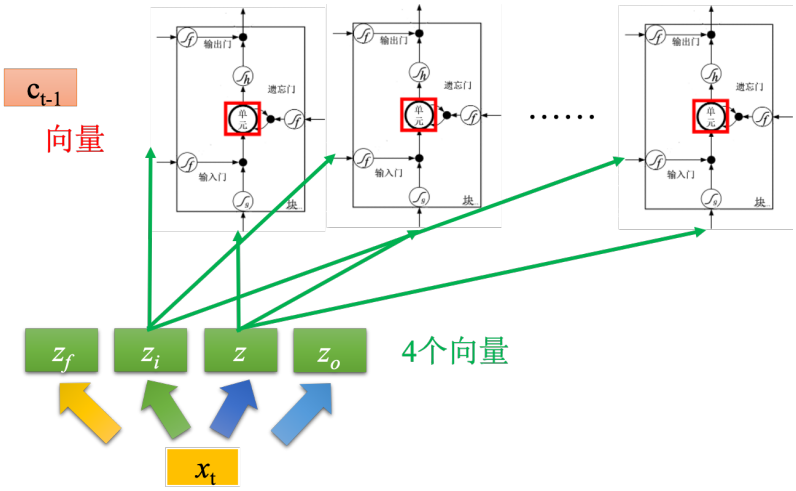


图 5.23　输入向量与记忆元的关系

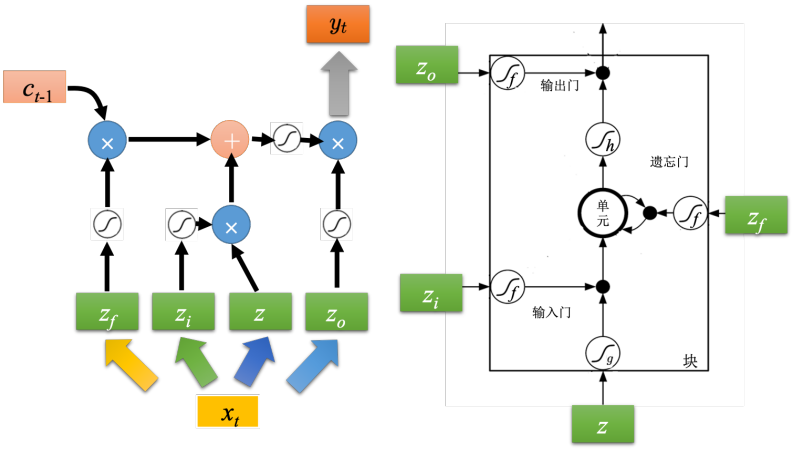


图 5.24　记忆元一起运算示例

过来。操控 LSTM 四个门的时候，同时考虑了 $\boldsymbol{x}_{t+1}, \boldsymbol{h}^t, \boldsymbol{c}^t$ ，把这三个向量并在一起乘上不同的变换得到四个不同的向量再去操控 LSTM。

LSTM 通常不会只有一层，若有五六层的话，如图 5.26 所示。一般做 RNN 的时候，其实指的就用 LSTM。

门控循环单元（Gated Recurrent Unit，GRU）是 LSTM 稍微简化的版本，它只有两个门。虽然少了一个门，但其性能跟 LSTM 差不多，少了 1/3 的参数，也是比较不容易过拟合。

5.6　RNN 学习方式

如果要做学习，需要定义一个**损失函数（loss function）**来评估模型的好坏，选一个参数要让损失最小。以槽填充为例，如图 5.27 所示，给定一些句子，要给句子一些标签，告诉机器说第一个单词它是属于 other 槽，“上海”是目的地槽，“on”属于 other 槽，“June”和“1st”属于时间槽。“抵达”丢到循环神经网络的时候，循环神经网络会得到一个输出 $\boldsymbol{y}_1$。接下来这个 $\boldsymbol{y}_1$ 会看它的参考向量（reference vector）算它的交叉熵。我们会期望如果丢进去的是“抵达”，其

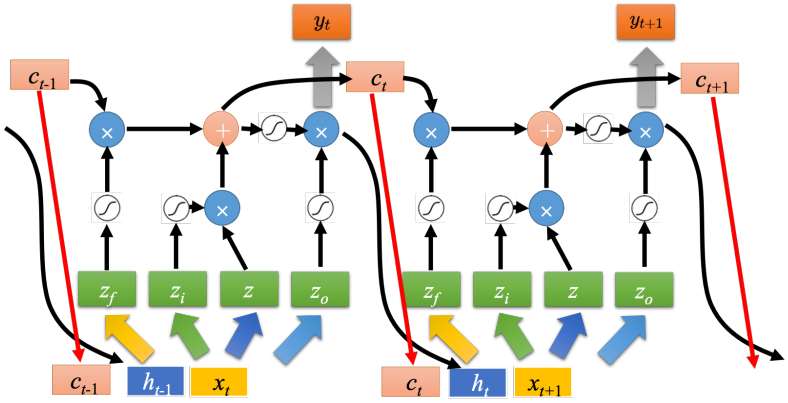


图 5.25 peephole 连接

参考向量应该对应到 other 槽的维度，其他为 0，这个参考向量的长度就是槽的数量。如果有四十个槽，参考向量的维度就是 40。输入的这个单词对应到 other 槽的话，对应到 other 槽维度为 1, 其它为 0。把“上海”丢进去之后，因为“上海”属于目的地槽，希望说把 $\boldsymbol{x}_2$ 丢进去的话，$\boldsymbol{y}_2$ 要跟参考向量距离越近越好。那 $\boldsymbol{y}_2$ 的参考向量是对应到目的地槽是 1，其它为 0。注意，在丢 $\boldsymbol{x}_2$ 之前，一定要丢 $\boldsymbol{x}_1$（在丢“上海”之前先把“抵达”丢进去），不然就不知道存到记忆元里面的值是多少。所以在训练的时候，不能够把这些单词序列打散来看，单词序列仍然要当做一个整体来看。把“on”丢进去，参考向量对应的 other 的维度是 1，其它是 0. RNN 的损失函数输出和参考向量的交叉熵的和就是要最小化的对象。

有了这个损失函数以后，对于训练也是用梯度下降来做。也就是现在定义出了损失函数 L，要更新这个神经网络里面的某个参数 w，就是计算对 w 的偏微分，偏微分计算出来以后，就用梯度下降的方法去更新里面的参数。梯度下降用在前馈神经网络里面我们要用一个有效率的算法称为反向传播。循环神经网络里面，为了要计算方便，提出了反向传播的进阶版，即**随时间反向传播 (BackPropagation Through Time，BPTT)**。BPTT 跟反向传播其实是很类似的，只是循环神经网络它是在时间序列上运作，所以 BPTT 它要考虑时间上的信息，如图 5.28 所示。

RNN 的训练是比较困难的，如图 5.29 所示。一般而言，在做训练的时候，期待学习曲线是像蓝色这条线，这边的纵轴是总损失（total loss），横轴是回合的数量，我们会希望随着回合的数量越来越多，随着参数不断的更新，损失会慢慢地下降，最后趋向收敛。但是不幸的是，在训练循环神经网络的时候，有时候会看到绿色这条线。如果第一次训练循环神经网络，绿色学习曲线非常剧烈的抖动，然后抖到某个地方，我们会觉得这程序有 bug。

如图 5.30 所示，RNN 的误差表面是总损失的变化是非常陡峭的或崎岖的。误差表面有一些地方非常平坦，一些地方非常陡峭。纵轴是总损失，x 和 y 轴代表是两个参数。这样会造成什么样的问题呢？假设我们从橙色的点当做初始点，用梯度下降开始调整参数，更新参数，可能会跳过一个悬崖，这时候损失会突然爆长，损失会非常上下剧烈的震荡。有时候我们可能会遇到更惨的状况，就是以正好我们一脚踩到这个悬崖上，会发生这样的事情，因为在悬崖上的梯度很大，之前的梯度会很小，所以措手不及，因为之前梯度很小，所以可能把学习率调的比较大。很大的梯度乘上很大的学习率结果参数就更新很多，整个参数就飞出去了。裁剪（clipping）可以解决该问题，当梯度大于某一个阈值的时候，不要让它超过那个阈值，当梯度大于 15 时，让梯度等于 15 结束。因为梯度不会太大，所以我们要做裁剪的时候，就算是踩

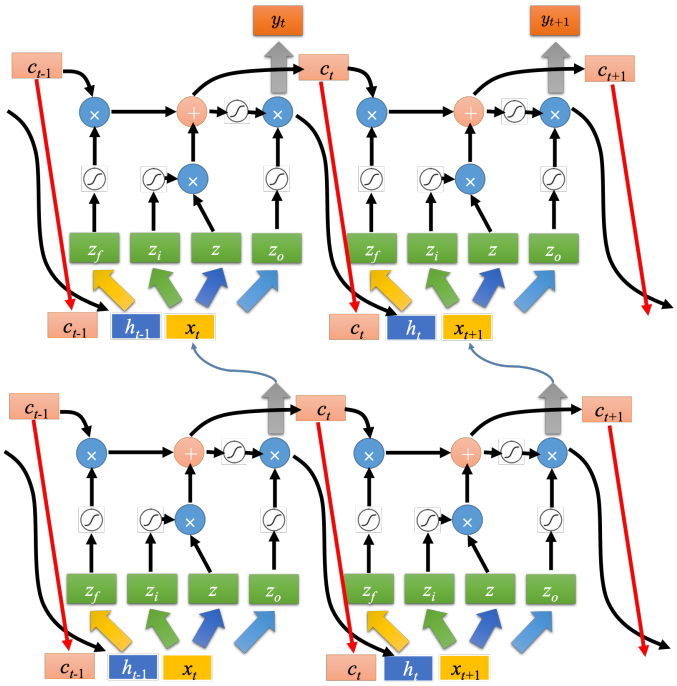


图 5.26　多层 LSTM

着这个悬崖上，也不飞出来，会飞到一个比较近的地方，这样还可以继续做 RNN 的训练。

之前讲过 ReLU 激活函数的时候，**梯度消失（vanishing gradient）**来源于 Sigmoid 函数。但 RNN 会有很平滑的误差表面不是来自于梯度消失。把 Sigmoid 函数换成 ReLU，其实在 RNN 性能通常是比较差的，所以激活函数并不是关键点。

有更直观的方法来知道一个梯度的大小，可以把某一个参数做小小的变化，看它对网络输出的变化有多大，就可以测出这个参数的梯度大小，如图 5.31 所示。举一个很简单的例子，只有一个神经元，这个神经元是线性的。输入没有偏置，输入的权重是 1，输出的权重也是 1，转移的权重是 w。也就是说从记忆元接到神经元的输入的权重是 w。

如图 5.32 所示，假设给神经网络的输入是 $[1,0,0,0]^{\mathrm{T}}$，比如神经网络在最后一个时间点（1000 个输出值是 w^{999}）。假设 w 是要学习的参数，我们想要知道它的梯度，所以是改变 w 的值时候，对神经元的输出有多大的影响。假设 $w=1$，$y_{1000}=1$，假设 $w=1.01$，$y_{1000}\approx 20000$，w 有一点小小的变化，会对它的输出影响是非常大的。所以 w 有很大的梯度。有很大的梯度也并没有，把学习率设小一点就好了。但把 w 设为 0.99，那 $y_{1000}\approx 0$。如果把 w 设为 0.01，$y_{1000}\approx 0$。也就是说在 1 的这个地方有很大的梯度，但是在 0.99 这个地方就突然变得非常小，这个时候需要一个很大的学习率。设置学习率很麻烦，误差表面很崎岖，梯度是时大时小的，在非常小的区域内，梯度有很多的变化。从这个例子可以看出，RNN 训练的问题其实来自它把同样的东西在转移的时候，在时间按时间的时候，反复使用。所以 w 只要一有变化，它完全由可能没有造成任何影响，一旦造成影响，影响很大，梯度会很大或很小。所以 RNN 不好训练的原因不是来自激活函数而是来自于它有时间序列同样的权重在不同的时间点被反复的使用。

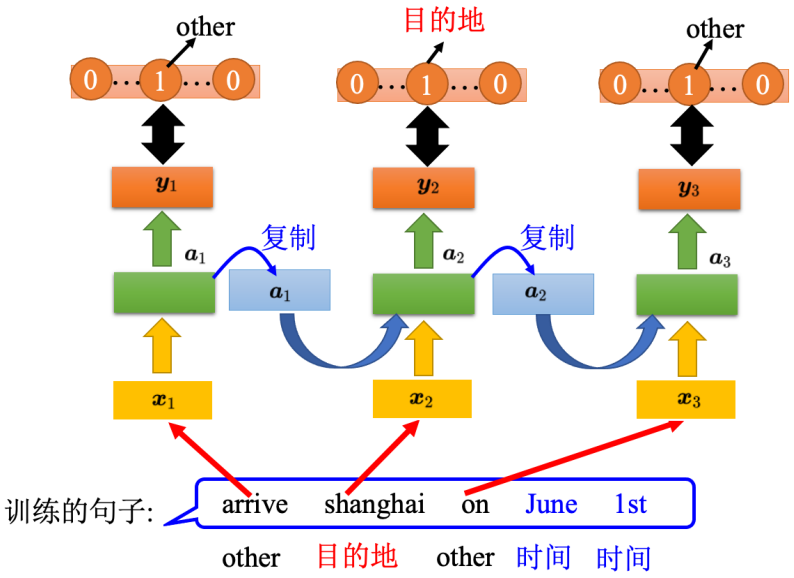


图 5.27 RNN 计算损失示意

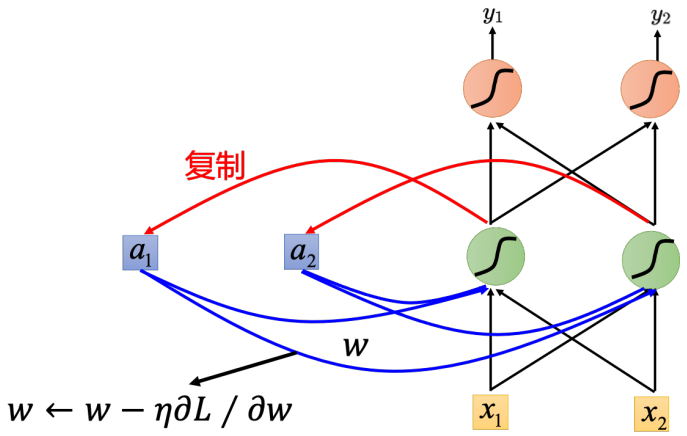


图 5.28 随时间反向传播

5.7 如何解决 RNN 梯度消失或者爆炸

有什么样的技巧可以解决这个问题呢？广泛被使用的技巧是 LSTM，LSTM 可以让误差表面不要那么崎岖。它会把那些平坦的地方拿掉，解决梯度消失的问题，不会解决**梯度爆炸 (gradient exploding)** 的问题。有些地方还是非常的崎岖的，有些地方仍然是变化非常剧烈的，但是不会有特别平坦的地方。如果做 LSTM 时，大部分地方变化的很剧烈，所以做 LSTM 的时候，可以把学习率设置的小一点，保证在学习率很小的情况下进行训练。

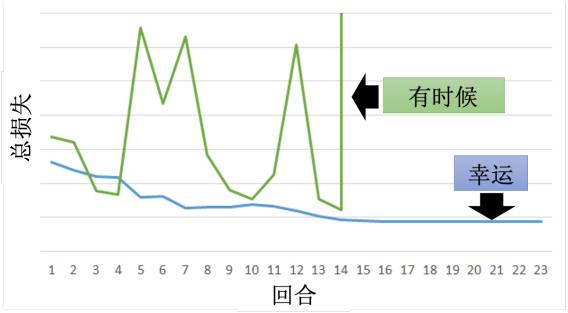


图 5.29　训练 RNN 时的学习曲线

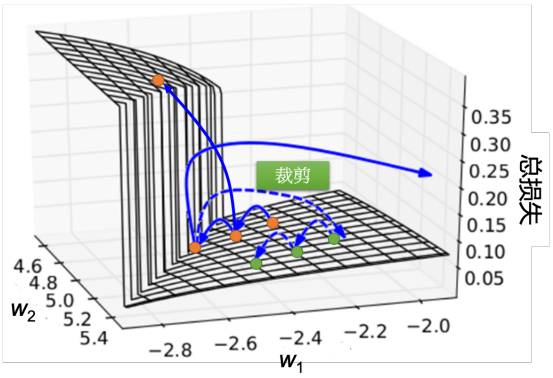


图 5.30　RNN 训练中的裁剪技巧

Q: 为什么 LSTM 可以解决梯度消失的问题，可以避免梯度特别小呢？为什么把 RNN 换成 LSTM?。

A：LSTM 可以处理梯度消失的问题。用这边的式子回答看看。RNN 跟 LSTM 在面对记忆元的时候，它处理的操作其实是不一样的。在 RNN 里面，在每一个时间点，神经元的输出都要记忆元里面去，记忆元里面的值都是会被覆盖掉。但是在 LSTM 里面不一样，它是把原来记忆元里面的值乘上一个值再把输入的值加起来放到单元里面。所以它的记忆和输入是相加的。LSTM 和 RNN 不同的是，如果权重可以影响到记忆元里面的值，一旦发生影响会永远都存在。而 RNN 在每个时间点的值都会被格式化掉，所以只要这个影响被格式化掉它就消失了。但是在 LSTM 里面，一旦对记忆元造成影响，影响一直会被留着，除非遗忘门要把记忆元的值洗掉。不然记忆元一旦有改变，只会把新的东西加进来，不会把原来的值洗掉，所以它不会有梯度消失的问题。

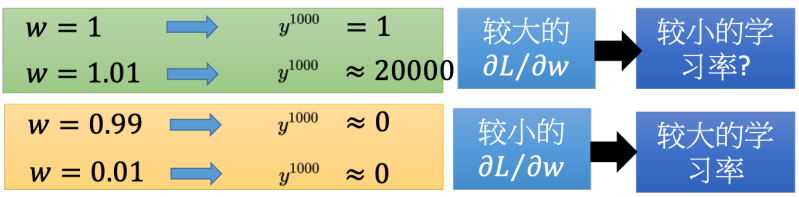


图 5.31　参数变化对网络输出的影响

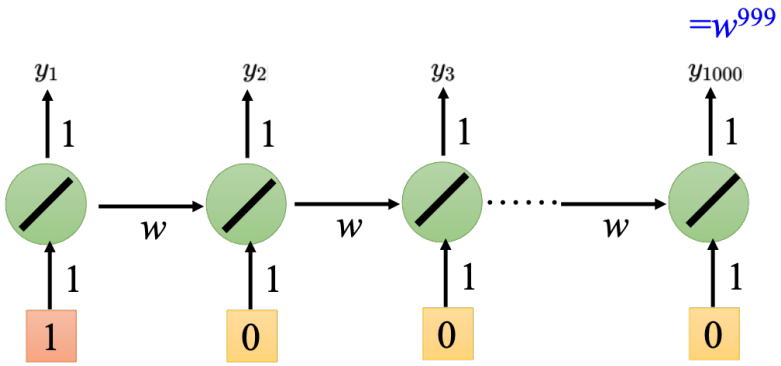


图 5.32 RNN 难以训练的原因

遗忘门可能会把记忆元的值洗掉。其实 LSTM 的第一个版本其实就是为了解决梯度消失的问题，所以它是没有遗忘门，遗忘门是后来才加上去的。甚至有个传言是：在训练 LSTM 的时候，要给遗忘门特别大的偏置，确保遗忘门在多数的情况下都是开启的，只要少数的情况是关闭的。

有另外一个版本用门操控记忆元，叫做 GRU，LSTM 有三个门，而 GRU 有两个门，所以 GRU 需要的参数是比较少的。因为它需要的参数量比较少，所以它在训练的时候是比较鲁棒的。如果训练 LSTM 的时候，过拟合的情况很严重，可以试下 GRU。GRU 的精神就是：旧的不去，新的不来。它会把输入门跟遗忘门联动起来，也就是说当输入门打开的时候，遗忘门会自动的关闭 (格式化存在记忆元里面的值)，当遗忘门没有要格式化里面的值，输入门就会被关起来。也就是要把记忆元里面的值清掉，才能把新的值放进来。

其实还有其他技术可以处理梯度消失的问题。比如顺时针循环神经网络（clockwise RNN）[1] 或结构约束的循环网络（Structurally Constrained Recurrent Network，SCRN）[2] 等等。

论文“A Simple Way to Initialize Recurrent Networks of Rectified Linear Units”[3] 采用了不同的做法。一般的 RNN 用单位矩阵（identity matrix）来初始化转移权重和 ReLU 激活函数可以得到很好的性能。刚才不是说用 ReLU 的性能会比较呀，如果用一般训练的方法随机初始化权重，ReLU 跟 sigmoid 函数来比的话，sigmoid 性能会比较好。但是使用了单位矩阵，这时候用 ReLU 性能会比较好。

5.8 RNN 其他应用

槽填充的例子中假设输入跟输出的数量是一样的，也就是说输入有几个单词，我们就给每一个单词槽标签，RNN 可以做到更复杂的事情。

5.8.1 多对一序列

比如输入是一个序列，输出是一个向量。**情感分析（sentiment analysis）**是典型的应用，如图 5.33 所示，某家公司想要知道，他们的产品在网上的评价是正面的还是负面的。他们可能会写一个爬虫，把跟他们产品有关的文章都爬下来。那这一篇一篇的看太累了，所以可以用一个机器学习的方法学习一个**分类器（classifier）**来判断文档的正、负面。或者在电影上，情感分析就是给机器看很多的文章，机器要自动判断哪些文章是正类，哪些文章是负类。

机器可以学习一个循环神经网络，输入是字符序列，循环神经网络把这个序列读过一遍。在最后一个时间点，把隐藏层拿出来，在通过几个变换，就可以得到最后的情感分析。

情感分析是一个分类问题，但是因为输入是序列，所以用 RNN 来处理。

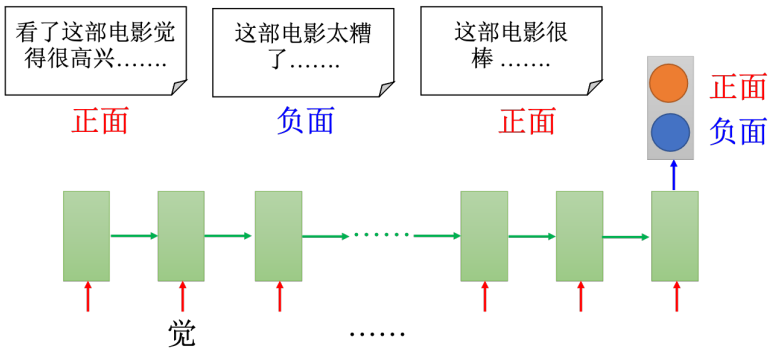


图 5.33　情感分析示例

用 RNN 来作关键术语抽取（key term extraction）。关键术语抽取意思就是说给机器看一个文章，机器要预测出这篇文章有哪些关键单词。如图 5.34 所示，如果能够收集到一些训练数据 (一些文档，这些文档都有标签，哪些单词是对应的，那就可以直接训练一个 RNN)，那这个 RNN 把文档当做输入，通过嵌入层（embedding layer），用 RNN 把这个文档读过一次，把出现在最后一个时间点的输出拿过来做注意力，可以把这样的信息抽出来再丢到前馈神经网络得到最后的输出。

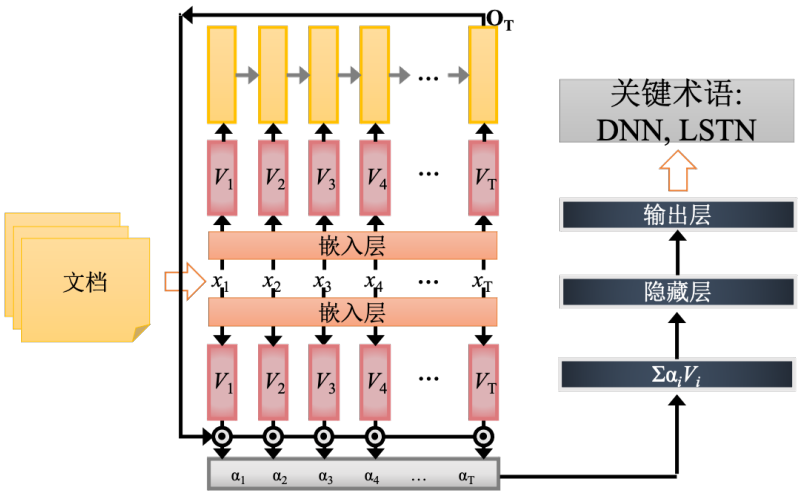


图 5.34　关键术语抽取

5.8.2　多对多序列

RNN 也可以处理多对多的问题，比如输入和输出都是序列，但输出序列比输入序列短。如图 5.35 所示，在语音识别这个任务里面输入是声音序列，一句话就是一段声音信号。一般

处理声音信号的方式就是在这个声音信号里面，每隔一小段时间，就把它用向量来表示。这个一小段时间是很短的（比如 0.01 秒）。那输出序列是字符序列。

如果是原来的 RNN（槽填充的那个 RNN），把这一串输入丢进去，它充其量只能做到，告诉我们每一个向量对应到哪一个字符。加入说中文的语音识别的话，那输出目标理论上就是这个世界上所有可能中文的单词，常用的可能是八千个，RNN 分类器的数量可能就是八千个。虽然很大，但也是没有办法做的。但是充其量只能做到说：每一个向量属于一个字符。每一个输入对应的时间间隔是很小的（0.01 秒），所以通常是好多个向量对应到同一个字符。所以识别结果为“好好好棒棒棒棒棒”，这不是语音识别的结果。有一招叫做修剪（trimming），即把重复的东西拿掉，就变成“好棒”。这样会有一个严重的问题，因为它没有识别“好棒棒”。

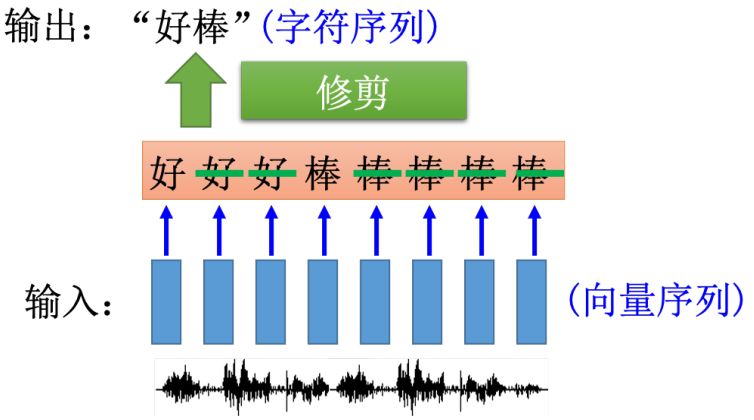


图 5.35　语音识别示例

需要把“好棒”跟“好棒棒”分开来，怎么办，有一招叫做“CTC”，如图 5.36 所示，在输出时候，不只是输出所有中文的字符，还可以输出一个符号”null”，其代表没有任何东西。所以输入一段声音特征序列，它的输出是“好 null null 棒 null null null null”，然后把“null”的部分拿掉，它就变成“好棒”。如果我们输入另外一个序列，它的输出是“好 null null 棒 null 棒 null null”，然后把“null”拿掉，所以它的输出就是“好棒棒”。这样就可以解决叠字的问题了。

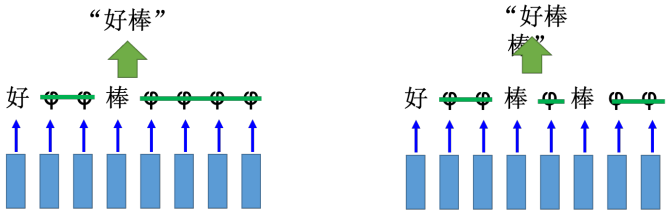


图 5.36　CTC 技巧

CTC 怎么做训练呢？如图 5.37 所示，CTC 在做训练的时候，手上的训练数据就会告诉我们说，这一串声音特征对应到这一串字符序列，但它不会告诉我们说“好”是对应第几个字符到第几个字符。这时候要穷举所有可能的对齐，简单来说，我们不知道“好”对应到那几个字符，“棒”对应到哪几个字符。假设我们所有的状况都是可能的。可能第一个是“好 null 棒 null null null”，可能是“好 null null 棒 null null”，也可能是“好 null null null 棒 null”。假设全部都是对的，一起训练。穷举所有的可能，可能性太多了。

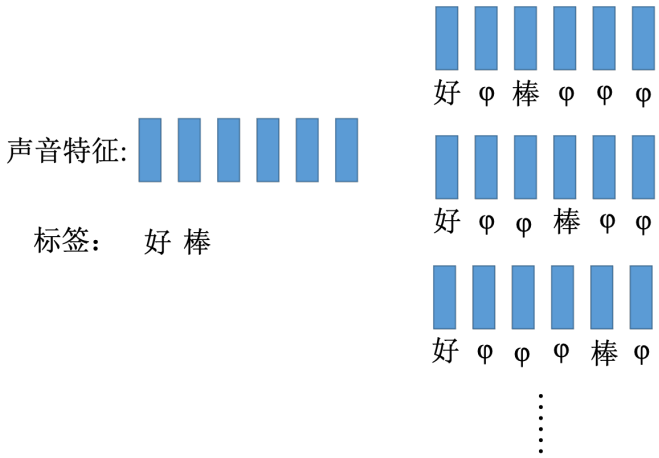


图 5.37 CTC 训练

在做英文识别的时候，RNN 输出目标就是字符（英文的字母 + 空白）。直接输出字母，然后如果字和字之间有边界，就自动有空白。如图 5.38 所示，第一帧是输出 H，第二帧是输出 null，第三帧是输出 null，第四帧是输出 I 等等。如果我们看到输出是这样子话，最后把“null”的地方拿掉，这句话的识别结果就是“HIS FRIEND’S”。我们不需要告诉机器说：”HIS” 是一个单词，“FRIEND’s”是一个单词，机器通过训练数据会自己学到这件事情。如果用 CTC 来做语音识别，就算是有某一个单词在训练数据中从来没有出现过（比如英文中的人名或地名），机器也是有机会把它识别出来。

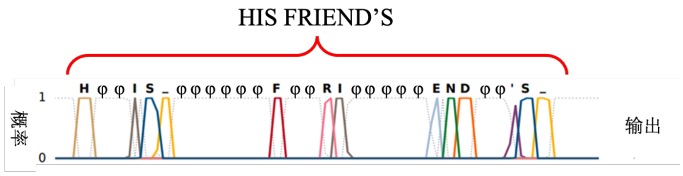


图 5.38 CTC 语音识别示例

5.8.3 序列到序列

另一个 RNN 的应用是**序列到序列（Sequence-to-Sequence，Seq2Seq）**学习，在序列到序列学习里面,RNN 的输入跟输出都是序列 (但是两者的长度是不一样的)。刚在在 CTC 时，输入比较长，输出比较短。在这边我们要考虑的是不确定输入跟输出谁比较长谁比较短。比如机器翻译（machine translation），输入英文单词序列把它翻译成中文的字符序列。英文和中文序列的长短是未知的。

假如输入机器学习，然后用 RNN 读过去，然后在最后一个时间点，这个记忆元里面就存了所有输入序列的信息，如图 5.39 所示。

接下来，我们让机器吐一个字符（“机”），就让它输出下一个字符，把之前的输出出来的字符当做输入，再把记忆元里面的值读进来，它就会输出“器”。那这个“机”怎么接到这个地方呢，有很多支支节节的技巧。在下一个时间输入“器”，输出“学”，然后输出“习”，然后一直输出下去，如图 5.40 所示。

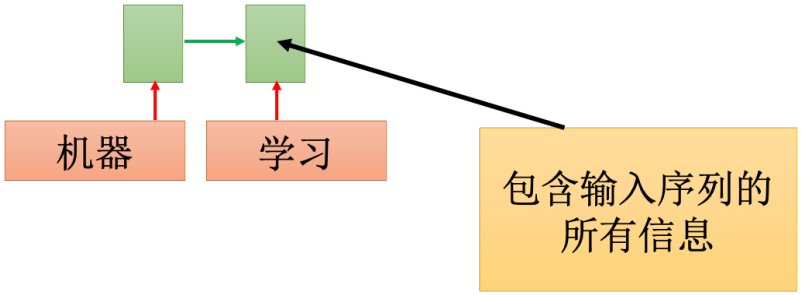


图 5.39　记忆元存储所有序列信息

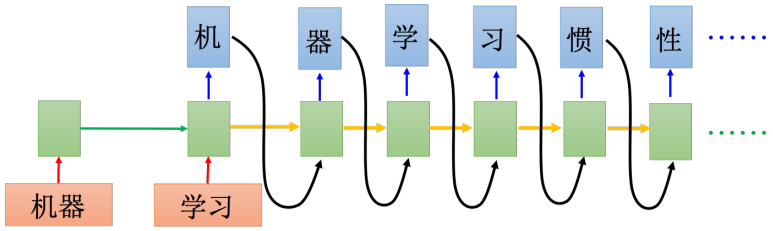


图 5.40　RNN 会一直生成字符

要怎么阻止让它产生单词呢？要多加一个符号“断”，所以机器的输出不是只有字符，它还有一个可能输出“断”。如果“习”后面是符号“===”（断）的话，就停下来了，如图 5.41 所示。这是训练的起来的。序列到序列学习，假设做翻译，原来是输入某种语言的文字，翻译成另外一种语言的文字。有没有可能直接输入某种语言的声音信号，输出另外一种语言的文字呢？我们完全不做语音识别。比如把英文翻译成中文，收集一大堆英文的句子，看看它对应的中文翻译。我们完全不要做语音识别，直接把英文的声音信号丢到这个模型里面去，看它能不能输出正确的中文。这一招居然是行得通的。假设要把闽南语转成英文，但是闽南语的语音识别系统不好做，因为闽南语根本就没有标准文字系统。如果训练闽南语转英文语音识别系统的时候，只需要收集闽南语的声音信号跟它的英文翻译就可以了，不需要闽南语语音识别的结果，也不需要知道闽南语的文字。

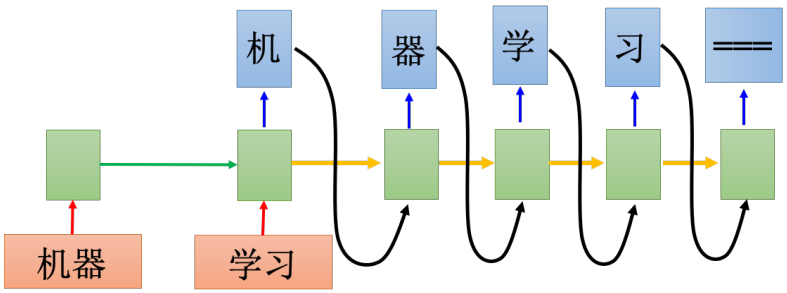


图 5.41　添加截止符号

序列到序列的技术也被用到句法解析（syntactic parsing）。句法解析，让机器看一个句子，得到句子结构树。如图 5.42 所示，只要把树状图描述成一个序列，比如：“John has a dog.”，序列到序列学习直接学习一个序列到序列模型，其输出直接就是句法解析树，这个是可以训练的起来的。LSTM 的输出的序列也是符合文法结构，左、右括号都有。

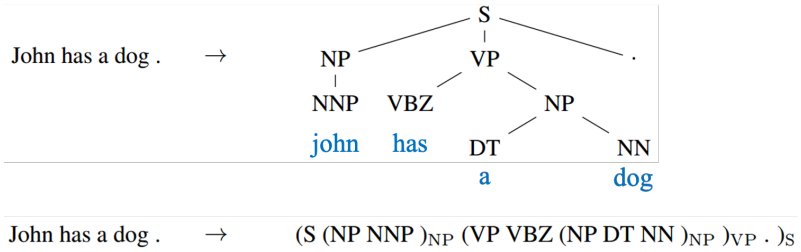


图 5.42 句法解析示例

要将一个文档表示成一个向量，如图 5.43 所示，往往会用**词袋（Bag-of-Words，BoW）**的方法，用这个方法的时候，往往会忽略掉单词顺序信息。举例来说，有一个单词序列是“white blood cells destroying an infection”，另外一个单词序列是：“an infection destroying white blood cells”，这两句话的意思完全是相反的。但是我们用词袋的方法来描述的话，他们的词袋完全是一样的。它们里面有完全一摸一样的六个单词，因为单词的顺序是不一样的，所以他们的意思一个变成正面的，一个变成负面的，他们的意思是很不一样的。

可以用序列到序列自编码器这种做法来考虑单词序列顺序的情况下，把一个文档变成一个向量。

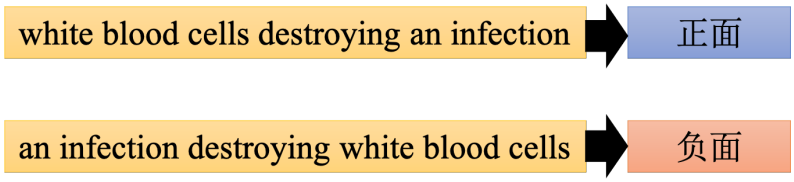


图 5.43 文档转成向量示例

参考文献

[1] KOUTNIK J, GREFF K, GOMEZ F, et al. A clockwork rnn[C]//International conference on machine learning. PMLR, 2014: 1863-1871.

[2] MIKOLOV T, JOULIN A, CHOPRA S, et al. Learning longer memory in recurrent neural networks[C]//ICLR. 2015.

[3] LE Q V, JAITLY N, HINTON G E. A simple way to initialize recurrent networks of rectified linear units[J]. arXiv preprint arXiv:1504.00941, 2015.

第 6 章 自注意力机制

讲完了卷积神经网络以后，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 6.1 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。

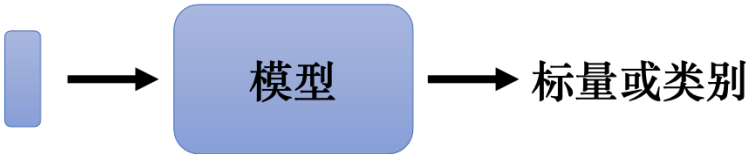


图 6.1 输入是一个向量

6.1 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 6.2 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 6.2 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，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，创建一个十万维的向量，每一个维度对应到一个词汇，如式 (6.1) 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。

$$\begin{aligned} \text{apple} &= [1,0,0,0,0,\ldots\ldots] \\ \text{bag} &= [0,1,0,0,0,\ldots\ldots] \\ \text{cat} &= [0,0,1,0,0,\ldots\ldots] \\ \text{dog} &= [0,0,0,1,0,\ldots\ldots] \\ \text{elephant} &= [0,0,0,0,1,\ldots\ldots] \end{aligned} \tag{6.1}$$

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 6.3 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。

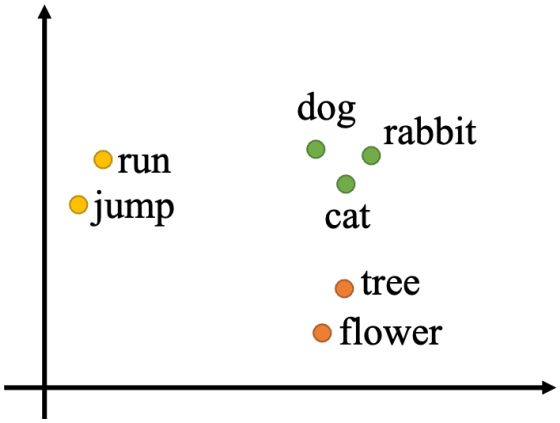


图 6.3　词嵌入

接下来举一些把一个向量的序列当做输入的例子。如图 6.4 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。他们尝试了大量可能的值，这样得到的结果往往最理想。

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。

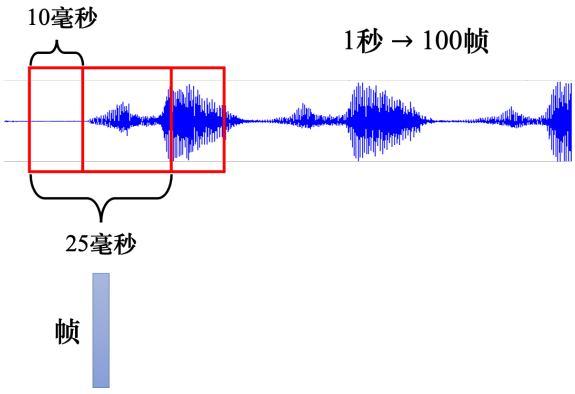


图 6.4　语音处理

一个图（graph）也是一堆向量。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。

药物发现（drug discovery）跟图有关，如图 6.5 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 (6.2) 所示。

$$\begin{aligned} \mathrm{H} &= [1,0,0,0,0,\ldots\ldots] \\ \mathrm{C} &= [0,1,0,0,0,\ldots\ldots] \\ \mathrm{O} &= [0,0,1,0,0,\ldots\ldots] \end{aligned} \tag{6.2}$$

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。

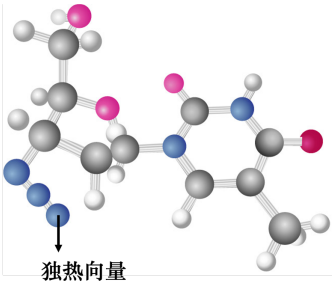


图 6.5　药物发现【需要重绘，版权问题】

6.1.1　类型 1：输入与输出数量相同

模型的输入是一组向量，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 6.6 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 6.6　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 6.7 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。

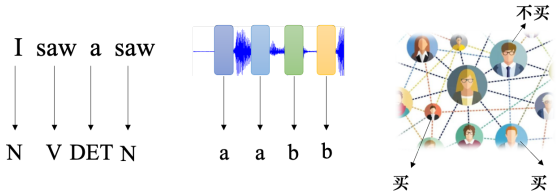


图 6.7　类型 1 应用的例子[1]

6.1.2　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 6.8 所示，整个序列只需要输出一个标签就好。

图 6.8　类似 2：输入是一个序列，输出是一个标签

举例而言，如图 6.9 所示，输入是文字，比如情感分析。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，想要知道网友的评价，但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。

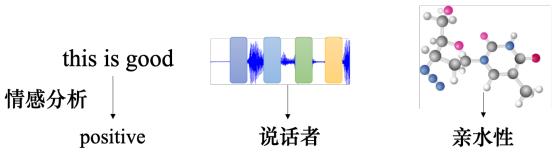


图 6.9　类型 2 的应用例子（亲水性这张图需要重绘，版权问题）

6.1.3　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 6.10 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 6.10 类型 3：序列到序列任务

6.2 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。如图 6.11 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。有没有可能让全连接网络考虑更

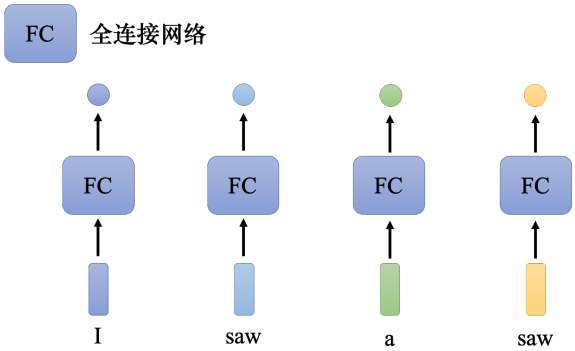


图 6.11 序列标注

多的信息，比如上下文的信息呢？这是有可能的，如图 6.12 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。

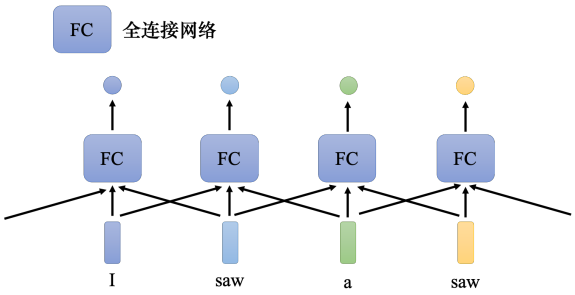


图 6.12 考虑上下文

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 6.13 所示。但是这种的方法还是有极限的，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。

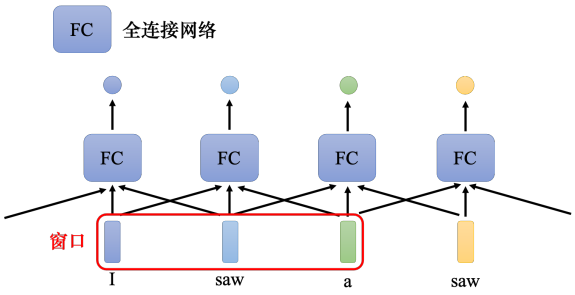


图 6.13　使用窗口来考虑上下文

自注意力模型的运作方式如图 6.14 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 6.14 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。

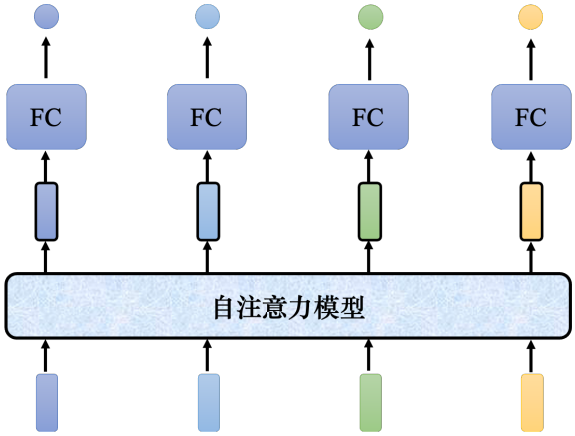


图 6.14　自注意力模型的运作方式

自注意力模型不是只能用一次，可以叠加很多次。如图 6.15 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。其中最重要的模块是自注意力，就像变形金刚的火种源。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。

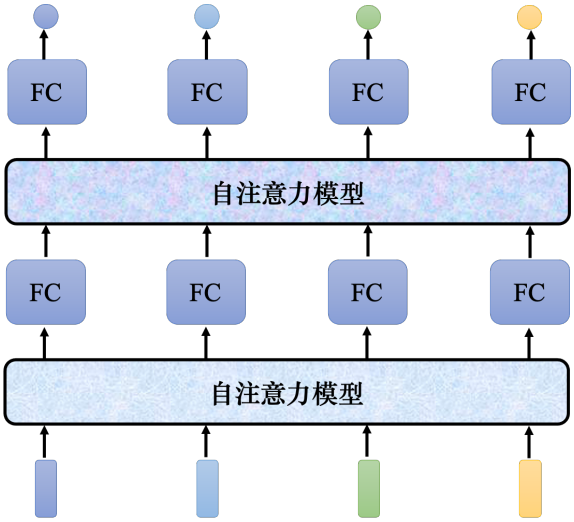


图 6.15　自注意力模型与全连接网络的叠加使用

自注意力模型的运作过程如图 6.16 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的输出，所以不用 $\boldsymbol{x}$ 来表示它，而用 $\boldsymbol{a}$ 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 $\boldsymbol{a}$，自注意力要输出一组向量 $\boldsymbol{b}$，每个 $\boldsymbol{b}$ 都是考虑了所有的 $\boldsymbol{a}$ 以后才生成出来的。$\boldsymbol{b}^1$、$\boldsymbol{b}^2$、$\boldsymbol{b}^3$、$\boldsymbol{b}^4$ 是考虑整个输入的序列 $\boldsymbol{a}^1$、$\boldsymbol{a}^2$、$\boldsymbol{a}^3$、$\boldsymbol{a}^4$ 才产生出来的。

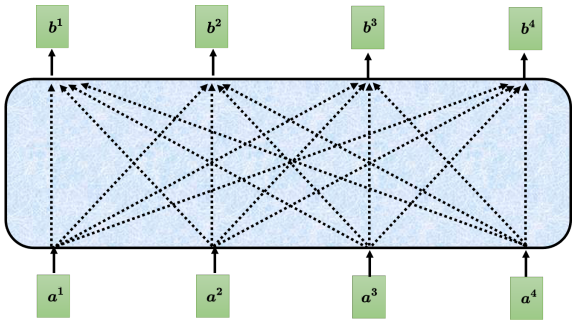


图 6.16　自注意力模型的运作方式

接下来介绍下向量 $\boldsymbol{b}^1$ 产生的过程，了解产生向量 $\boldsymbol{b}^1$ 的过程后，剩下向量 $\boldsymbol{b}^2$、$\boldsymbol{b}^3$、$\boldsymbol{b}^4$ 产生的过程以此类推。怎么产生向量 $\boldsymbol{b}^1$ 呢？如图 6.17 所示，第一个步骤是根据 $\boldsymbol{a}^1$ 找出输入序列里面跟 $\boldsymbol{a}^1$ 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 $\boldsymbol{a}^1$ 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 $\boldsymbol{a}^1$ 是哪一个标签是有关系的。每一个向量跟 $\boldsymbol{a}^1$ 的关联的程度可以用数值 α 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 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^4$，它怎么计算出一个数值 α 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 α，α 可以当做两个向量的关联的程度。怎么计算 α？比较常见的做法是用点积（dot product）。如图 6.18(a) 所示，把输

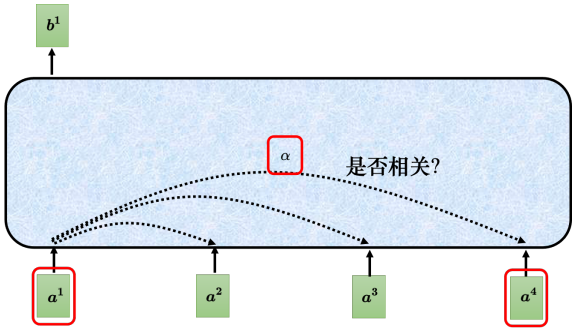
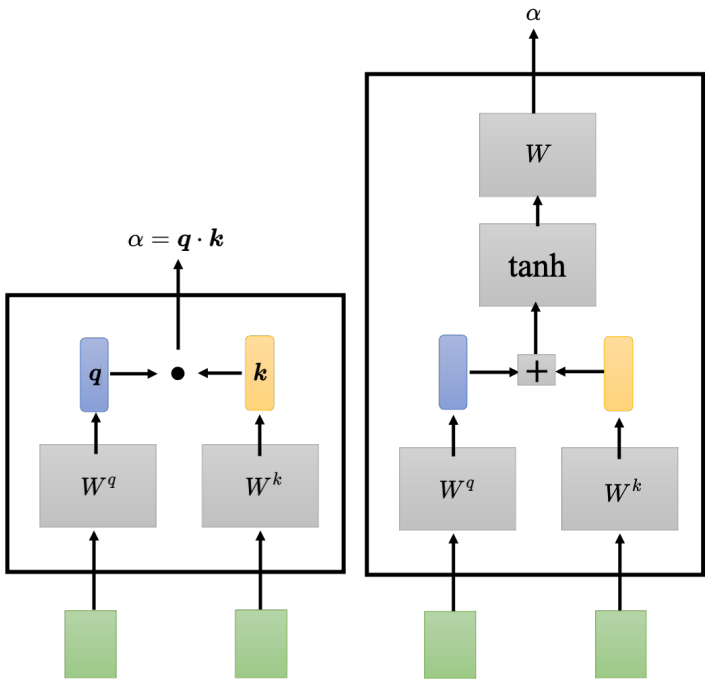


图 6.17　向量 $\boldsymbol{b}^1$ 产生的过程

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 $\boldsymbol{W^q}$，右边这个向量乘上矩阵 $\boldsymbol{W^k}$，得到两个向量 $\boldsymbol{q}$ 跟 $\boldsymbol{k}$，再把 $\boldsymbol{q}$ 跟 $\boldsymbol{k}$ 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）α，这是一种计算 α 的方式。

其实还有其他的计算方式，如图 6.18(b) 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 $\boldsymbol{q}$ 和 $\boldsymbol{k}$，但不是把它们做点积，而是把 $\boldsymbol{q}$ 和 $\boldsymbol{k}$ “串”起来“丢”到一个 tanh 函数，再乘上矩阵 W 得到 α。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 α。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。



(a) 点积　　(b) 相加

图 6.18　计算向量关联程度的方法

接下来如何把它套用在自注意力模型里面呢？自注意力模型一般采用查询-键-值（Query-Key-Value，QKV）模式。分别计算 $\boldsymbol{a}^1$ 与 $\boldsymbol{a}^2$、$\boldsymbol{a}^3$、$\boldsymbol{a}^4$ 之间的关联性 α。如图 6.19 所示，把 $\boldsymbol{a}^1$ 乘上 W^q 得到 $\boldsymbol{q}^1$。$\boldsymbol{q}$ 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 $\boldsymbol{a}^2$、$\boldsymbol{a}^3$、$\boldsymbol{a}^4$ 乘上 W^k 得到向量 $\boldsymbol{k}$，向量 $\boldsymbol{k}$ 称为键（key）。把查询 $\boldsymbol{q}^1$ 跟键 $\boldsymbol{k}^2$ 算内积（inner-product）就得到 $\alpha_{1,2}$。$\alpha_{1,2}$ 代表查询是 $\boldsymbol{q}^1$ 提供的，键是 $\boldsymbol{k}^2$ 提供的时候，$\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^2$ 之间的关联性。关联性 α 也被称为注意力的分数。计算 $\boldsymbol{q}^1$ 与 $\boldsymbol{k}^2$ 的内积也就是计算 $\boldsymbol{a}^1$ 与 $\boldsymbol{a}^2$ 的注意力的分数。计算出 $\boldsymbol{a}^1$ 与 $\boldsymbol{a}^2$ 的关联性以后，接下来还需要计算 $\boldsymbol{a}^1$ 与 $\boldsymbol{a}^3$、$\boldsymbol{a}^4$ 的关联性。把 $\boldsymbol{a}^3$ 乘上 W^k 得到键 $\boldsymbol{k}^3$，$\boldsymbol{a}^4$ 乘上 W^k 得到键 $\boldsymbol{k}^4$，再把键 $\boldsymbol{k}^3$ 跟查询 $\boldsymbol{q}^1$ 做内积，得到 $\boldsymbol{a}^1$ 与 $\boldsymbol{a}^3$ 之间的关联性，即 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^3$ 的注意力分数。把 $\boldsymbol{k}^4$ 跟 $\boldsymbol{q}^1$ 做点积，得到 $\alpha_{1,4}$，即 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^4$ 之间的关联性。

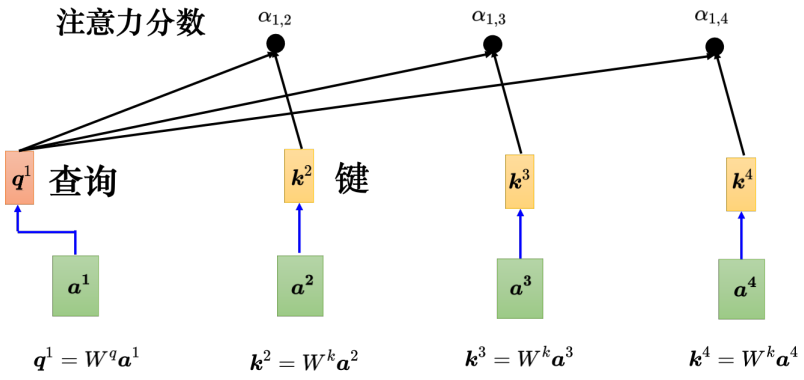


图 6.19　自注意力机制中使用点乘

一般在实践的时候，如图 6.20 所示，$\boldsymbol{a}^1$ 也会跟自己算关联性，把 $\boldsymbol{a}^1$ 乘上 W^k 得到 $\boldsymbol{k}^1$。用 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^1$ 去计算 $\boldsymbol{a}^1$ 与自己的关联性。计算出 $\boldsymbol{a}^1$ 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 (6.3) 所示，把 α 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 α'。这里的 softmax 操作跟分类的 softmax 操作是一模一样的。

$$\alpha'_{1,i} = \exp\left(\alpha_{1,i}\right) / \sum_j \exp\left(\alpha_{1,j}\right) \tag{6.3}$$

所以本来有一组 α，通过 softmax 就得到一组 α'。

> 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 α' 以后，接下来根据 α' 去抽取出序列里面重要的信息。如图 6.21 所示，根据 α 可知哪些向量跟 $\boldsymbol{a}^1$ 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 $\boldsymbol{a}^1$ 到 $\boldsymbol{a}^4$ 乘上 $\boldsymbol{W^v}$ 得到新的向量：$\boldsymbol{v}^1$、$\boldsymbol{v}^2$、$\boldsymbol{v}^3$ 和 $\boldsymbol{v}^4$，接下来把每一个向量都去乘上注意力的分数 α'，再把它们加起来，如式 (6.4) 所示。

$$\boldsymbol{b}^1 = \sum_i \alpha'_{1,i}\boldsymbol{v}^i \tag{6.4}$$

如果 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^2$ 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

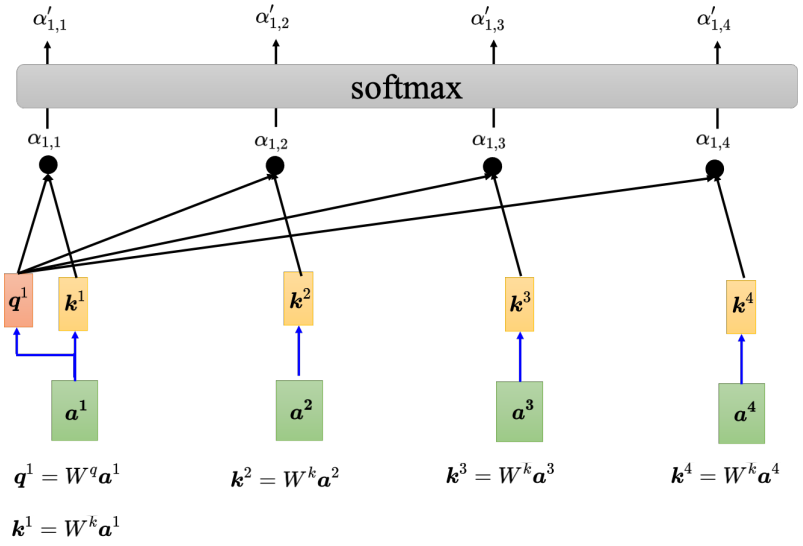


图 6.20　添加 softmax

的 $\boldsymbol{b}^1$ 的值就可能会比较接近 $\boldsymbol{v}^2$，所以谁的注意力的分数最大，谁的 $\boldsymbol{v}$ 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 $\boldsymbol{b}^1$。同理，可以计算出 $\boldsymbol{b}^2$ 到 $\boldsymbol{b}^4$。

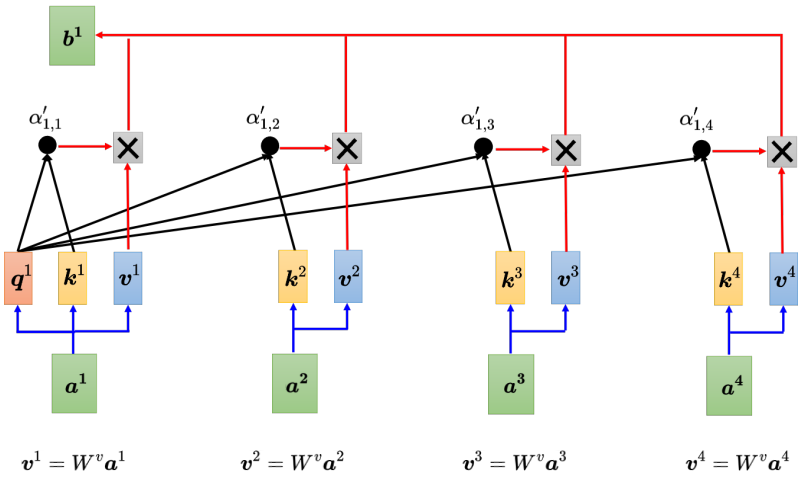


图 6.21　根据 α' 抽取序列中重要的信息

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

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

$\boldsymbol{q}$、$\boldsymbol{k}$ 和 $\boldsymbol{v}$。

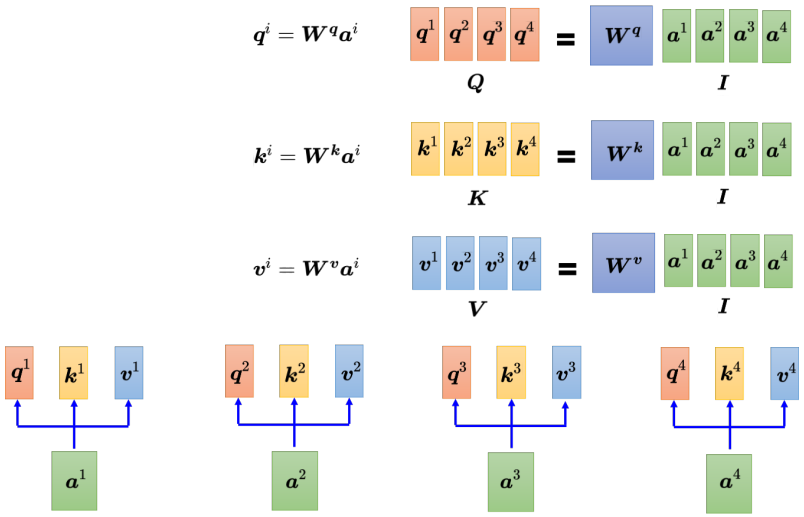


图 6.22　从矩阵乘法的角度理解自注意力的运作过程

如图 6.23 所示，下一步是每一个 $\boldsymbol{q}$ 都会去跟每一个 $\boldsymbol{k}$ 去计算内积，去得到注意力的分数，先计算 $\boldsymbol{q}^1$ 的注意力分数。

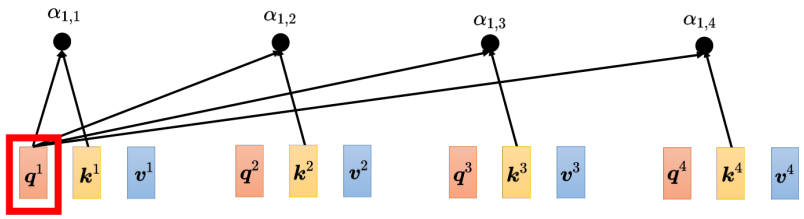


图 6.23　使用 $\boldsymbol{q}^1$ 计算注意力分数

如图 6.24 所示，如果从矩阵操作的角度来看注意力计算这个操作，把 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^1$ 做内积，得到 $\alpha_{1,1}$。$\boldsymbol{q}^1$ 乘上 $(\boldsymbol{k}^1)^{\mathrm{T}}$，也就是 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^1$ 做内积。同理，$\alpha_{1,2}$ 是 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^2$ 做内积，$\alpha_{1,3}$ 是 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^3$ 做内积，$\alpha_{1,4}$ 就是 $\boldsymbol{q}^1$ 跟 $\boldsymbol{k}^4$ 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。$\boldsymbol{q}^1$ 乘 $\boldsymbol{k}^1$，$\boldsymbol{q}^1$ 乘 $\boldsymbol{k}^2$，$\boldsymbol{q}^1$ 乘 $\boldsymbol{k}^3$，$\boldsymbol{q}^1$ 乘 $\boldsymbol{k}^4$ 这四个动作，可以看作是把 $(\boldsymbol{k}^1)^{\mathrm{T}}$ 到 $(\boldsymbol{k}^4)^{\mathrm{T}}$ 拼起来当作是一个矩阵的四行，把这个矩阵乘上 $\boldsymbol{q}^1$ 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 $\alpha_{1,4}$。

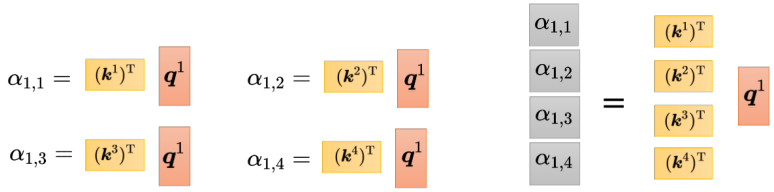


图 6.24　从矩阵操作角度来理解注意力分数计算的过程

如图 6.25 所示，不只是 $\boldsymbol{q}^1$ 要对 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$ 计算注意力，$\boldsymbol{q}^2$ 也要对 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$ 计算注意力。我们把 $\boldsymbol{q}^2$ 也乘上 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$，得到 $\alpha_{2,1}$ 到 $\alpha_{2,4}$。现在的操作是一模一样的，把 $\boldsymbol{q}^3$ 乘 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$，

把 $\boldsymbol{q}^4$ 乘上 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$ 可以得到注意力的分数。

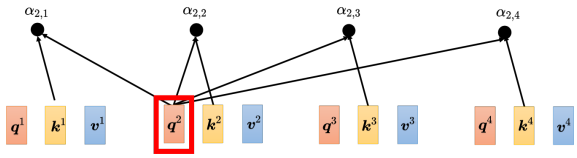


图 6.25　使用 $\boldsymbol{q}^2$ 计算注意力分数

如图 6.26 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 $\boldsymbol{k}$，即 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$。另外一个矩阵的列就是 $\boldsymbol{q}$，即 $\boldsymbol{q}^1$ 到 $\boldsymbol{q}^4$。把 $\boldsymbol{k}$ 所形成的矩阵 $\boldsymbol{K}^{\mathrm{T}}$ 乘上 $\boldsymbol{q}$ 所形成的矩阵 $\boldsymbol{Q}$ 就得到这些注意力的分数。假设 $\boldsymbol{K}$ 的列是 $\boldsymbol{k}^1$ 到 $\boldsymbol{k}^4$，在这边相乘的时候，要对矩阵 $\boldsymbol{K}$ 做一下转置得到 $\boldsymbol{K}^{\mathrm{T}}$，$\boldsymbol{K}^{\mathrm{T}}$ 乘上 $\boldsymbol{Q}$ 就得到矩阵 $\boldsymbol{A}$，$\boldsymbol{A}$ 里面存的就是 $\boldsymbol{Q}$ 跟 $\boldsymbol{K}$ 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 $\boldsymbol{A}$ 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 α 做 softmax 操作以后，它得到的值有异于 α 的原始值，所以用 $\boldsymbol{A}'$ 来表示通过 softmax 以后的结果。

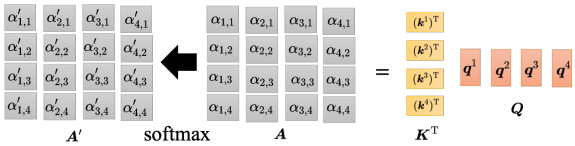


图 6.26　注意力分数的计算过程

如图 6.27 所示，计算出 $\boldsymbol{A}'$ 以后，需要把 $\boldsymbol{v}^1$ 到 $\boldsymbol{v}^4$ 乘上对应的 α 再相加得到 $\boldsymbol{b}$。如果把 $\boldsymbol{v}^1$ 到 $\boldsymbol{v}^4$ 当成是矩阵 $\boldsymbol{V}$ 的 4 个列拼起来，则把 $\boldsymbol{A}'$ 的第一个列乘上 $\boldsymbol{V}$ 就得到 $\boldsymbol{b}^1$，把 $\boldsymbol{A}'$ 的第二个列乘上 $\boldsymbol{V}$ 得到 $\boldsymbol{b}^2$，以此类推。所以等于把矩阵 $\boldsymbol{A}'$ 乘上矩阵 $\boldsymbol{V}$ 得到矩阵 $\boldsymbol{O}$。矩阵 $\boldsymbol{O}$ 里面的每一个列就是自注意力的输出 $\boldsymbol{b}^1$ 到 $\boldsymbol{b}^4$。所以整个自注意力的操作过程可分为以下步骤，先产生了 $\boldsymbol{q}$、$\boldsymbol{k}$ 和 $\boldsymbol{v}$，再根据 $\boldsymbol{q}$ 去找出相关的位置，然后对 $\boldsymbol{v}$ 做加权和。这一串操作就是一连串矩阵的乘法。

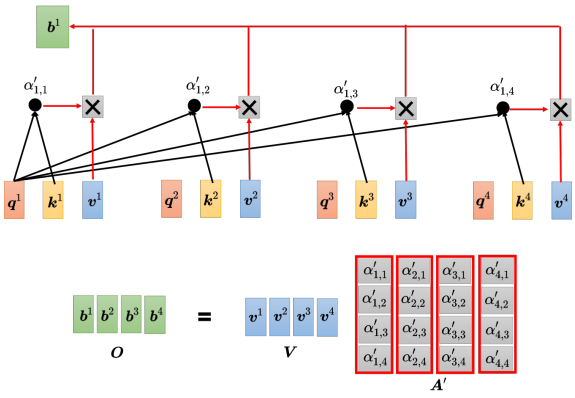


图 6.27　自注意力输出的计算过程

如图 6.28 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 $\boldsymbol{I}$。输入 $\boldsymbol{I}$ 分别乘上三个矩阵：$\boldsymbol{W}^{\boldsymbol{q}}$、$\boldsymbol{W}^{\boldsymbol{k}}$ 跟 $\boldsymbol{W}^{\boldsymbol{v}}$，得到三个矩阵 $\boldsymbol{Q}$、$\boldsymbol{K}$ 和 $\boldsymbol{V}$。接下来 $\boldsymbol{Q}$ 乘上 $\boldsymbol{K}^{\mathrm{T}}$

得到矩阵 $\boldsymbol{A}$。把矩阵 $\boldsymbol{A}$ 做一些处理可得到 $\boldsymbol{A}'$，$\boldsymbol{A}'$ 称为注意力矩阵（attention matrix）。把 $\boldsymbol{A}'$ 再乘上 $\boldsymbol{V}$ 就得到自注意力层的输出 $\boldsymbol{O}$。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 $\boldsymbol{W^q}$、$\boldsymbol{W^k}$ 跟 $\boldsymbol{W^v}$。只有 $\boldsymbol{W^q}$、$\boldsymbol{W^k}$、$\boldsymbol{W^v}$ 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。

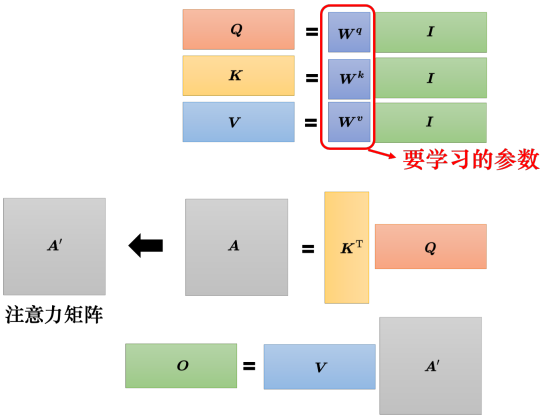


图 6.28　从矩阵乘法的角度来理解注意力

6.3　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，也是需要调的。为什么会需要比较多的头呢？在使用自注意力计算相关性的时候，就是用 $\boldsymbol{q}$ 去找相关的 $\boldsymbol{k}$。但是相关有很多种不同的形式，所以也许可以有多个 $\boldsymbol{q}$，不同的 $\boldsymbol{q}$ 负责不同种类的相关性，这就是多头注意力。如图 6.29 所示，先把 $\boldsymbol{a}$ 乘上一个矩阵得到 $\boldsymbol{q}$，接下来再把 $\boldsymbol{q}$ 乘上另外两个矩阵，分别得到 $\boldsymbol{q}^1$、$\boldsymbol{q}^2$。用两个上标，$\boldsymbol{q}^{i,1}$ 跟 $\boldsymbol{q}^{i,2}$ 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 $\boldsymbol{q}$，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 $\boldsymbol{q}$ 有两个，$\boldsymbol{k}$ 也就要有两个，$\boldsymbol{v}$ 也就要有两个。怎么从 $\boldsymbol{q}$ 得到 $\boldsymbol{q}^1$、$\boldsymbol{q}^2$，怎么从 $\boldsymbol{k}$ 得到 $\boldsymbol{k}^1$、$\boldsymbol{k}^2$，怎么从 $\boldsymbol{v}$ 得到 $\boldsymbol{v}^1$、$\boldsymbol{v}^2$？其实就是把 $\boldsymbol{q}$、$\boldsymbol{k}$、$\boldsymbol{v}$ 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 $\boldsymbol{a}$ 以后，它也会得到两个 $\boldsymbol{q}$、两个 $\boldsymbol{k}$、两个 $\boldsymbol{v}$。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 $\boldsymbol{q}^1$ 在算这个注意力的分数的时候，就不要管 $\boldsymbol{k}^2$ 了，它就只管 $\boldsymbol{k}^1$ 就好。$\boldsymbol{q}^{i,1}$ 分别与 $\boldsymbol{k}^{i,1}$、$\boldsymbol{k}^{j,1}$ 算注意力，在做加权和的时候也不要管 $\boldsymbol{v}^2$ 了，看 $\boldsymbol{v}^{i,1}$ 跟 $\boldsymbol{v}^{j,1}$ 就好，把注意力的分数乘 $\boldsymbol{v}^{i,1}$ 和 $\boldsymbol{v}^{j,1}$，再相加得到 $\boldsymbol{b}^{i,1}$，这边只用了其中一个头。

如图 6.30 所示，我们可以使用另外一个头做相同的事情。$\boldsymbol{q}^2$ 只对 $\boldsymbol{k}^2$ 做注意力，在做加权和的时候，只对 $\boldsymbol{v}^2$ 做加权和得到 $\boldsymbol{b}^{i,2}$。如果有多个头，如 8 个头、16 个头，操作也是一样的。

如图 6.31 所示，得到 $\boldsymbol{b}^{i,1}$ 跟 $\boldsymbol{b}^{i,2}$，可能会把 $\boldsymbol{b}^{i,1}$ 跟 $\boldsymbol{b}^{i,2}$ 接起来，再通过一个变换，即再乘上一个矩阵然后得到 $\boldsymbol{b}^i$，再送到下一层去，这就是自注意力的变形——多头自注意力。

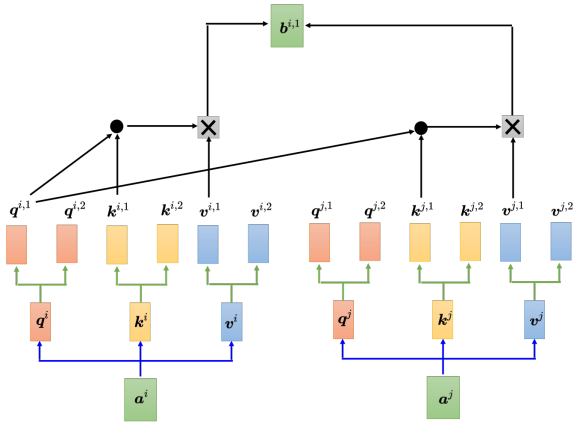


图 6.29　多头自注意力的计算过程

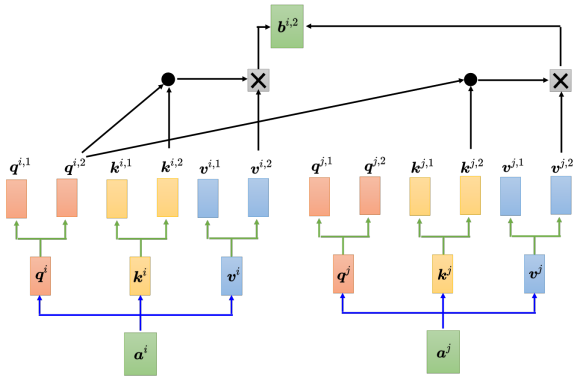


图 6.30　多头自注意力另一个头的计算过程

6.4　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，$\boldsymbol{q}^1$ 跟 $\boldsymbol{q}^4$ 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，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：

图 6.31　从矩阵乘法的角度来理解多头自注意力

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 6.32 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 $\boldsymbol{e}^i$ 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 $\boldsymbol{e}$，把 $\boldsymbol{e}$ 加到 $\boldsymbol{a}^i$ 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 $\boldsymbol{a}^i$ 被加上 $\boldsymbol{e}^i$，它就知道现在出现的位置应该是在 i 这个位置。

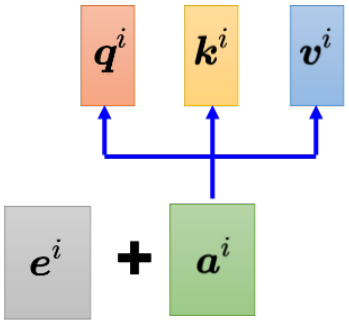


图 6.32　位置编码

最早的 Transformer 论文 “Attention Is All You Need” 用的 $\boldsymbol{e}^i$ 如图 6.33 所示。图上面每一列就代表一个 $\boldsymbol{e}$，第一个位置就是 $\boldsymbol{e}^1$，第二个位置就是 $\boldsymbol{e}^2$，第三个位置就是 $\boldsymbol{e}^3$，以此类推。每一个位置的 $\boldsymbol{a}$ 都有一个专属的 $\boldsymbol{e}$。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。

如图 6.34a 所示，最早的位置编码是用正弦函数所产生的，图 6.34a 中每一行代表一个位置向量。如图 6.34b 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。

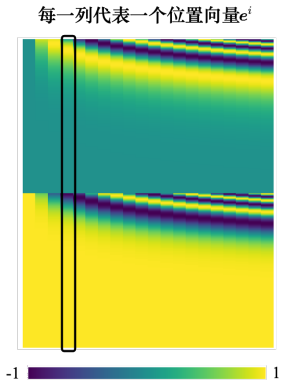
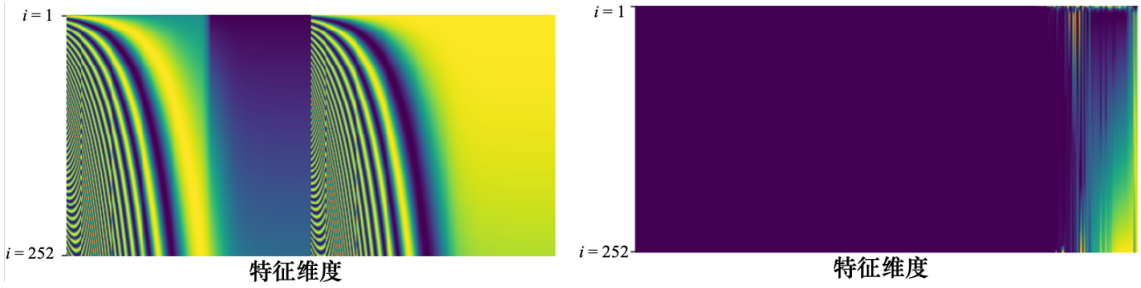


图 6.33　Transformer 中的自注意力



(a) 正弦函数　　(b) 循环神经网络

图 6.34　产生位置编码的各种方法[2]

6.5　截断自注意力

自注意力的应用很广泛，在**自然语言处理（Natural Language Processing，NLP）**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 $\boldsymbol{A}'$ 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，大到不容易处理，不容易训练，

截断自注意力（truncated self-attention）可以处理向量序列长度过大的问题。截断自注意力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。

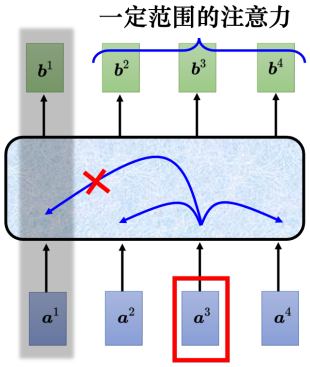
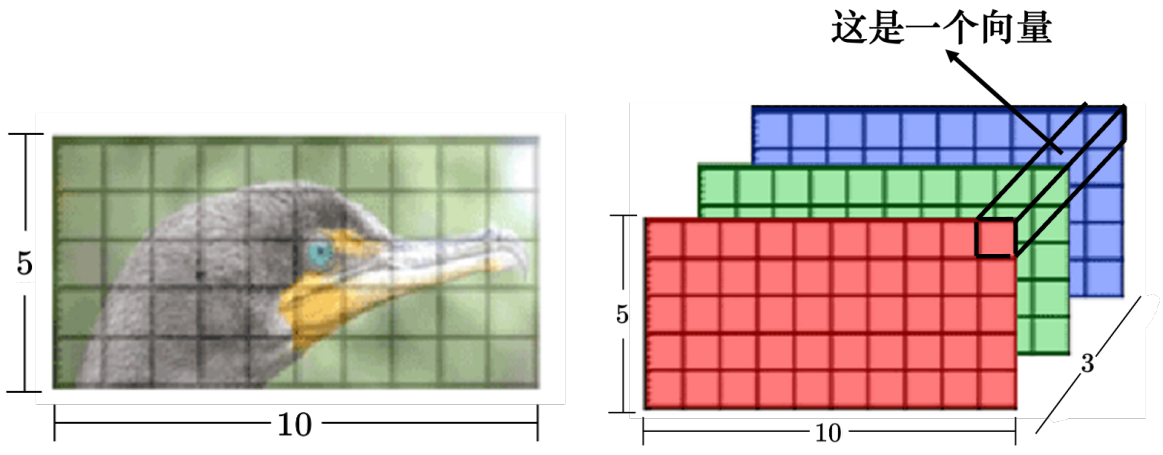


图 6.35　截断自注意力

6.6　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 6.36 所示，一张分辨率为 5×10 的图像（图 6.36a）可以表示为一个大小为 $5 \times 10 \times 3$ 的张量（图 6.36b），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 5×10 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。

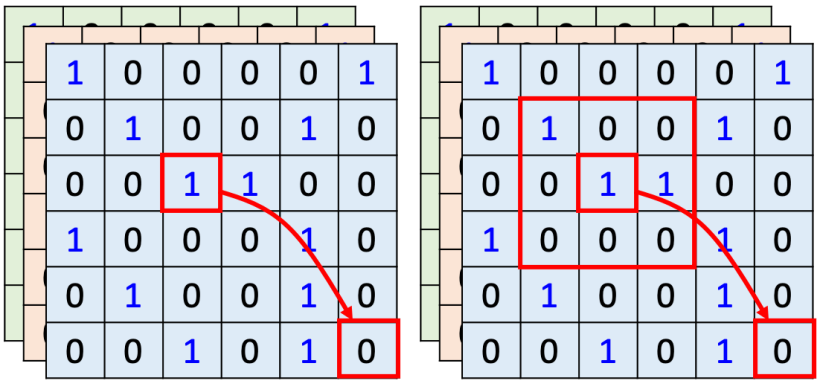


(a) 图像　　(b) 图像的张量表示

图 6.36　使用自注意力处理图像[3]

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 6.37(a) 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 6.37(b) 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，每一个滤波器，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。



(a) 自注意力　　(b) 卷积神经网络

图 6.37　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 6.38 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 6.38　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 16×16 个图像块（patch），它把每一个图像块就想像成是一个字（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 6.39 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。

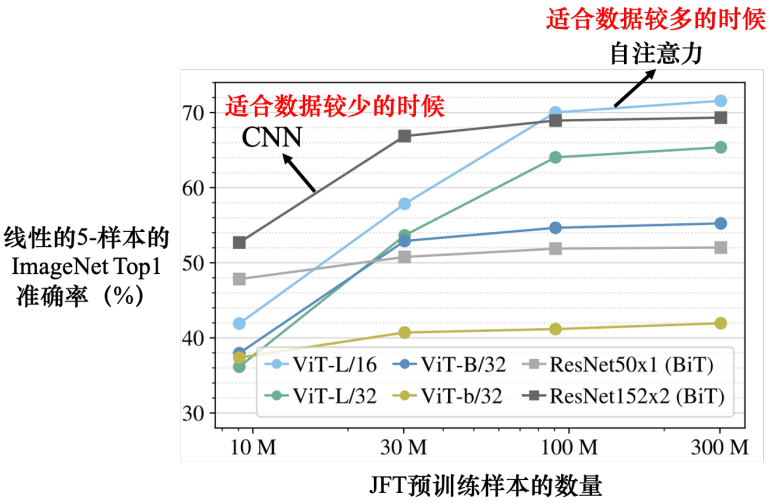


图 6.39　自注意力与卷积神经网络对比[4]

6.7　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。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 6.40b 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。

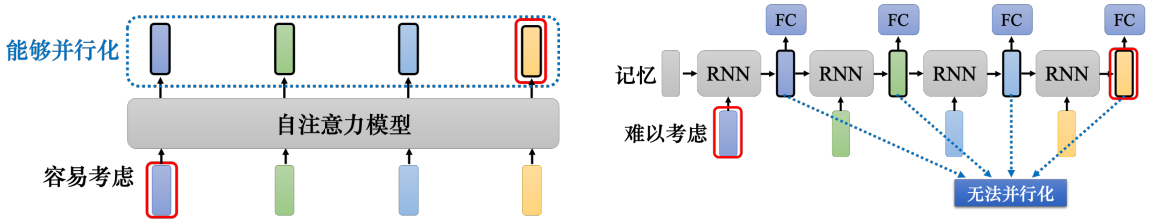
循环神经网络中的隐状态存储了历史信息，可以看作一种记忆（Memory）。

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。输入第三个向量时，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。输入第四个向量输入时，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。如图 6.40(a) 所示，循环神经网络的输入都是一个向量序列。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 6.40(b) 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。但自注意力输出一个查询，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。

自注意力跟循环神经网络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，输出的时候，每一个向量是同时并行产生的，因此在运算速度上，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”。



(a) 使用自注意力处理序列　　(b) 使用循环神经网络处理序列

图 6.40　自注意力与循环神经网络对比

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 6.41 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，这两个节点之间就没有关系。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，所以从领域知识可知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）。

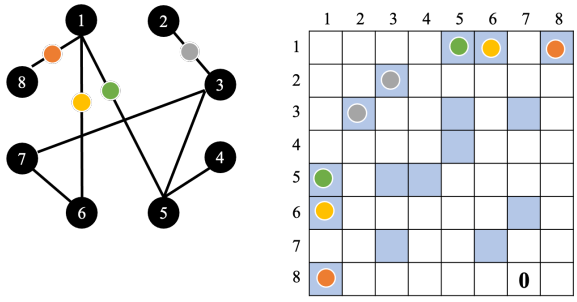


图 6.41　自注意力在图上的应用

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。自注意力最大的问题是其运算量非常大，如何减少自注意力的运算量是未来可研究的重点方向。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，所以后来各种的自注意力的变形都叫做是 xxformer，比如 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。论文 “Efficient Transformers: A Survey” 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 7 章 Transformer

Transformer 是一个基于自注意力的序列到序列模型，与基于循环神经网络的序列到序列模型不同，其可以能够并行计算。本章从两方面介绍了 Transformer，一方面介绍了 Transformer 的结构，即编码器和解码器、编码器-解码器注意力，另一方面介绍了 Transformer 的训练过程以及 序列到序列 模型的训练技巧。

7.1 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。序列到序列模型有广泛的应用，通过这些应用可以更好地了解序列到序列模型。

7.1.1 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 7.1 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: 既然把语音识别系统跟机器翻译系统接起来就能达到语音翻译的效果，那么为什么要做语音翻译?
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白话文的句子，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，输出是另外一种语言的文字，需要学到闽南语的声音信号跟白话文文字的对应关系。YouTube 上面有很多的乡土剧，乡土剧是闽南语语音、白话文字幕，所以只要下载它的闽南语语音和白话文字幕，这样就有闽南语声音信号跟白话文之间的对应关系，就可以训练一个模型来做闽南语的语音识别：输入闽南语，输出白话文。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，输出直接是白话文的文字，这样训练能够做一个闽南语语音识别系统。

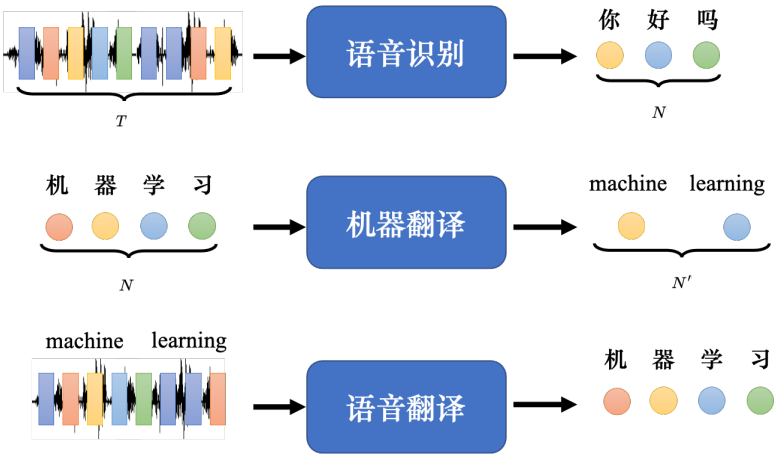


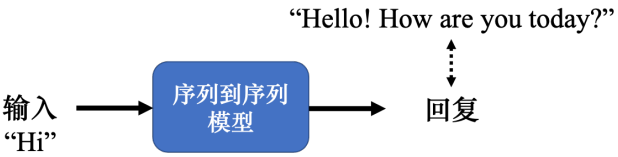
图 7.1 序列到序列的常见应用

7.1.2 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，首先模型会先把白话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。

7.1.3 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 7.2 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。



训练数据:

[PERSON 1:] Hi
[PERSON 2:] Hello ! How are you today ?
[PERSON 1:] I am good thank you , how are you.
[PERSON 2:] Great, thanks ! My children and I were just about to watch Game of Thrones.
[PERSON 1:] Nice ! How old are your children?
[PERSON 2:] I have four that range in age from 10 to 21. You?
[PERSON 1:] I do not have children at the moment.
[PERSON 2:] That just means you get to keep all the popcorn for yourself.
[PERSON 1:] And Cheetos at the moment!
[PERSON 2:] Good choice. Do you watch Game of Thrones?
[PERSON 1:] No, I do not have much time for TV.
[PERSON 2:] I usually spend my time painting: but, I love the show.

图 7.2 聊天机器人的例子

7.1.4 问答任务

序列到序列模型在自然语言处理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：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，砍柴可以用瑞士刀，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。

7.1.5 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 7.3 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。

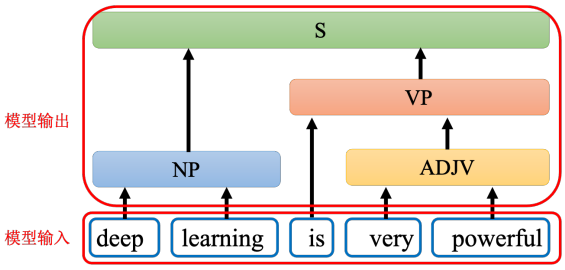


图 7.3 句法分析示例

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 7.4 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”

[1]。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。

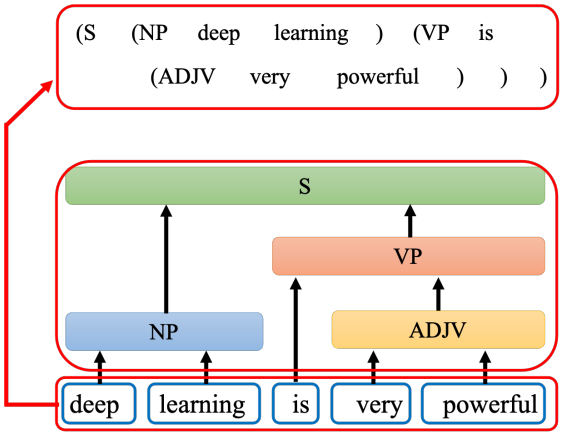


图 7.4 树状结构对应的序列

7.1.6 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 7.5 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。

> 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。

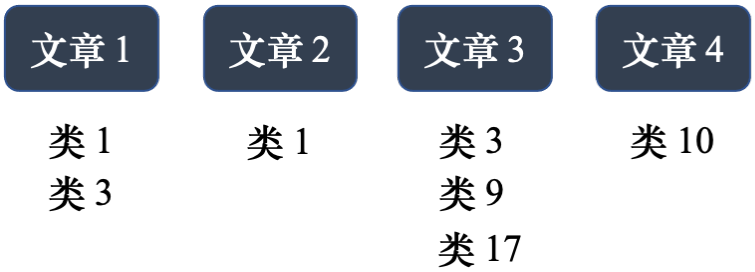


图 7.5 多标签分类示例

多标签分类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 7.6 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]。

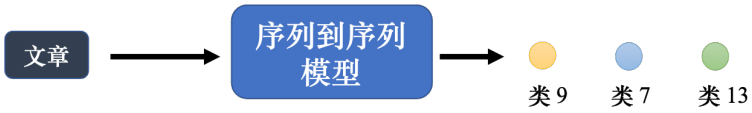


图 7.6　序列到序列模型来解多标签分类问题

7.2　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 7.7 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。

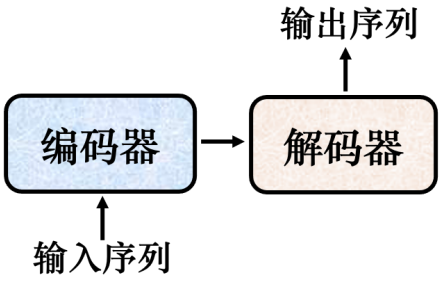


图 7.7　序列到序列模型结构

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”[3]。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 7.8 所示。

7.3　Transformer 编码器

接下来介绍下 Transformer 的编码器。如图 7.9 所示，编码器输入一排向量，输出另外一排向量。自注意力、循环神经网络、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 7.10 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 7.11 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。接下来这排向量会“丢”到全连接网络网络里面，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 7.12 所示，最左边的向量 $\boldsymbol{b}$ 输入到自注意力层后得到向量 $\boldsymbol{a}$，输出向量 $\boldsymbol{a}$ 加上其输入向量 $\boldsymbol{b}$ 得到新的输出。得到残差的结果以后，再做层归一化（layer normalization）。层归一化比信念网络更简单，不需要考虑批量的信息，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平均值和标准差。

批量归一化是对不同样本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 $\boldsymbol{x}$

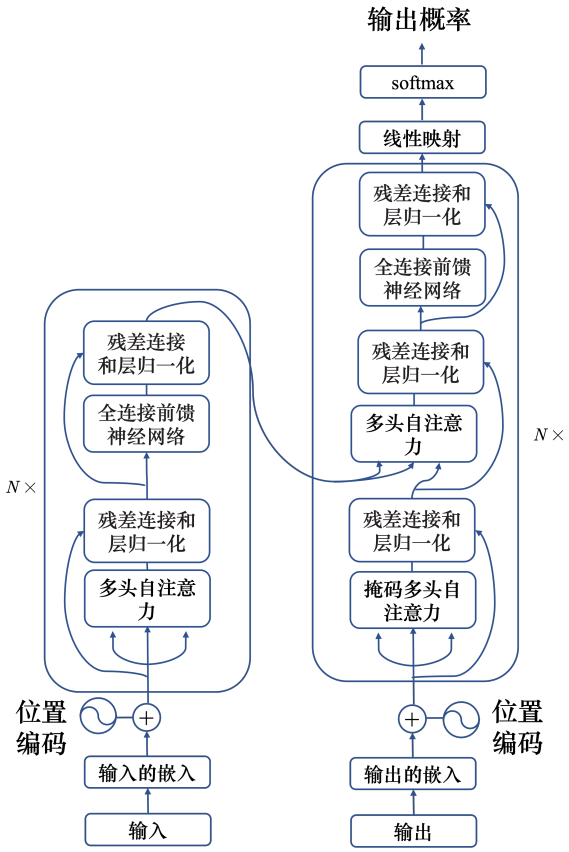


图 7.8 Transformer 结构

里面每一个维度减掉均值 m，再除以标准差 σ 以后得到 $\boldsymbol{x}'$ 就是层归一化的输出，如式 (7.1) 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。

$$x_i' = \frac{x_i - m}{\sigma} \tag{7.1}$$

图 7.13 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。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。多头自注意力就是自注意力的块。经过自注意力后，还要加上残差连接和层归一化。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。

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。

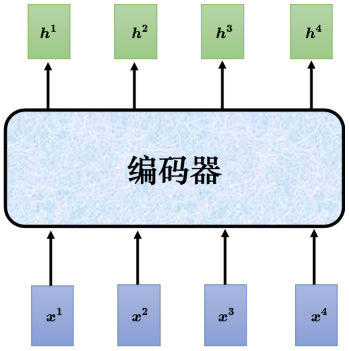


图 7.9 Transformer 编码器的功能

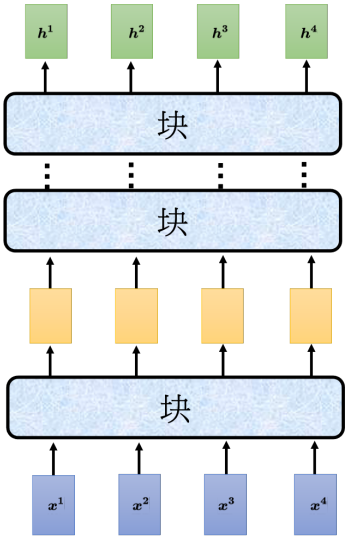


图 7.10 Transformer 编码器结构

7.4 Transformer 解码器

接下来介绍解码器，解码器比较常见的称为自回归的（autoregressive）解码器。

7.4.1 自回归解码器

以语音识别为例，输入一段声音，输出一串文字。如图 7.14 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 <BOS>，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 <BOS>。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 <BOS> 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。

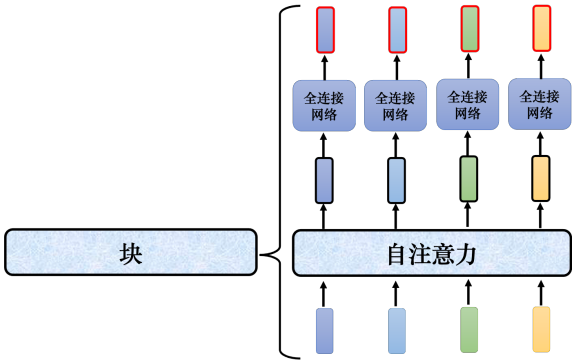


图 7.11　Transformer 编码器中每个块的结构

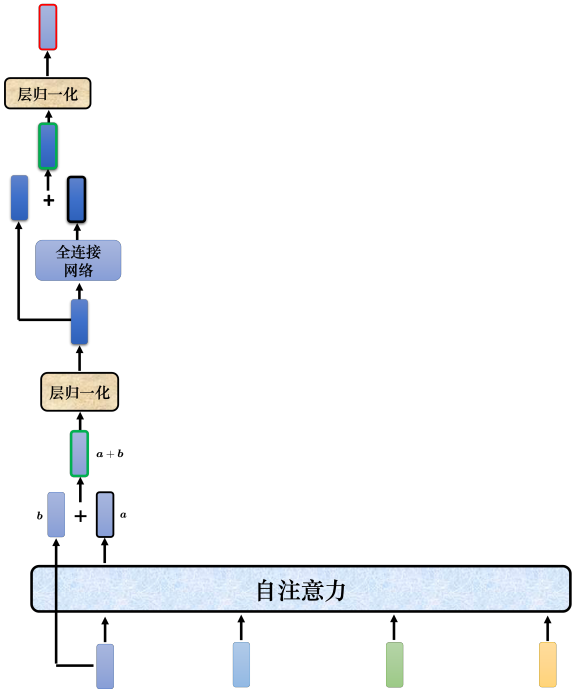


图 7.12　Transformer 中的残差连接

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。比如我们觉得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 7.15 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>

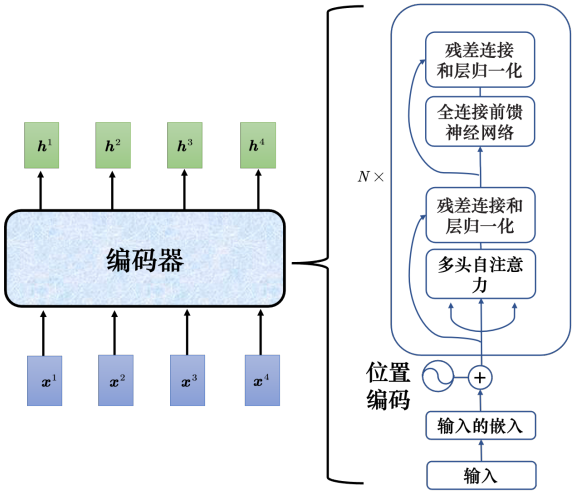


图 7.13　Transformer 编码器结构

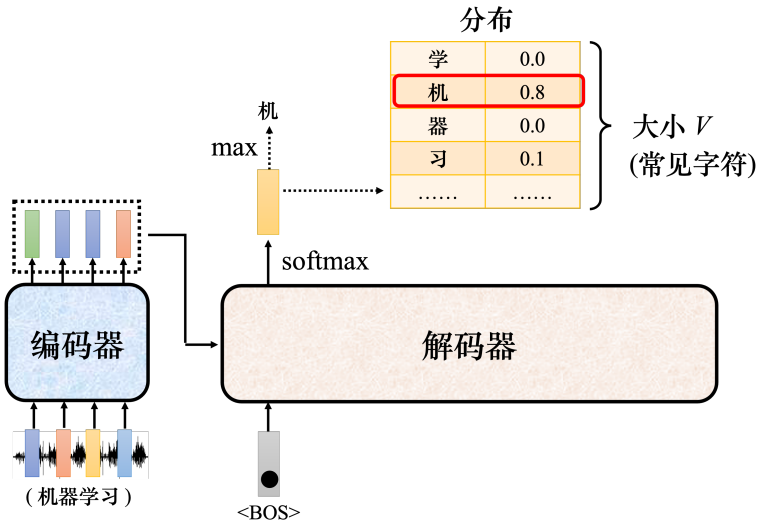


图 7.14　解码器的运作过程

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。

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 7.16 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 7.17 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

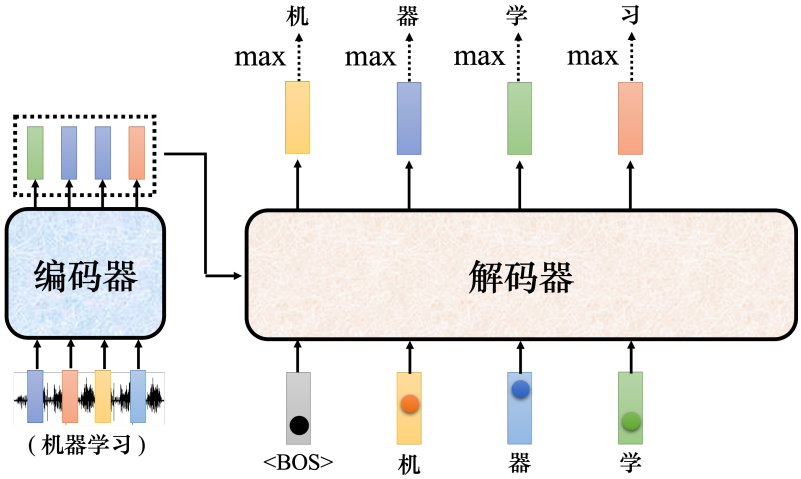


图 7.15　解码器示例

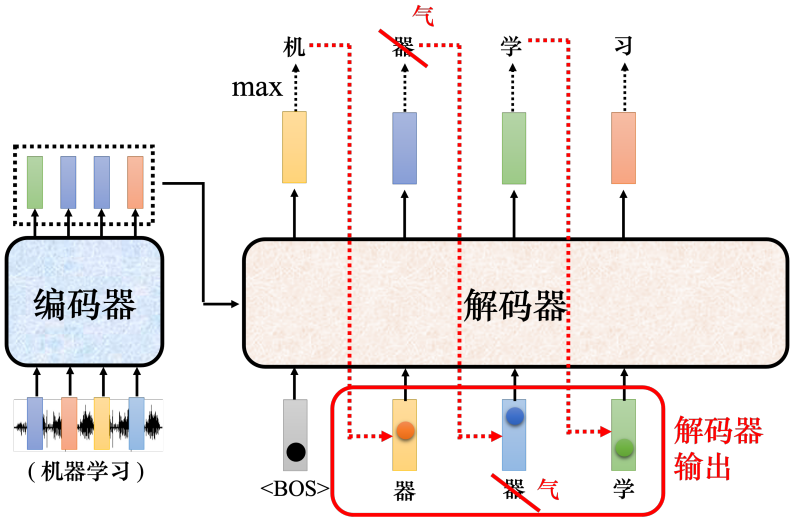


图 7.16　解码器中的误差传播

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 7.18 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 $\boldsymbol{a}^1$ 到 $\boldsymbol{a}^4$ 所有的信息去输出 $\boldsymbol{b}^1$。掩蔽自注意力的不同点是不能再看右边的部分，如图 7.19 所示，产生 $\boldsymbol{b}^1$ 的时候，只能考虑 $\boldsymbol{a}^1$ 的信息，不能再考虑 $\boldsymbol{a}^2$、$\boldsymbol{a}^3$、$\boldsymbol{a}^4$。产生 $\boldsymbol{b}^2$ 的时候，只能考虑 $\boldsymbol{a}^1$、$\boldsymbol{a}^2$ 的信息，不能再考虑 $\boldsymbol{a}^3$、$\boldsymbol{a}^4$ 的信息。产生 $\boldsymbol{b}^3$ 的时候，不能考虑 $\boldsymbol{a}^4$ 的信息。产生 $\boldsymbol{b}^4$ 的时候，可以用整个输入序列的信息。

一般自注意力产生 $\boldsymbol{b}^2$ 的过程如图 7.20 所示。掩蔽自注意力的计算过程如图 7.21 所示，我们只拿 $\boldsymbol{q}^2$ 和 $\boldsymbol{k}^1$、$\boldsymbol{k}^2$ 计算注意力，最后只计算 $\boldsymbol{v}^1$ 跟 $\boldsymbol{v}^2$ 的加权和。不管 $\boldsymbol{a}^2$ 右边的地方，只考虑 $\boldsymbol{a}^1$、$\boldsymbol{a}^2$、$\boldsymbol{q}^1$、$\boldsymbol{q}^2$、$\boldsymbol{k}^1$ 以及 $\boldsymbol{k}^2$。输出 $\boldsymbol{b}^2$ 的时候，只考虑了 $\boldsymbol{a}^1$ 和 $\boldsymbol{a}^2$，没有考虑到 $\boldsymbol{a}^3$ 和 $\boldsymbol{a}^4$。

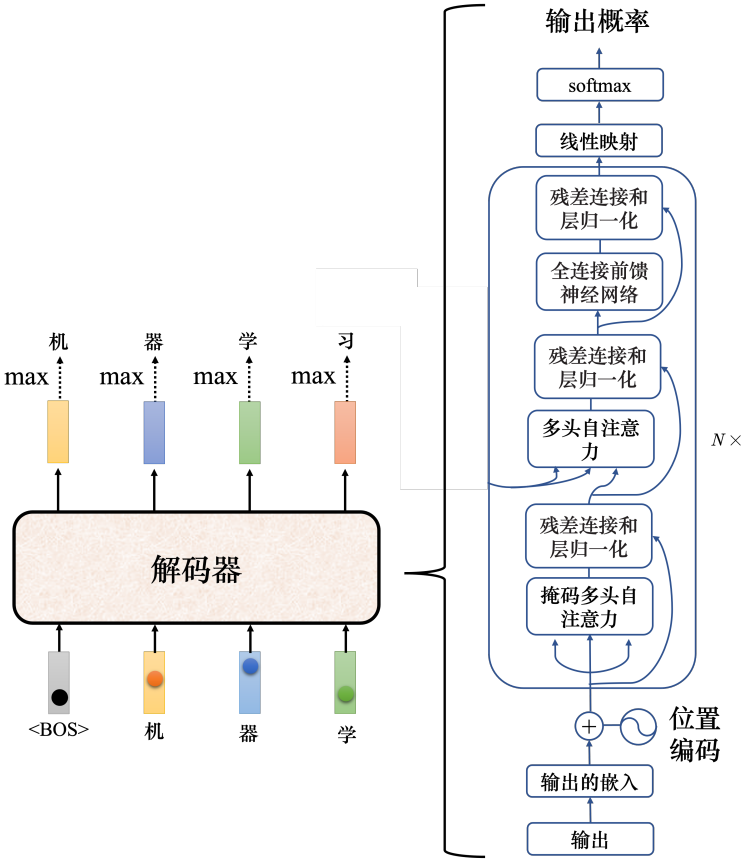


图 7.17 解码器内部结构

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 $\boldsymbol{a}^1$ 再有 $\boldsymbol{a}^2$，再有 $\boldsymbol{a}^3$，再有 $\boldsymbol{a}^4$。这跟原来的自注意力不一样，原来的自注意力 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^4$ 是一次整个输进去模型里面的。编码器是一次把 $\boldsymbol{a}^1$ 跟 $\boldsymbol{a}^4$ 都整个都读进去。但是对解码器而言，先有 $\boldsymbol{a}^1$ 才有 $\boldsymbol{a}^2$，才有 $\boldsymbol{a}^3$ 才有 $\boldsymbol{a}^4$。所以实际上当我们有 $\boldsymbol{a}^2$，要计算 $\boldsymbol{b}^2$ 的时候，没有 $\boldsymbol{a}^3$ 跟 $\boldsymbol{a}^4$ 的，所以无法考虑 $\boldsymbol{a}^3$ $\boldsymbol{a}^4$。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 7.22 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 7.23 所示，要让解码器停止运作，需要特别准备一个特别的符号 <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 <EOS>，解码器看到编码器输出的嵌入、<BOS>、“机”、“器”、“学”、“习”以后，其产生出来的向量里面 <EOS> 的概率必须是最大的，于是输出 <EOS>，整个解码器产生序列的过程就结束了。

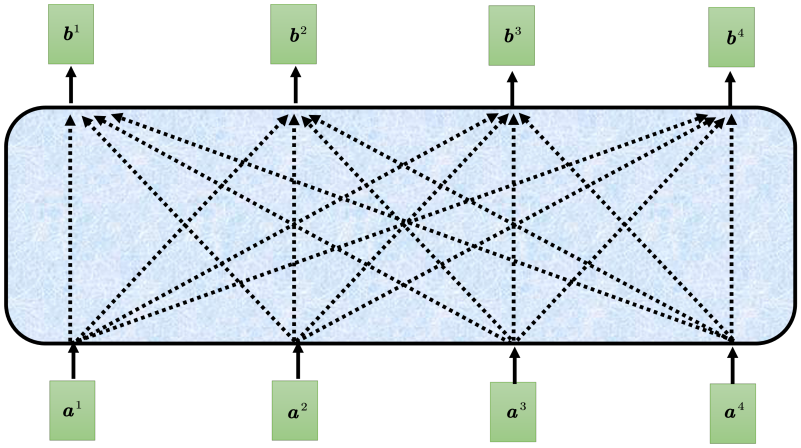


图 7.18　一般的自注意力示例

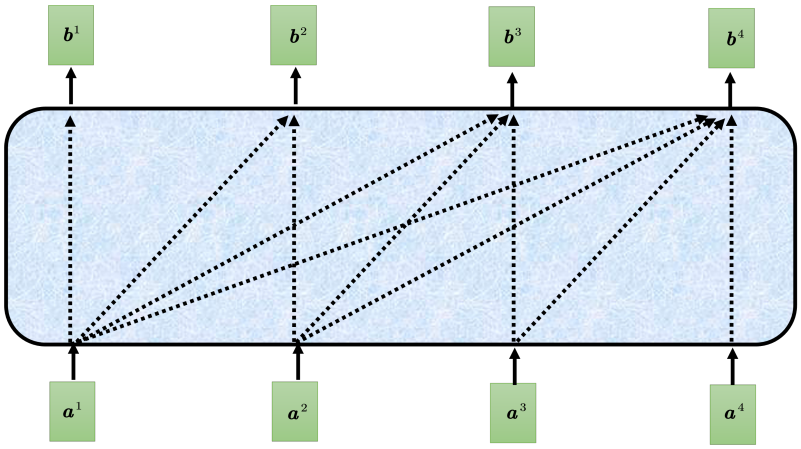


图 7.19　掩蔽自注意力示例

7.4.2　非自回归解码器

接下来讲下非自回归（non-autoregressive）的模型。如图 7.24 所示，自回归的模型是先输入 <BOS>，输出 $\boldsymbol{w}_1$，再把 $\boldsymbol{w}_1$ 当做输入，再输出 $\boldsymbol{w}_2$，直到输出 <EOS> 为止。假设产生是中文的句子，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 <BOS> 词元，一次产生产生一排词元。比如输入 4 个 <BOS> 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 <BOS> 的数量也是未知的，因此有如下两个做法。

- 用分类器来解决这个问题。用分类器“吃”编码器的输入，输出是一个数字，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 <BOS> 的词元，产生 4 个中文的字。
- 给编码器一堆 <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 <BOS>，就会输出 300 个字，输出 <EOS> 右边的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

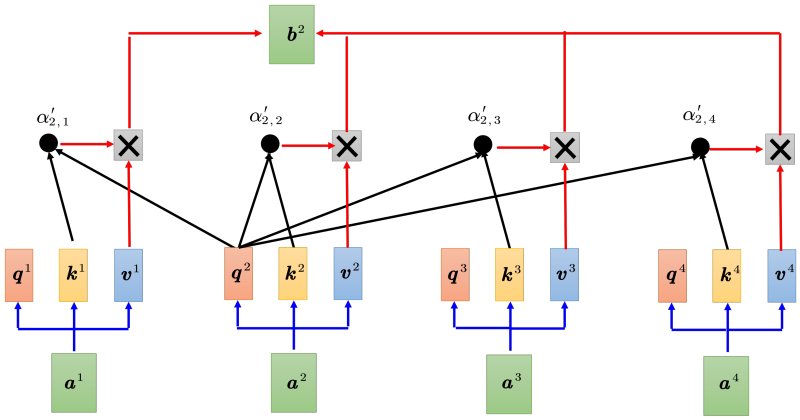


图 7.20　一般自注意力具体计算过程

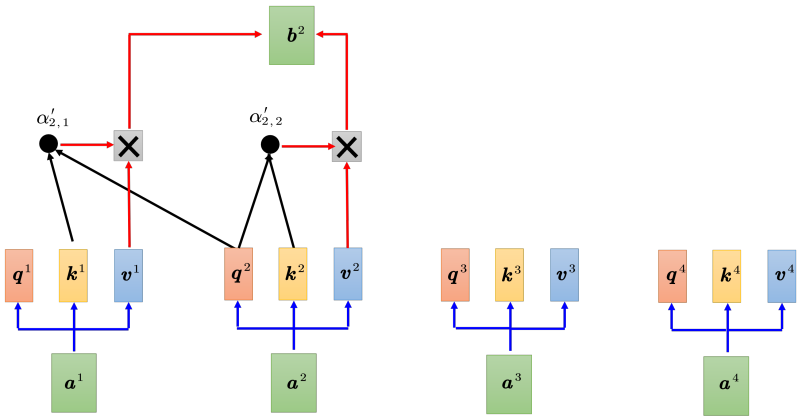


图 7.21　掩蔽自注意力具体计算过程

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 <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。

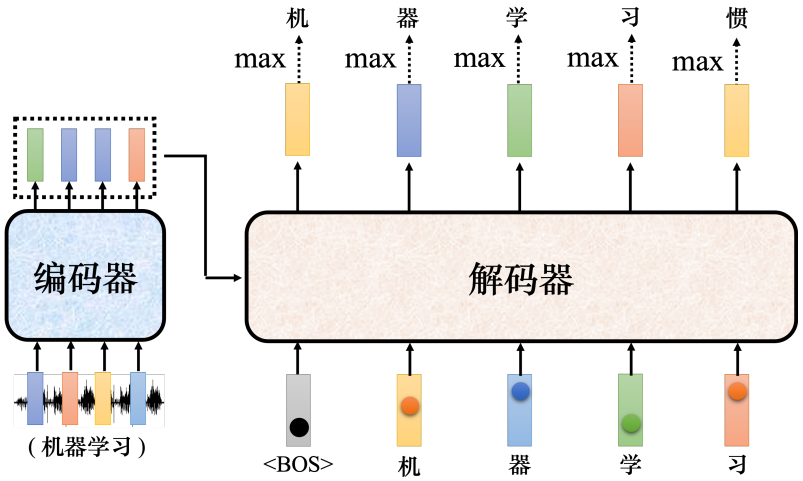


图 7.22　解码器运作的问题

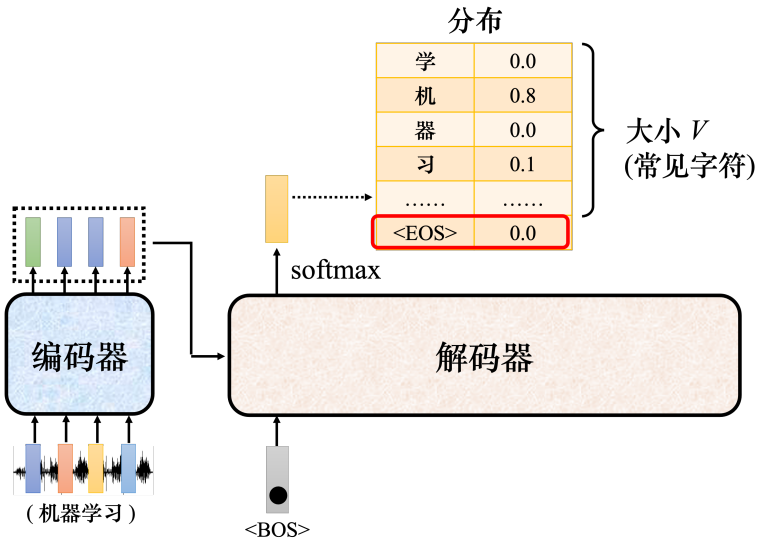


图 7.23　添加 <EOS> 词元

7.5　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 7.25 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 7.26 所示，编码器输入一排向量，输出一排向量 $\boldsymbol{a}^1$、$\boldsymbol{a}^2$、$\boldsymbol{a}^3$。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 $\boldsymbol{q}$，$\boldsymbol{a}^1$、$\boldsymbol{a}^2$、$\boldsymbol{a}^3$ 也都产生键：$\boldsymbol{k}^1$、$\boldsymbol{k}^2$、$\boldsymbol{k}^3$。把 $\boldsymbol{q}$ 跟 $\boldsymbol{k}^1$、$\boldsymbol{k}^2$、$\boldsymbol{k}^3$ 去计算注意力的分数，得到 $\boldsymbol{\alpha}_1$、$\boldsymbol{\alpha}_2$、$\boldsymbol{\alpha}_3$，接下来做 softmax，得到 $\boldsymbol{\alpha}'_1$、$\boldsymbol{\alpha}'_2$、$\boldsymbol{\alpha}'_3$。接下来通过式 (7.2) 可得加权和 $\boldsymbol{v}$。

$$\boldsymbol{v} = \boldsymbol{\alpha}'_1 \times \boldsymbol{v}^1 + \boldsymbol{\alpha}'_2 \times \boldsymbol{v}^2 + \boldsymbol{\alpha}'_3 \times \boldsymbol{v}^3 \tag{7.2}$$

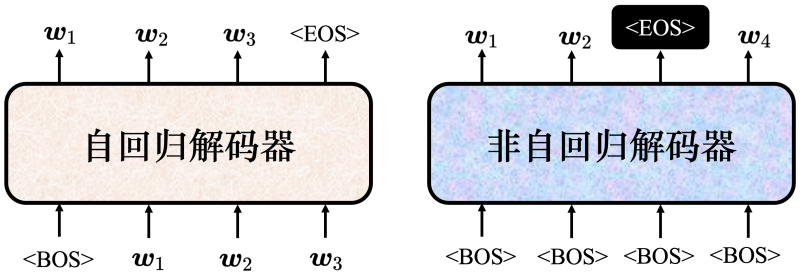


图 7.24　自回归解码器与非自回归解码器对比

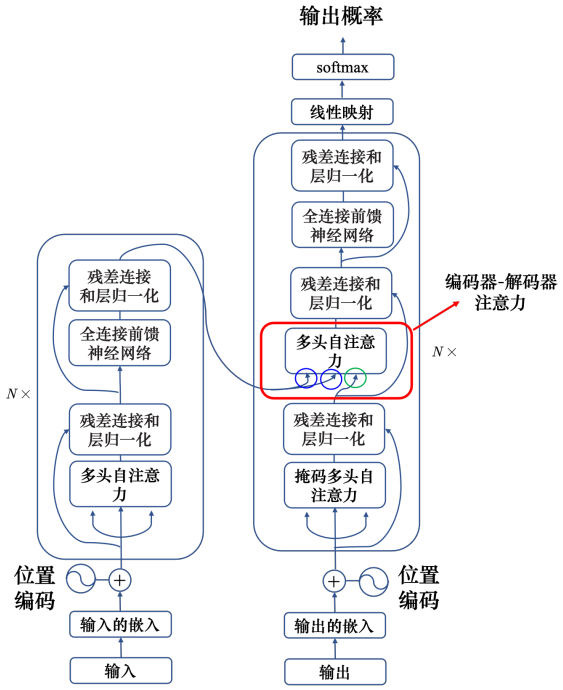


图 7.25　编码器-解码器注意力

$\boldsymbol{v}$ 接下来会“丢”给全连接网络，这个步骤 $\boldsymbol{q}$ 来自于解码器，$\boldsymbol{k}$ 跟 $\boldsymbol{v}$ 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 $\boldsymbol{q}$，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的输入。

如图 7.27 所示，假设产生“机”，输入 <BOS> 、“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 $\boldsymbol{q}'$。$\boldsymbol{q}'$ 会跟 $\boldsymbol{k}^1$、$\boldsymbol{k}^2$、$\boldsymbol{k}^3$ 计算注意力的分数。接着用注意力分数跟 $\boldsymbol{v}^1$、$\boldsymbol{v}^2$、$\boldsymbol{v}^3$ 做加权和，加起来得到 $\boldsymbol{v}'$，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”[4]。

7.6　Transformer 的训练过程

如图 7.28 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 <BOS> 丢给编码器的时候，其第一个输出应该要跟“机”越接近越

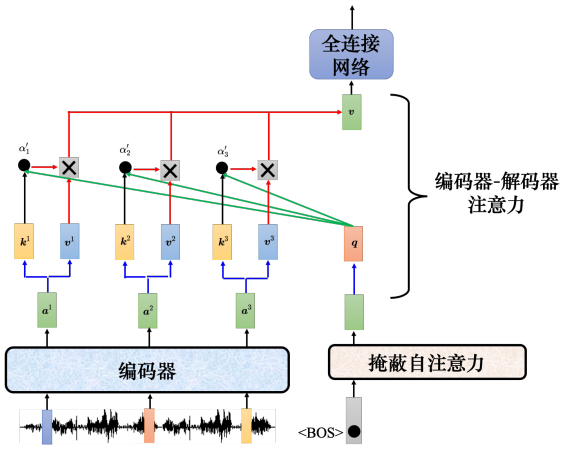


图 7.26　编码器-解码器注意力运作过程

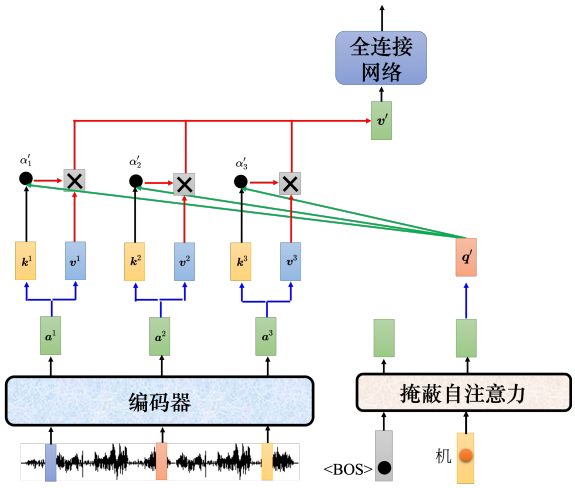


图 7.27　编码器-解码器注意力运作过程示例

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 7.29 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 7.29 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）。

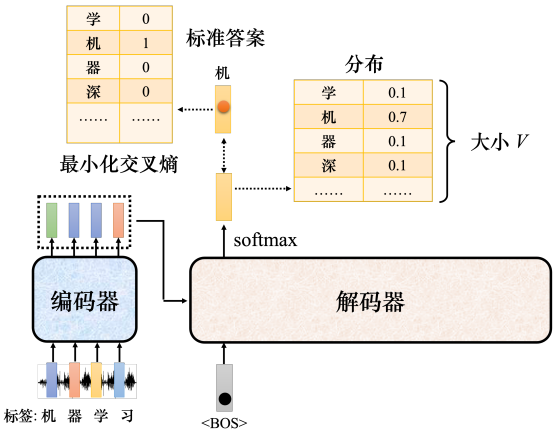


图 7.28 Transformer 的训练过程

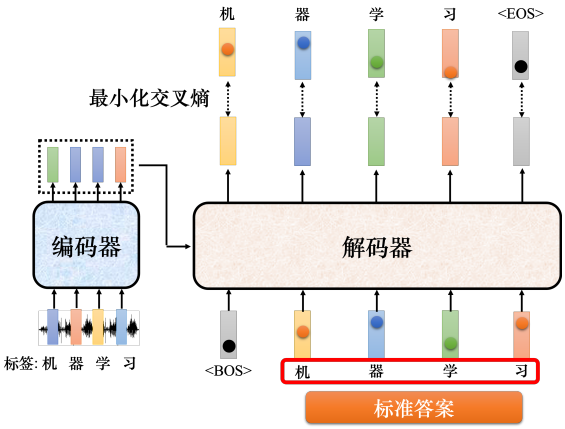


图 7.29 教师强制

7.7 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。

7.7.1 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

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）。

7.7.2 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 7.30 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。

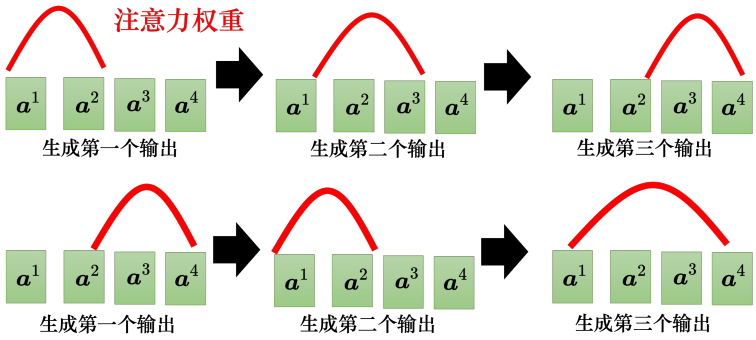


图 7.30 引导注意力

7.7.3 束搜索

如图 7.31 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。对中文而言，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。

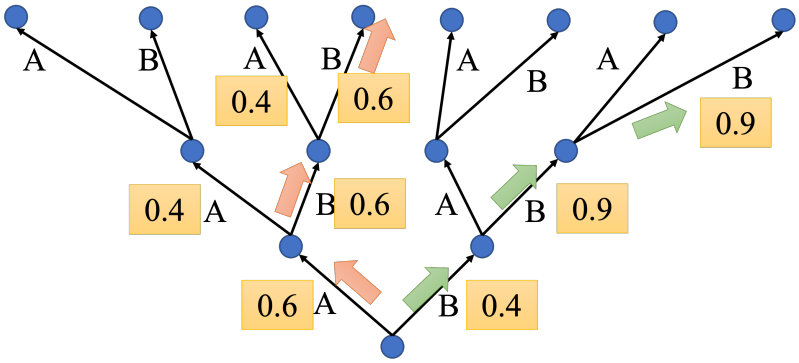


图 7.31　解码器搜索示例

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。

7.7.4　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。

7.7.5 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 Understudy）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”。

7.7.6 计划采样

如图 7.32 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）。

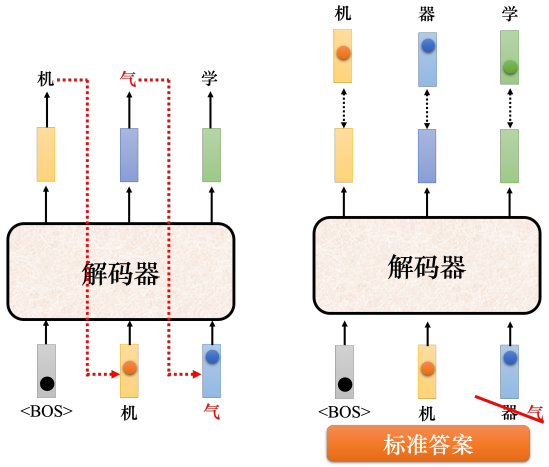


图 7.32 曝光偏差

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，这一技巧称为**计划采样（scheduled sampling）**[6]，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”[7]、“Parallel Scheduled Sampling”[8]。

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, JAITLY 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 8 章 生成模型

目前越来越多的基于生成式的软件出现，极大程度上改变了我们的生活。例如，我们可以通过一张照片，让软件自动生成一段音乐，或者是让软件自动生成一段视频。本章具体介绍它们背后的基础模型——生成模型。

8.1 生成对抗网络

本章介绍**生成模型（generative model）**。到目前为止，我们学习到的网络本质上都是一个函数，即提供一个输入 x，网络就可以输出一个结果 y。并且我们在前几章已经了解各种各样的网络，它们可以应对不同类型的输入 x 和输出 y。例如，当输入 x 是一张图片时，可以使用例如卷积神经网络等模型进行处理，当输入 x 是序列数据时，可以使用基于循环神经网络架构的模型进行处理，其中输出 y 既可以是数值、类别，也可以是一个序列。目前，这些网络已经可以涵盖多数我们日常会遇到的问题了。

8.1.1 生成器

接下来，我们将介绍另一种架构————生成模型。与先前介绍的模型不同的是，生成模型中网络会被作为一个**生成器（generator）**来使用。具体来说，在模型输入时会将一个随机变量 z 与原始输入 x 一并输入到模型中，这个变量是从随机分布中采样得到。输入时可以采用向量拼接的方式将 x 和 z 一并输入，或在 x、z 长度一样时，将二者的加和作为输入。这个变量 z 特别之处在于其非固定性，即每一次我们使用网络时都会从一个随机分布中采样得到一个新的 z。通常，我们对于该随机分布的要求是其足够简单，可以较为容易地进行采样，或者可以直接写出该随机分布的函数，例如**高斯分布（Gaussian distribution）**、**均匀分布（uniform distribution）**等等。所以每次有一个输入 x 的同时，我们都从随机分布中采样得到 z，来得到最终的输出 y。随着采样得到的 z 的不同，我们得到的输出 y 也会不一样。同理，对于网络来说，其输出也不再固定，而变成了一个复杂的分布，我们也将这种可以输出一个复杂分布的网络称为生成器，如图 8.1 所示。

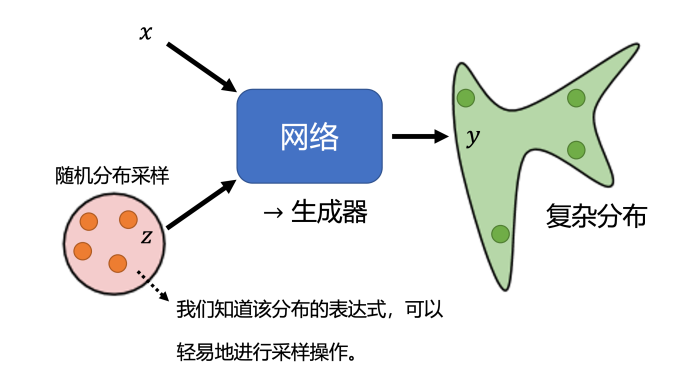


图 8.1 生成器示意图

下面我们介绍如何训练这个生成器。首先，我们为什么要需要训练生成器，为什么需要输出一个分布呢？下面介绍一个视频预测的例子，即给模型一段的视频短片，然后让它预测接下来发生的事情。视频环境是小精灵游戏，预测下一帧的游戏画面，如图 8.2 所示。

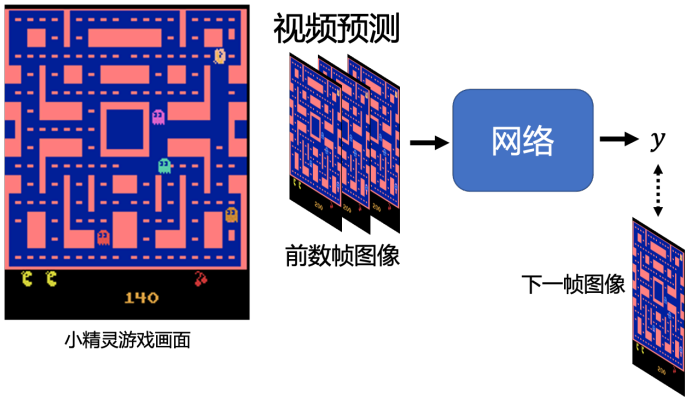


图 8.2　视频预测例子——以小精灵游戏为例

要预测下一帧的游戏画面，我们只需要输入给网络过去几帧游戏画面。要得到这样的训练数据很简单，只需要在玩小精灵的同时进行录制，就可以训练我们的网络，只要让网络的输出 y，与我们的真实图像越接近越好。当然在实践中，我们为了保证高效训练，我们会将每一帧画面分割为很多块作为输入，并行分别进行预测。我们接下来为了简化，假设网络是一次性输入的整个画面。如果我们使用前几章介绍的基于监督学习的训练方法，我们得到的结果可能会是的十分模糊的甚至游戏中的角色消失、出现残影的，如图 8.3 所示。

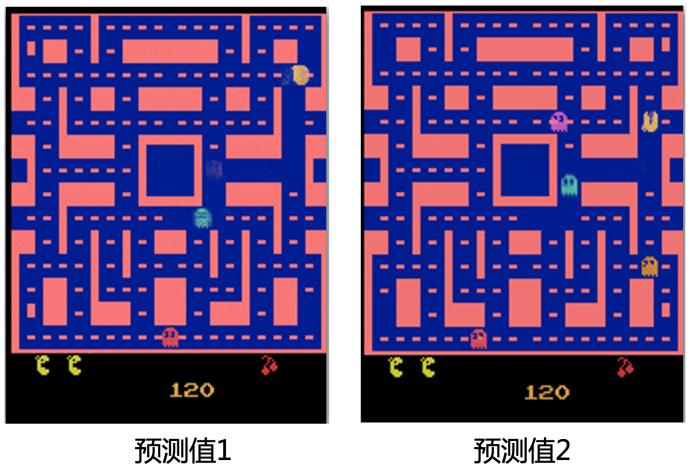


图 8.3　基于监督学习的小精灵游戏的预测值

造成该问题的原因是，我们监督学习中的训练数据对于同样的转角同时存储有角色向左转和向右转两种输出。当我们在训练的时候，对于一条向左转的训练数据，网络得到的指示就是要学会游戏角色向左转的输出。同理，对于一条向右转的训练数据，网络得到的指示就是学会角色向右转的输出。但是实际上这两种数据可能会被同时训练，所以网络就会学到的是“两面讨好”。当这个输出同时距离向左转和向右转最近，网络就会得到一个错误的结果————向左转是对的，向右转也是对的。

所以我们应该如何解决这个问题呢？答案是让网络有概率的输出一切可能的结果，或者说输出一个概率的分布，而不是原来的单一的输出，如图 8.4 所示。当我们给网络一个随机分布时，网络的输入会加上是一个 z，这时输出就变成了一个非固定的分布，其包含了向左转

和向右转的可能。举例来说，假设我们选择的 z 服从一个二项分布，即就只有 0 和 1 并且各占 50%。那么我们的网络就可以学到 z 采样到 1 的时候就向左转，采样到 0 的时候就向右转，这样就可以解决了。

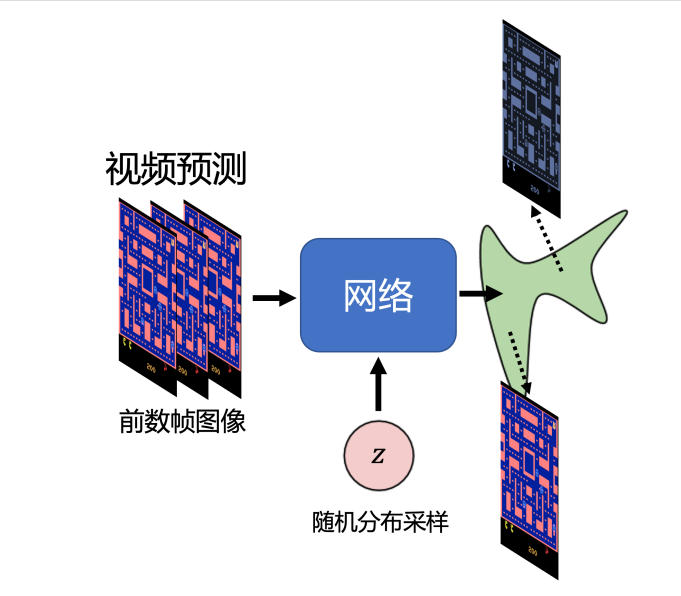


图 8.4　基于生成模型的小精灵游戏的预测结果

回到生成器的讨论中，我们什么需要这类的生成模型呢？答案是当我们的任务需要“创造性”的输出，或者我们想知道一个可以输出多种可能的模型，且这些输出都是对的模型的时候。这可以类比于，让很多人一起处理一个开放式的问题，或者是头脑风暴，大家的回答五花八门可以各自发挥，但是回答都是正确的。所以生成模型也可以被理解为让模型自己拥有了创造的能力。再举两个更具体的例子，对于画图，假设画一个红眼睛的角色，那每个人可能画出来或者心中想的动画人物都不一样。对于聊天机器人，它也需要有创造力。比如我们对机器人说，你知道有哪些童话故事吗？聊天机器人会回答安徒生童话、格林童话甚至其他的，没有一个标准的答案。所以对于我们的生成模型来说，其需要能够输出一个分布，或者说多个答案。当然在生成模型中，非常知名的就是**生成式对抗网络（generative adversarial network）**，我们通常缩写为 GAN。这一节我们就讲介绍这个生成对抗网络。

我们通过让机器生成动画人物的面部来形象地介绍 GAN，首先介绍的是**无限制生成（unconditional generation）**，也就是我们不需要原始输入 x。其对应的就是需要原始输入 x 的**条件型生成（conditional generation）**。如图 8.5 所示，对于无限制的 GAN，它的唯一输出就是 z，这里假设为正态分布采样出的向量。其通常是一个低维的向量，例如 50、100 的维度。

我们首先从正态分布中采样得到一个向量 z，并输入到生成器中，生成器会给我们一个对应的输出——一个动漫人物的脸。我们聚焦一下生成器输出一个动漫人物面部的过程。其实很简单，一张图片就是一个高维的向量，所以生成器实际上做的事情就是输出一个高维的向量，比如是一个 64×64 的图片（如果是彩色图片那么输出就是 64×64×3）。当输入的向量 z 不同的时候，生成器的输出就会跟着改变，所以我们从正态分布中采样出不同的 z，得到的输出 y 也就会不同，动漫人脸照片也不同。当然，我们也可以选择其他的分布，但是根据经验，

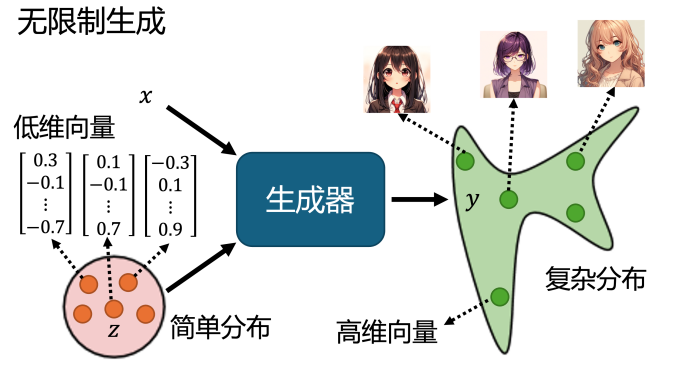


图 8.5　基于无限制生成的 GAN

分布之间的差异可能并没有非常大。大家可以找到一些文献，并且尝试去探讨不同的分布之间的差异。我们这里选择正态分布是因为其简单且常见，而且生成器自己会想方设法把这个简单的分布对应到一个更复杂的分布。所以我们后续的讨论都以正态分布为前提。

8.1.2　辨别器

在 GAN 中，除了生成器以外，我们要多训练一个**判别器（discriminator）**，其通常是一个神经网络。判别器会输入一张图片，输出一个标量，其数值越大就代表现在输入的图片越像是真实的动漫人物的图像，如图 8.6 所示。举例来说，对于图 8.6中的动漫人物头像，那输出就是 1。这里假设 1 是最大的值，画得很好的动漫图像输出就是 1，不知道在画什么就输出 0.5，再差一些就输出 0.1 等等。判别器从本质来说与生成器一样也是神经网络，是由我们自己设计的，可以用卷积神经网络，也可以用 Transformer，只要能够产生出我们要的输入输出即可。当然对于这个例子，因为输入是一张图片，所以选择卷积神经网络，因为其在处理图像上有非常大的优势。

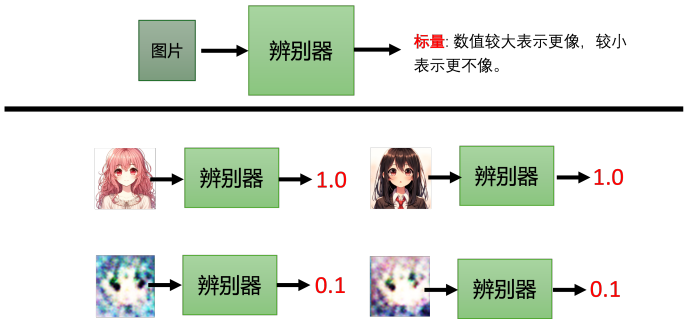


图 8.6　GAN 中的辨别器

我们回到动漫人物图片的例子，生成器学习画出动漫的人物的过程如图 8.7所示。首先，第一代生成器的参数几乎是完全随机的，所以它根本就不知道要怎么画动漫人物，所以其画出来的东西就是一些莫名其妙的噪音。那判别器学习的目标是成功分辨生成器输出的动漫图片。当然在图 8.7的例子里面可能非常容易，对判别器来说它只要看图片中是否有两个黑黑的眼睛即可。接下来生成器就要通过训练调整里面的参数来骗过判别器。假设判别器判断一张图片是不是真实图片的依据是看图片有没有眼睛，那新的生成器就需要输出有眼睛的图片。所以

生成器产生眼睛出来，它是可以骗过第一代的判别器的。同时判别器也是会进化的，其会试图分辨新的生成器与真实图片之间的差异。例如，通过有没有嘴巴来识别真假。所以第三代的生成器就会想办法去骗过第二代的判别器，比如把嘴巴加上去。当然同时判别器也会逐渐的进步，会越来越严苛，来“逼迫”生成器产生出来的图片越来越像动漫的人物。所以生成器和判别器彼此之间是一直的互动、促进关系，和我们所说的“内卷”一样。最终，生成器会学会画出动漫人物的脸，而判别器也会学会分辨真假图片，这就是 GAN 的训练过程。

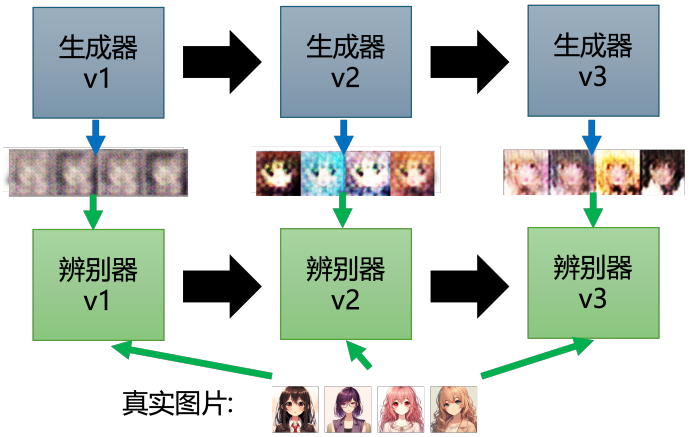


图 8.7　GAN 训练的过程

GAN 最早出现在 14 年的一篇文章中，其作者把生成器和判别器当作是敌人，并且生成器和判别器中间有一个对抗的关系，所以就用了一个“对抗”这个单词，当然这只是一个拟人化的说法而已。但是其实我们也可以把他们想为亦敌亦友的关系，毕竟它们一直在比这更新，比着提升自己，超越对方。

8.2　生成器与辨别器的训练过程

下面，我们从算法角度来解释生成器和判别器是如何运作的，如图 8.8所示。生成器和判别器是两个网络，在训练前我们要先分别进行参数初始化。训练的第一步是固定生成器，只训练判别器。因为生成器的初始参数是随机初始化的，所以它什么都没有学习到，输入一系列采样得到的向量给它，它的输出肯定都是些随机、混乱的图片，就像是坏掉的老式电视收不到信号时的花屏一样，与真实的动漫头像完全不同。同时，我们会有一个很多动漫人物头像的图像数据库，可以通过爬虫等方法得到。我们会从这个图库中采样一些动漫人物头像图片出来，来与生成器产生出来的结果对比从而训练判别器。判别器的训练目标是要分辨真正的动漫人物与生成器产生出来的动漫人物间的差异。具体来说，我们把真正的图片都标 1，生成器产生出来的图片都标 0。接下来对于判别器来说，这就是一个分类或回归的问题。如果是分类的问题，我们就把真正的人脸当作类别 1，生成器产生出来的图片当作类别 2，然后训练一个分类器。如果当作回归的问题，判别器看到真实图片就要输出 1，生成器的图片就要输出 0，并且进行 0-1 之间的打分。总之，判别器就学着去分辨这个真实图像和产生出来的图像间的差异。

我们训练完判别器以后，接下来第二步，固定判别器，训练生成器，如图 8.9所示。训练生成器的目的就是让生成器想办法去骗过判别器，因为在第一步中判别器已经学会分辨真图和假图间的差异。生成器如果可以骗过判别器，那生成器产生出来的图片可能就可以以假乱真。具体的操作如下：首先生成器输入一个向量，其可以来源于我们之前介绍的高斯分布中采

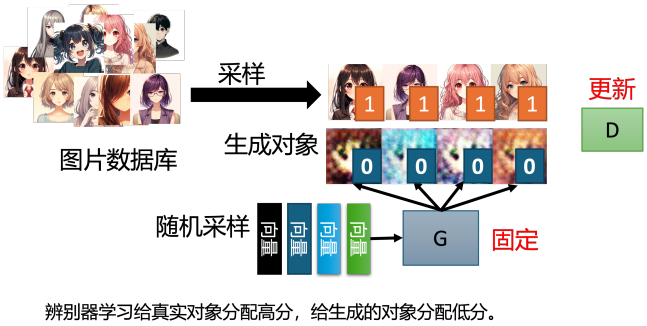


图 8.8　GAN 算法的第一步

样数据，并产生一个图片。接着我们将这个图片输入到判别器中，判别器会给这个图片一个打分。这里判别器是固定的，它只需要给更“真”的图片更高的分数即可，生成器训练的目标就是让图片更加真实，也就是提高分数。

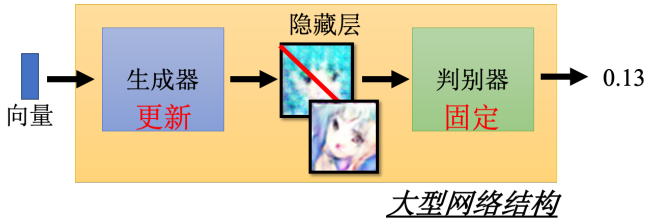


图 8.9　GAN 算法的第二步

对于真实场景中生成器和判别器都是有很多层的神经网络，我们通常将两者一起当作一个比较大的网络来看待，但是不会调整判别器部分的模型参数。因为假设要输出的分数越大越好，那我们完全可以直接调整最后的输出层，改变一下偏差值设为很大的值，那输出的得分就会很高，但是完全达不到我们想要的效果。所以我们只能训练生成的部分，训练方法与前几章介绍的网络训练方法基本一致，只是我们希望优化目标越大越好，这个与之前我们希望损失越小越好不同。当然我们也可以直接在优化目标前加“负号”，就当作损失看待也可以，这样就变为了让损失变小。另一种方法，我们可以使用梯度上升进行优化，而取代之前的梯度下降优化算法。

总结一下，GAN 算法的两个步骤。步骤一，固定生成器训练判别器；步骤二，固定判别器训练生成器。接下来就是重复以上的训练，训练完判别器固定判别器训练生成器。训练完生成器以后再用生成器去产生更多的新图片再给判别器做训练。训练完判别器后再训练生成器，如此反覆地去执行。当其中一个进行训练的时候，另外一个就固定住，期待它们都可以在自己的目标处达到最优，如图 8.10所示。

8.3　GAN 的应用案例

下面介绍一些 GAN 算法的应用。首先介绍 GAN 生成动画人物人脸的例子，如图 8.11所示。这些分别是训练 100 轮、1000 轮、2000 轮、5000 轮、10000 轮、20000 轮和 50000 轮的结果。我们可以看到训练 100 轮时，生成的图片还比较模糊；训练到 1000 轮的时候，机器就产生了眼睛；训练到 2000 轮的时候，嘴巴就生成出来了；训练到 5000 轮的时候，已经开始

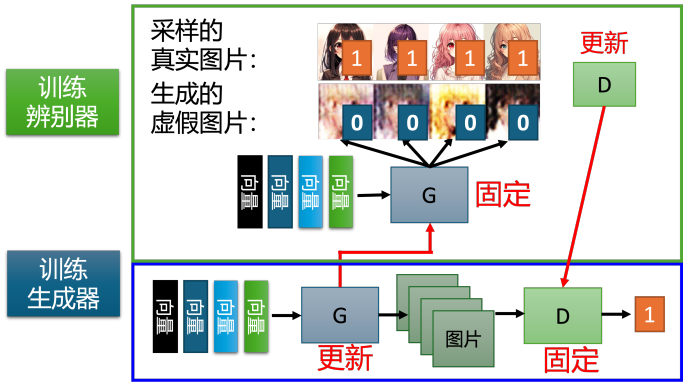


图 8.10　GAN 算法的完整训练过程

有一点人脸的轮廓了，并且机器学到了动画人物水汪汪大眼睛的特征；训练到 10000 轮以后，外部轮廓已经可以明显感觉到了，只是还有些模糊，有些水彩画感觉；更新 20000 轮后生成的图片完全可以以假乱真，并且 50000 轮后生成的图片已经十分逼真了。

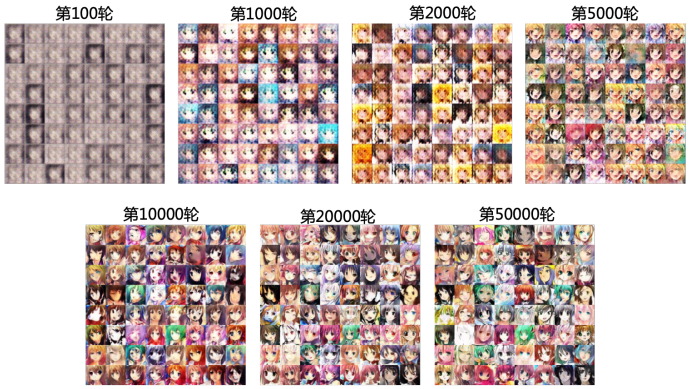


图 8.11　GAN 生成动画人物人脸的可视化效果

除了产生动画人物以外，当然也可以产生真实的人脸，如图 8.12产生高清人脸的技术，叫做**渐进式 GAN（progressive GAN）**，上下两排都是由机器产生的人脸。

同样，我们可以用 GAN 产生我们从没有看过的人脸，如图 8.13所示。举例来说，先前我们介绍的 GAN 中的生成器，就是输入一个向量，输出一张图片。此外，我们还可以把输入的向量做内差，在输出部分我们就会看到两张图片之间连续的变化。比如我们输入一个向量通过 GAN 产生一个看起来非常严肃的男人，同时输入另一个向量通过 GAN 产生一个微笑着的女人。那我们输入这两个向量中间的数值向量，就可以看到这个男人逐渐地笑了起来。另一个例子，输入一个向量产生一个往左看的人，同时输入一个向量产生一个往右看的人，我们在之间做内差，机器并不会傻傻地将两张图片叠在一起，而是生成一张正面的脸。神奇的是，我们在训练的时候其实并没有真的输入正面的人脸，但机器可以自己学到把这两张左右脸做内差，应该会得到一个往正面看的人脸。

不过如果我们不加约束，GAN 会产生一些很奇怪的图片，如图 8.14所示。比如我们使用 BigGAN 算法，会产生一个左右不对称的玻璃杯子，甚至产生一个网球狗，还是很有趣的。

图 8.12　渐进式 GAN 生成人脸的效果

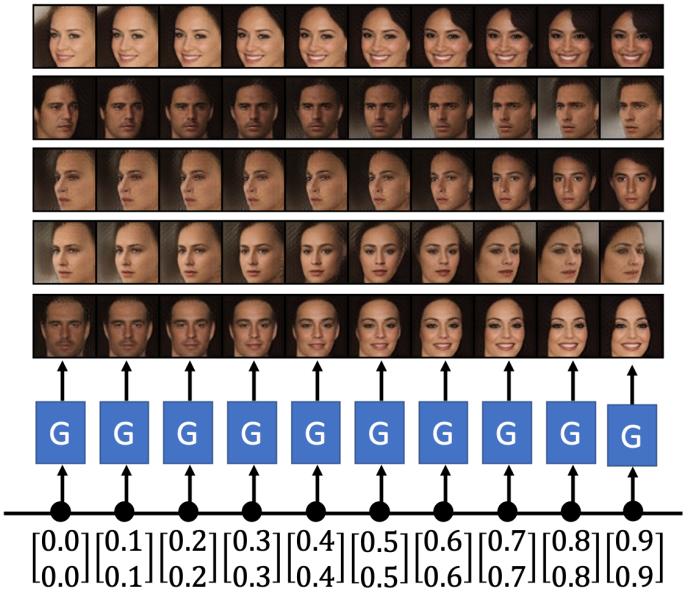


图 8.13　GAN 产生连续人脸的过程

8.4　GAN 的理论介绍

这一小节我们将从理论层面介绍 GAN，即为什么生成器与判别器的交互可以产生人脸图片。首先，我们需要了解训练的目标是什么。在训练网络时，我们要确定一个损失函数，然后使梯度下降策略来调整网络参数，并使得设定的损失函数的数值最小或最大即可。在8.1中我们介绍了，生成器的输入是一系列的从分布中采样出的向量，生成器就会产生一个比较复杂的分布，如图 8.15 所示，我们称之为 P_G。另外我们还有一系列的数据，这些原始的数据本身会形成另外一个分布，我们称之为 P_{data}。训练的效果是希望 P_G 和 P_{data} 尽可能的相似。

我们再举一个一维的简单例子说明 P_G 和 P_{data}，我们假设生成器的输入是一个一维的向量，如图 8.15的橙色曲线，生成器的输出也是一维的向量，如图 8.15的绿色曲线，真正的数据同样是一个一维的向量，它的分布用蓝色曲线来表示。假设每次我们输入 5 个点，那每一个点的位置就会随着训练次数而改变，就会产生一个新的分布。可能本来所有的点都集中在中间，但是通过生成器，通过一个网络里面很复杂的训练后，这些点就分成两边，变成图片中的分布的样子。而其中 P_{data} 就是指真正数据的分布，在实际应用中真正数据分布可能是更极端的，比如左边的数据比较多，右边的数据比较少。我们训练的结果是希望两个分布 P_G 和

图 8.14 GAN 产生不符合常理的可视化例子

P_{data} 越接近越好，即图片中的公式所示，表达的是这两个分布之间的差异，我们可以将其视为两个分布间的某种距离，如果这个距离越大，就代表这两个分布越不像；差异越小，代表这两个分布越相近。所以差异就是衡量两个的分布相似度的一个指标。我们现在的目标就是训练一组生成器模型中的网络参数，可以让生成的 P_G 和 P_{data} 之间的差异越小越好，这个最优生成器称为 G^*。

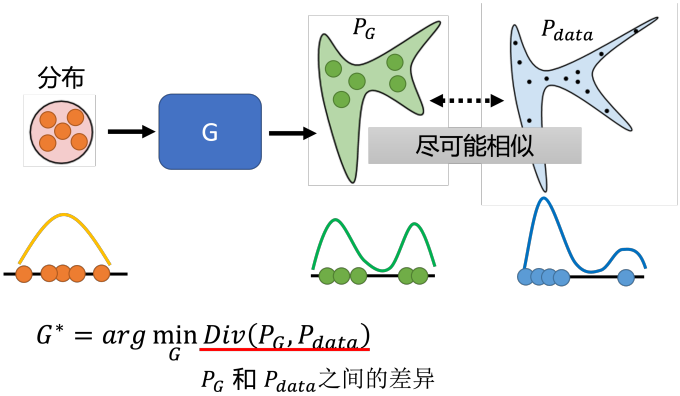


图 8.15 GAN 的训练目标

训练生成器的过程训练例如卷积神经网络等简单网络非常地像，相比于之前的找一组参数最小化损失函数，我们现在其实也定义了生成器的损失函数，即 P_G 和 P_{data} 之间的差异。对于一般的神经网络，其损失函数是可以计算的，但是对于生成器的差异，我们应该怎么处理呢？对于连续的差异例如 KL 散度和 JS 散度是很复杂的，在实际离散的数据中，我们或许无法计算其对应的积分。

对于 GAN，只要我们知道怎样从 P_G 和 P_{data} 中采样，就可以计算得到差异，而不需要知道实际的公式。例如，我们对于图库进行随机采样时，就会得到 P_{data}。对于生成器，我们需要从正态分布中采样出来的向量通过生成器生成一系列的图片，这些图片就是 P_G 采样出来的结果。所以我们有办法从 P_G 采样，也可以从 P_{data} 进行采样。接下来，我们将介绍如何只做以上采样的前提下，即不知道 P_G 和 P_{data} 的形式以及公式的情况下，如何估算得到差异，这其中要依靠判别器的力量。

我们首先回顾下判别器的训练方式。首先，我们有一系列的真实数据，也就是从 P_{data} 采样得到的数据。同时，还有一系列的生成数据，从 P_G 中采样出来的数据。根据真实数据和生

成数据，我们会去训练一个判别器，其训练目标是看到真实数据就给它比较高的分数，看到生成的数据就给它比较低的分数。我们可以把它当做是一个优化问题，具体来说，我们要训练一个判别器，其可以最大化一个目标函数，当然如果我们最小化它就可以称它为损失函数。这个目标函数如图 8.16所示，其中有一些 y 是从 P_data 中采样得到的，也就是真实的数据，而我们把这个真正的数据输入到判别器中，得到一个分数。另一方面，我们还有一些 y 来源于生成器，即从 P_G 中采样出来的，将这些生成图片输入至判别器中同样得到一个分数，再取 $Log(1-D(Y))$。

我们希望目标函数 V 越大越好，其中 y 如果是从 P_{data} 中采样得到的真实数据，它就要越大越好；如果是从 P_G 采样得到的生成数据，它就要越小越好。这个过程在 GAN 提出之初，人们将其写为这样其实还有一个缘由，就是为了让判别器和二分类产生联系，因为这个目标函数本身就是一个交叉熵乘上一个负号。训练一个分类器时的操作就是要最小化交叉熵，所以当我们最大化目标函数的时候，其实等同于最小化交叉熵，也就是等同于是在训练一个分类器。这个它做的事情就是把图 8.16 中蓝色点，从 P_{data} 采样出的真实数据当作类别 1，把从 P_G 采样出的这些假的数据当作类别 2。有两个类别的数据，训练一个二分类的分类器，训练后就等同于是解了这个优化问题。而图中红框里面的数值，它本身就和 JS 散度有关。或许最原始的 GAN 的文章，它的出发点是从二分类开始的，一开始是把判别器写成二分类的分类器然后有了这样的目标函数，然后再经过一番推导后发现这个目标函数的最大值和 JS 散度是相关的。

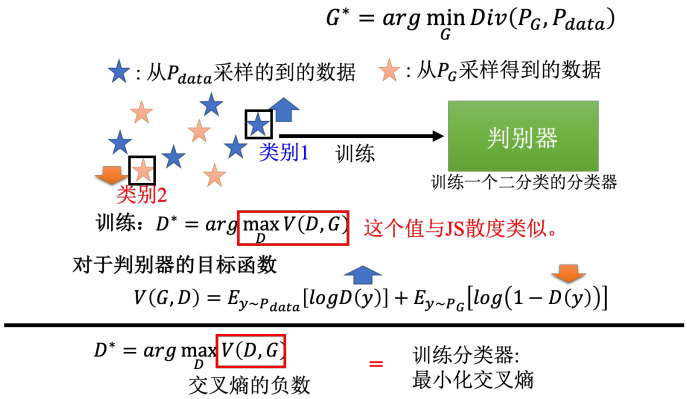


图 8.16　GAN 中判别器目标函数和优化过程

当然我们还是要直观理解下为什么目标函数的值会和散度有关。这里我们假设 P_G 和 P_{data} 的差距很小，就如图 8.16 所示蓝色的星星和红色的星星混在一起。这里，判别器就是在训练一个 0、1 分类的分类器，但是因为这两组数据差距很小，所以在解决这个优化问题时，就很难让目标函数 V 达到最大值。但是当两组数据差距很大时，也就是蓝色的星星和红色的星星并没有混在一起，那么就可以轻易地把它们分开。当判别器可以轻易把它们分开的时候，目标的函数就可以变得很大。所以当两组数据差距很大的时候，目标函数的最大值就可以很大。当然这里面有很多的假设，例如判别器的分类能力无穷大。

我们再来看下计算生成器 + 判别器的过程，我们的目标是要找一个生成器去最小化两个分布 P_G 和 P_{data} 的差异。这个差异就是使用训练好的判别器来最大化它的目标函数值来实现。最小和最大的 MinMax 过程就像是生成器和判别器进行互动，互相“欺骗”的过程。注意，这里的差异函数不一定使用 KL 或者 JS 等函数，可以尝试不同的函数来得到不同差异衡量指

标。

8.5 WGAN 算法

因为要进行 MinMax 操作，所以 GAN 是很不好训练的。我们接下来介绍一个 GAN 训练的小技巧，就是著名的**Wasserstein GAN (Wasserstein Generative Adversarial Network)**。在讲这个之前，我们分析下 JS 散度有什么问题。首先，JS 散度的两个输入 P_G 和 P_{data} 之间的重叠部分往往非常少。这个其实也是可以预料到的，我们从不同的角度来看：图片其实是高维空间里低维的流形，因为在高维空间中随便采样一个点，它通常都没有办法构成一个人物的头像，所以人物头像的分布，在高维的空间中其实是非常狭窄的。换个角度解释，如果是以二维空间为例，图片的分布可能就是二维空间的一条线，也就是 P_G 和 P_data 都是二维空间中的两条直线。而二维空间中的两条直线，除非它们刚好重合，否则它们相交的范围是几乎可以忽略的。从另一个角度解释，我们从来都不知道 P_G 和 P_data 的具体分布，因为其源于采样，所以也许它们是有非常小的重叠分布范围。比如采样的点不够多，就算是这两个分布实际上很相似，也很难有任何的重叠的部分。

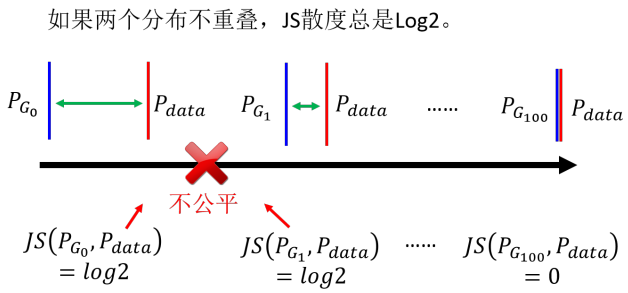


图 8.17 JS 散度的局限性

所以以上的问题就会对于 JS 分布造成以下问题：首先，对于两个没有重叠的分布，JS 散度的值都为 Log2，与具体的分布无关。就算两个分布都是直线，但是它们的距离不一样，得到的 JS 散度的值就都会是 Log2，如图 8.17 所示。所以 JS 散度的值并不能很好地反映两个分布的差异。另外，对于两个有重叠的分布，JS 散度的值也不一定能够很好地反映两个分布的差异。因为 JS 散度的值是有上限的，所以当两个分布的重叠部分很大时，JS 散度不好区分不同分布间的差异。所以既然从 JS 散度中，看不出来分布的差异。那么在训练的时候，我们就很难知道我们的生成器有没有在变好，我们也很难知道我们的判别器有没有在变好。所以我们需要一个更好的衡量两个分布差异的指标。

我们从更直观的实际操作角度来说明，当使用 JS 散度训练一个二分类的分类器，来去分辨真实和生成的图片时，会发现实际上正确率几乎都是 100%。原因在于采样的图片根本就没几张，对于判别器来说，采样 256 张真实图片和 256 张假的图片它直接用硬背的方法都可以把这两组图片分开。所以实际上如果用二分类的分类器训练判别器下去，识别正确率都会是 100%。根本就没有办法看出生成器有没有越来越好。所以过去尤其是在还没有 WGAN 这样的技术时，训练 GAN 真的就很像盲盒。根本就不知道训练的时候，模型有没有越来越好，所以旧时的做法是每次更新几次生成器后，就需要把图片打印可视化出来看。然后就要一边吃饭，一边看图片生成的结果，然后跑一跑就发现内存报错了就需要重新再来，所以过去训练 GAN 真的是很辛苦的。这也不像我们在训练一般的网络的时候，有个损失函数，然后那个损

失值随着训练的过程慢慢变小，当我们看到损失值慢慢变小时，我们就放心网络有在训练。但是对于 GAN 而言，我们根本就没有这样的指标，所以我们需要一个更好的衡量两个分布差异的指标。否则只能够用人眼看，用人眼守在电脑前面看，发现结果不好，就重新用一组超参数调一下网络。

既然是 JS 散度的问题，肯定有人就想问说会不会换一个衡量两个分布相似程度的方式，就可以解决这个问题了呢？是的，于是就有了 Wasserstein，或使用 Wasserstein 距离的想法。Wasserstein 距离的想法如下，假设两个分布分别为 P 和 Q，我们想要知道这两个分布的差异，我们可以想像有一个推土机，它可以把 P 这边的土堆挪到 Q 这边，那么推土机平均走的距离就是 Wasserstein 距离。在这个例子里面，我们假设 P 集中在一个点，Q 集中在一个点，对推土机而言，假设它要把 P 这边的土挪到 Q 这边，那它要平均走的距离就是 D，那么 P 和 Q 的 Wasserstein 距离就是 D。但是如果 P 和 Q 的分布不是集中在一个点，而是分布在一个区域，那么我们就要考虑所有的可能性，也就是所有的推土机的走法，然后看平均走的距离是多少，这个平均走的距离就是 Wasserstein 距离。Wasserstein 距离可以想象为有一个推土机在推土，所以 Wasserstein 距离也称为推土机距离（Earth Mover's Distance，EMD）。所以 Wasserstein 距离的定义如图 8.18 所示。

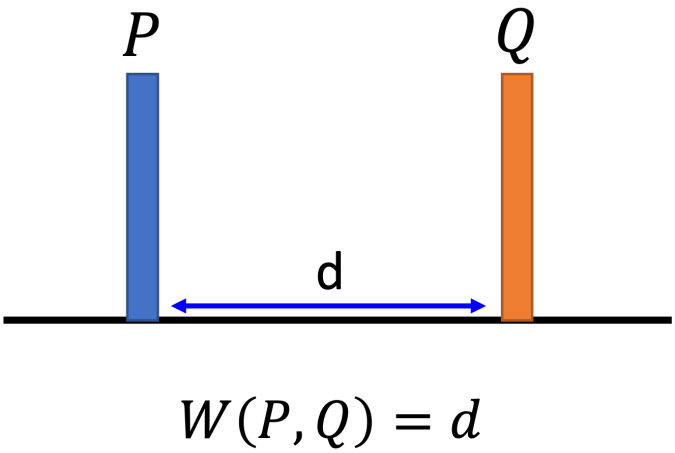


图 8.18　Wasserstein 距离的定义

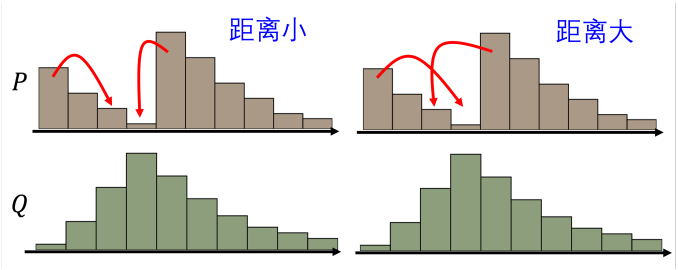


图 8.19　Wasserstein 距离的可视化理解

但是如果是更复杂的分布，要算 Wasserstein 距离就有点困难了，如图 8.19 所示。假设两个分布分别是 P 和 Q，我们要把 P 变成 Q，那有什么样的做法呢？我们可以把 P 的土搬到 Q 来，也可以反过来把 Q 的土搬到 P。所以当我们考虑比较复杂分布的时候，两种分

布计算距离就有很多不同的方法，即不同的“移动”方式，从中计算算出来的距离，即推土机平均走的距离就不一样。对于左边这个例子，推土机平均走的距离比较少；右边这个例子因为舍近求远，所以推土机平均走的距离比较大。那两个分布 P 和 Q 的 Wasserstein 距离会有很多不同的值吗？这样的话，我们就不知道到底要用哪一个值来当作是 Wasserstein 距离了。为了让 Wasserstein 距离只有一个值，我们将距离定义为穷举所有的“移动”方式，然后看哪一个推土的方法可以让平均的距离最小。那个最小的值才是 Wasserstein 距离。所以其实要计算 Wasserstein 距离挺麻烦的，因为里面还要解一个优化问题。

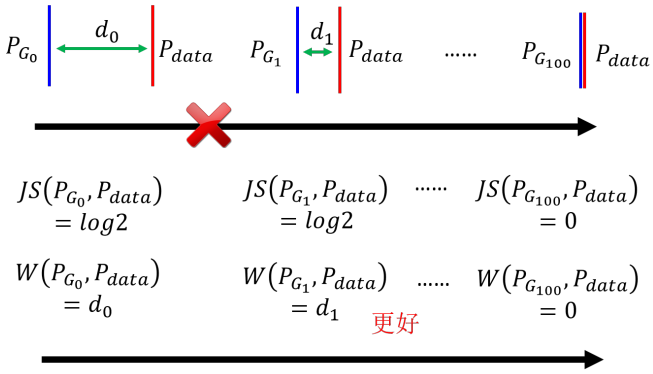


图 8.20　Wasserstein 距离与 JS 距离的对比

我们这里先避开这个问题，先来看看 Wasserstein 距离有什么好处，如图 8.20所示。假设两个分布 P_G 和 P_{data} 它们的距离是 d_0，那在这个例子中，Wasserstein 距离算出来就是 d_0。同样的，假设两个分布 P_G 和 P_{data} 它们的距离是 d_1，那在这个例子中，Wasserstein 距离算出来就是 d_1。假设 d_1 小于 d_0，那 d_1 的 Wasserstein 距离就会小于 d_0。所以 Wasserstein 距离可以很好地反映两个分布的差异。从左到右我们的生成器越来越进步，但是如果同时观察判别器，你会发现你观察不到任何规律。因为对于判别器而言，每一个例子算出来的 JS 散度，都是一样的 Log2，所以判别器根本就看不出来这边的分布有没有变好。但是如果换成 Wasserstein 距离，由左向右的时候我们会知道，我们的生成器做得越来越好。所以我们的 Wasserstein 距离越小，对应的生成器就越好。这就是为什么我们要用 Wasserstein 距离的原因，我们换一个计算差异的方式，就可以解决 JS 距离有可能带来的问题。

可以再举一个演化的例子——人类眼睛的生成。人类的眼睛是非常复杂的，它是由其他原始的眼睛演化而来的。比如说有一些细胞具备有感光的能力，这可以看做是最原始的眼睛。那这些最原始的眼睛怎么变成最复杂的眼睛呢？它只是一些感光的细胞在皮肤上经过一系列的突变产生更多的感光细胞，中间有很多连续的步骤。举例来说，感光的细胞可能会出现在一个比较凹陷的地方，皮肤凹陷下去，这样感光细胞可以接受来自不同方向的光源。然后慢慢地把凹陷的地方保护住并在里面放一些液体，最后就变成了人的眼睛。所以这个过程是一个连续的过程，是一个从简单到复杂的过程。当使用 WGAN 时，使用 Wasserstein 距离来衡量分布间的偏差的时候，其实就制造了类似的效果。本来两个分布 P_{G0} 和 P_{data} 距离非常遥远，你要它一步从开始就直接跳到结尾，这是非常困难的。但是如果用 Wasserstein 距离，你可以让 P_{G0} 和 P_{data} 慢慢挪近到一起，可以让它们的距离变小一点，然后再变小一点，最后就可以让它们对齐在一起。所以这就是为什么我们要用 Wasserstein 距离的原因，因为它可以让我们的生成器一步一步地变好，而不是一下子就变好。

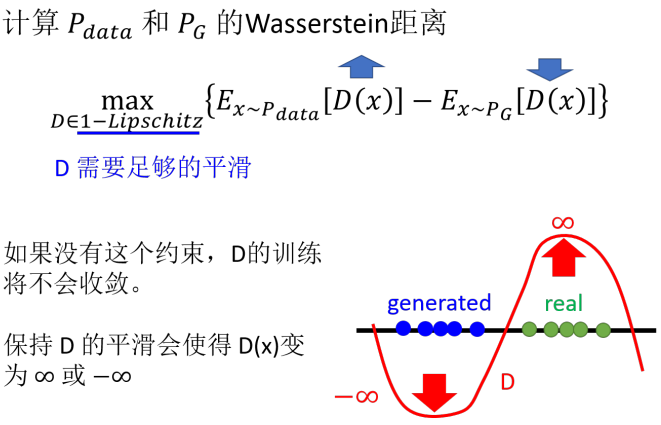


图 8.21 Wasserstein 距离的计算

所以 WGAN 实际上就是用 Wasserstein 距离来取代 JS 距离，这个 GAN 就叫做 WGAN。那接下来的问题是，Wasserstein 距离是要如何计算呢？我们可以看到，Wasserstein 距离的定义是一个最优化的问题，如图 8.21 所示。这里我们简化过程直接介绍结果，也就是解图中最大化问题的解，解出来以后所得到的值就是 Wasserstein 距离，即 P_{G0} 和 P_{data} 的 Wasserstein 距离。我们观察一下图 8.21 的公式，即我们要找一个函数 D，这个函数 D 是一个函数，我们可以想像成是一个神经网络，这个神经网络的输入是 x，输出是 $D(x)$。X 如果是从 P_{data} 采样来的，我们要计算它的期望值 $E_{x\sim P_{data}}$，如果 X 是从 P_G 采样来的，那我们要计算它的期望值 $E_{x\sim P_G}$，然后再乘上一个负号，所以如果要最大化这个目标函数就会达成。如果 X 如果是从 P_{data} 采样得到的，那么判别器的输出要越大越好，如果 X 如果是从 P_G 采样得到的，从生成器采样出来的输出应该要越小越好。

此外还有另外一个限制。函数 D 必须要是一个 1-Lipschitz 的函数。我们可以想像成，如果有一个函数的斜率是有上限的（足够平滑，变化不剧烈），那这个函数就是 1-Lipschitz 的函数。如果没有这个限制，只看大括号里面的值只单纯要左边的值越大越好，右边的值越小越好，那么在蓝色的点和绿色的点，也就是真正的图像和生成的图像没有重叠的时候，我们可以让左边的值无限大，右边的值无限小，这样的话，这个目标函数就可以无限大。这时整个训练过程就根本就没有办法收敛。所以我们要加上这个限制，让这个函数是一个 1-Lipschitz 的函数，这样的话，左边的值无法无限大，右边的值无法无限小，所以这个目标函数就可以收敛。所以当判别器够平滑的时候，假设真实数据和生成数据的分布距离比较近，那就没有办法让真实数据的期望值非常大，同时生成的值非常小。因为如果让真实数据的期望值非常大，同时生成的值非常小，那它们中间的差距很大，判别器的更新变化就很剧烈，它就不平滑了，也就不是 1-Lipschitz 了。

那接下来的问题就是如何确保判别器一定符合 1-Lipschitz 函数的限制呢？其实最早刚提出 WGAN 的时候也没有什么好想法。最早的一篇 WGAN 的文章做了一个比较粗糙的处理，就是训练网络时，把判别器的参数限制在一个范围内，如果超过这个范围，就把梯度下降更新后的权重设为这个范围的边界值。但其实这个方法并不一定真的能够让判别器变成 1-Lipschitz 函数。虽然它可以让判别器变得平滑，但是它并没有真的去解这个优化问题，它并没有真的让判别器符合这个限制。

后来就有了一些其它的方法，例如说有一篇文章叫做 Improved WGAN，它就是使用了**梯**

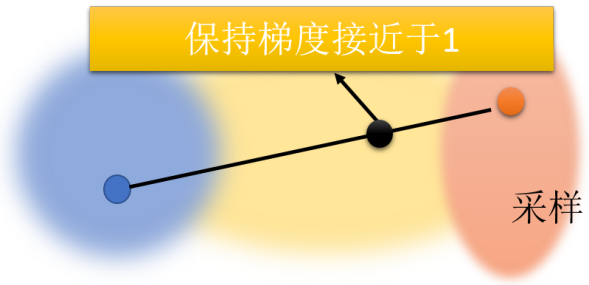


图 8.22 Improved WGAN 的梯度惩罚

度惩罚（gradient penalty）的方法，这个方法可以让判别器变成 1-Lipschitz 函数。具体来说，如图 8.22 所示，假设蓝色区域是真实数据的分布，橘色是生成数据的分布，在真实数据这边采样一个数据，在生成数据这边取一个样本，然后在这两个点之间取一个中间的点，然后计算这个点的梯度，使之接近于 1。就是在判别器的目标函数里面，加上一个惩罚项，这个惩罚项就是判别器的梯度的范数减去 1 的平方，这个惩罚项的系数是一个超参数，这个超参数可以让你的判别器变得越平滑。在 Improved WGAN 之后，还有 Improved Improved WGAN，就是把这个限制再稍微改一改。另外还有方法是将判别器的参数限制在一个范围内，让它是 1-Lipschitz 函数，这个叫做谱归一化。总之，这些方法都可以让判别器变成 1-Lipschitz 函数，但是这些方法都有一个问题，就是它们都是在判别器的目标函数里面加了一个惩罚项，这个惩罚项的系数是一个超参数，这个超参数会让你的判别器变得越平滑。

8.6 训练 GAN 的难点与技巧

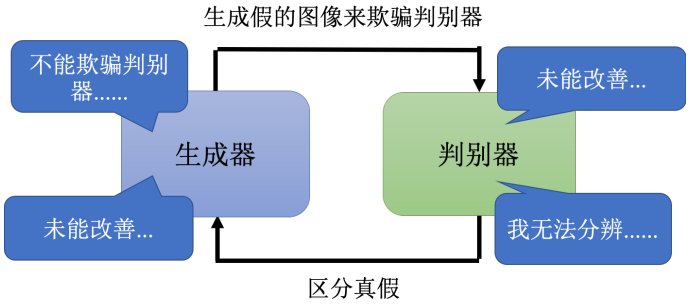


图 8.23 GAN 训练的难点

GAN 是以很难训练而闻名的，我们接下来介绍一些其中的原因和训练 GAN 的小技巧。首先，我们回顾一下判别器和生成器都在做些什么。判别器的目标是要分辨真的图片与产生出来的假图片间的差异，而生成器在做的事情是要去产生假的图片，骗过判别器。而事实上这两个网络，生成器和判别器它们是互相砥砺才能互相成长的，如图 8.23 所示。因为如果判别器太强了，那么生成器就会很难骗过它，生成器就会很难产生出真的图片。但是如果生成器太强了，那么判别器就会很难分辨真图片和假图片。只要其中一者发生什么问题停止训练，另外一个就会跟着停下训练就会跟着变差。假设在训练判别器的时候一下子没有训练好，那么判别器没有办法分辨真的跟产生出来的图片的差异，同时生成器就失去了可以进步的目标，生成器就没有办法再进步了。那么判别器也会跟着停下来，所以这两个网络是互相砥砺才能

互相成长的。所以这也是为什么 GAN 很难训练的原因，因为这两个网络必须要同时训练，而且必须要同时训练到一个比较好的状态。

所以 GAN 本质上它的训练仍然不是一件容易的事情，当然它是一个非常重要的前瞻技术。有一些训练 GAN 的小技巧，例如 Soumith、DCGAN、BigGAN 等等。大家可以自己看看相关文献进行尝试。

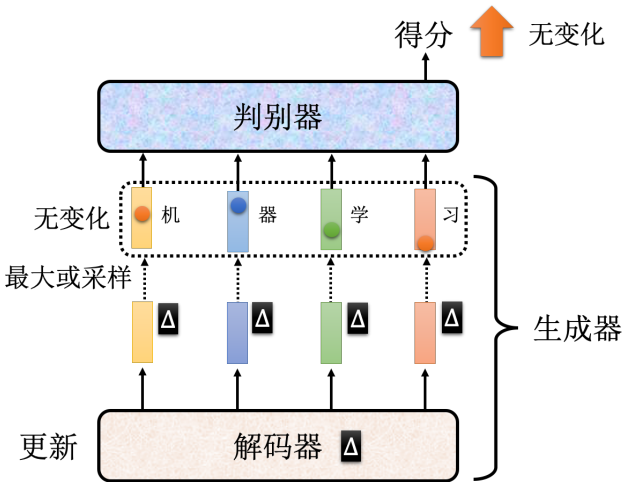


图 8.24 序列生成的 GAN

训练 GAN 最难的一个领域其实是要拿 GAN 来生成文字。如果要生成一段文字那需要一个序列到序列的模型，其中的一个解码器会产生一段文字，如图 8.24 所示。这个序列到序列的模型就是我们的生成器。著名的 Transformer 就是一个解码器，它现在在 GAN 里面，就扮演了生成器的角色，负责产生我们要它产生的东西，比如说一段文字。那这个序列到序列的 GAN 和原来的用于图像中的 GAN 有什么不同呢？就最高层次来看，就算法来讲它们没有太大的不同。因为本质上就是训练一个判别器，判别器把这段文字读进去，去判断说这段文字是真正的文字还是机器产生出来的文字。而解码器就是想办法去骗过判别器，生成器就是想办法去骗过判别器，我们来调整生成器的参数，想办法让判别器觉得生成器产生出来的东西是真正的文字。所以从算法的角度来讲，它们没有太大的不同。对于序列到序列的模型其真正的难点在于，如果要用梯度下降去训练解码器，去让判别器输出得分越大越好，你会发现很难做到。我们来思考下，假设我们改变了解码器的参数，这个生成器，也就是解码器的参数，有一点小小的变化的时候，到底对判别器的输出有什么样的影响。如果解码器的参数有一点小小的变化，那它现在输出的分布也会有小小的变化，那因为这个变化很小，所以它对于输出的词元不会有很大的影响。

其中词元就是现在在处理产生这个序列的单位。假设我们今天，在产生一个中文的句子的时候，我们是每次产生一个汉字，那这每一个汉字就是我们的词元。在处理英文的时候，每次产生一个英文的字母，那字母就是你的词元。所以词元就是你产生一个序列的单位，那这个单位是你自己定义的。假设你一次是产生一个英文的词，英文的词和词之间是以空白分开的，那就是词就是你的词元。

我们回到刚才的讨论，假设输出的分布只有小小的变化，并且在取最大值的时候，或者说在找分数最大那个词元的时候，你会发现分数最大的那个词元是没有改变的。输出的分布只有小小的变化，所以分数最大的那个词元是同一个。那对于判别器来说，它输出的分数是

没有改变的。判别器输出也不会改变，所以你根本就没有办法算微分，也根本就没有办法做梯度下降。当然就算是不能做梯度下降，我们还是可以用强化学习的方法来训练生成器。但是强化学习本身是以难训练而闻名，GAN 也是以难训练而闻名，这样的东西加在一起，就会非常非常地难训练。所以要用 GAN 产生一段文字，在过去一直被认为是一个非常大的难题。所以有很长一段时间，没有人可以成功地把生成器训练起来产生文字。

直到有一篇文章叫做 ScratchGAN，不需要**预训练（pre-training）**，可以直接从随机的初始化参数开始，训练生成器，然后让生成器可以产生文字。它的方法是调节超参数，并且加上一些训练技巧，就可以从零开始训练生成器。里面的技巧比如说要用 SeqGAN-Step 的技术，并且将训练批大小设置的很大，要上千，然后要用强化学习的方法，要改一下强化学习的参数，同时加一些正则化等等技巧，就可以从真的把 GAN 训练起来，然后让它来产生序列。

此外，其实有关生成式的模型不是只有 GAN 而已，还有其他的比如 VAE，比如流模型等等，这些模型都有各自的优缺点。当然，就假设目前想要训练一个生成器，想让机器可以生成一些东西还是那有很多方法，可以用 GAN，可以 VAE，也可以用流模型。但是如果我们想要产生一些图片，那就最好用 GAN，因为 GAN 是目前为止比较好的生成式的模型，它可以产生最好的图片。但是如果想要产生一些文字，那就只有用 VAE 或者流模型，因为 GAN 在产生文字的时候，还是有一些问题。从训练角度，你可能会觉得 GAN 从式子上看起来有一个判别器和生成器，它们要互动。然后像流模型和 VAE 它们都比较像是直接训练一个一般的模型，有一个很明确的目标，不过实际上训练时它们也没有那么容易成功地被训练起来。因为它们的分类里面有很多项，它们的损失函数里面有很多项，然后把每一项都平衡才能够有好的结果，但要达成平衡也非常地困难。

那为什么我们需要用生成式来做输出新图片的事情呢？如果我们今天的目标就是，输入一个高斯分布的变量，然后使用采样出来的向量，直接输出一张照片，能不能直接用监督学习的方式来实现呢？具体做法比如我有一堆图片，每一个图片都去配一个向量，这个向量来源于从高斯分布中采样得到的向量，然后我就可以用监督学习的方式来训练一个网络，这个网络的输入是这个向量，输出是这个图片。确实能这么做，也真的有这样的生成式模型。但是难点在于，如果纯粹放随机的向量，那训练起来结果会很差。所以需要有一些特殊的方法例如生成式潜在优化等方法，供大家参考。

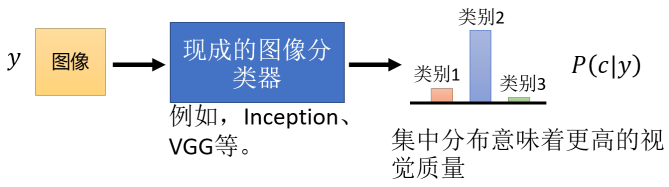


图 8.25　评估 GAN 生成图像的质量

8.7　GAN 的性能评估方法

接下来我们介绍 GAN 的评估方法，也就是我们产生出来的生成器它好或者是不好。要评估一个生成器的好坏，最直觉的做法也许是找人来看生成器产生出来的图片到底像不像真实的图片。所以其实很长一段时间，尤其是人们刚开始研究生成式技术的时候，很长一段时间没有好的评估方法。那时候要评估生成器的好坏，都是人眼看，直接在论文最后放几张图片，

然后说这个生成器产生出来的图片是不是比较好。所以我们可以发现比较早年 GAN 的论文，它没有数字结果，整篇论文里面没有准确度等等的数字结果，它只有一些图片，然后说这个生成器产生出来的图片是不是比较好，接着就结束了。这样显然是不行的，并且有很多的问题，比如说不客观、不稳定等等诸多的问题。所以有没有比较客观而且自动的方法来度量一个生成器的好坏呢？

针对特定的一些任务，是有办法设计一些特定方法的。比如说我们要产生一些动画人物的头像，那我们可以设计一个专门用于动画人物面部的识别系统，然后看看我们的生成器产生出来的图片里面，有没有可以被识别的动画人物的人脸。如果有的话，那就代表说这个生成器产生出来的图片是比较好的。但是这个方法只能针对特定的任务，如果我们要产生的东西不是动画人物的头像，而是别的东西，那这个方法就不行了。那如果是更一般的案例，比如它不一定是产生动画人物的，它专门产生猫、专门产生狗、专门产生斑马等等，那我们怎么知道它做得好不好呢？

其实有一个方法，是训练一个图像的分类系统，然后把 GAN 产生出来的图片输入到这个图像的分类系统里面，看它产生什么样的结果，如图 8.25 所示。这个图像分类系统的输入是一张图片，输出是一个概率分布，这个概率分布代表说这张图片是猫的概率、狗的概率、斑马的概率等等。如果这个概率分布越集中，就代表现在产生的图片可能越好。如果生成出来的图片是一个四不像，图像识别系统就会非常地困惑，它产生出来的这个概率分布就会是非常平均地分布。

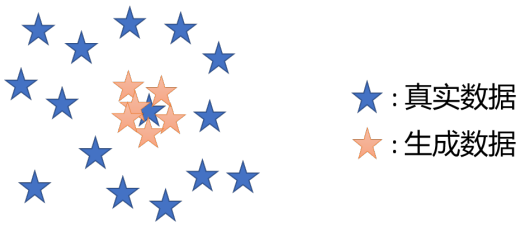


图 8.26　模式崩塌问题

这个是靠图像识别系统来判断产生出来的图片好坏，这是一个可能的做法，但是光用这个做法是不够的。光用这个做法会被一个叫做**模式崩塌（mode collapse）**的问题骗过去。模式崩塌是指在训练 GAN 的过程中遇到的一个状况，假设如图 8.26 蓝色的星星是真正的数据的分布，红色的星星是 GAN 的模型的分布。我们会发现生成式的模型它输出来的图片来来去去就是那几张，可能单一张拿出来你觉得好像还做得不错，但让它多产生几张就露出马脚，产生出来就只有那几张图片而已，这就是模式崩塌的问题。

发生模式崩塌的原因，从直觉上理解可以想成这个地方就是判别器的一个盲点，当生成器学会产生这种图片以后，它就永远都可以骗过判别器，判别器没办法看出来图片是假的。那对于如何避免模式坍塌，其实到今天其实还没有一个非常好的解答，不过有方法是模型在生成器训练的时候，一直将训练的节点存下来，在模式坍塌之前把训练停下来，就只训练到模式崩塌前，然后就把之前的模型拿出来用。不过模型崩塌这种问题，我们至少是知道有这个问题，是可以看得出的，生成器总是产生这张脸的时候，你不会说你的生成器是个好的生成器。但是有一些问题是你不知道的并且更难侦测到的，即你不知道生成器产生出来的图片是不是真的有多样性。

这个问题叫做模式丢失，指 GAN 能很好地生成训练集中的数据，但难以生成非训练集的

数据，“缺乏想象力”。你的产生出来的数据，只有真实数据的一部分，单纯看产生出来的数据，你可能会觉得还不错，而且分布的这个多样性也够，但你不知道真实数据的多样性的分布其实是更大的。事实上今天这些非常好的 GAN，BGAN、ProgressGAN 等等可以产生非常真实人脸这些 GAN，多多少少还是有模式丢失的问题。如果你看多了 GAN 产生出来的人脸，你会发现虽然非常真实但好像来来去去就是那么几张脸而已，并且有一个非常独特的特征是你看多了以后就觉得，这个脸好像是被生成出来的。今天也许模式丢失都还没有获得本质上的解决。

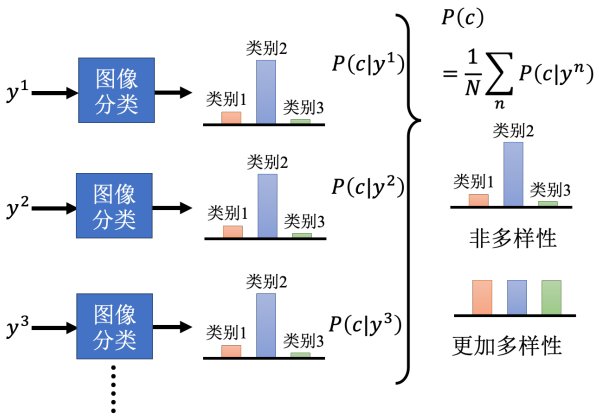


图 8.27　GAN 生成结果多样性问题

虽然存在以上模式坍塌、模式丢失等等的问题，但是我们需要去度量生成器产生出来的图片到底多样性够不够。有一个做法是借助我们之前介绍过的图像分类，把一系列图片都丢到图像分类器里，看它被判断成哪一个类别，如图 8.27 所示。每张图片都会给我们一个分布，我们将所有的分布平均起来，接下来看看平均的分布长什么样子。如果平均的分布非常集中，就代表现在多样性不够，如果平均的分布非常平坦，就代表现在多样性够了。具体来讲，如果什么图片输入到图像分类系统中的输出都是第二种类别，那代表说每一张图片也许都很像，也就代表输出的多样性是不够的，那如果另外一个案例不同张图片丢进去，它的输出分布都不一样，那就代表说多样性是够的。并且平均完以后发现结果是非常平坦的，那这个时候代表多样性是足够的。

当我们用这个图像分类器来做评估的时候，对于结果的多样性和质量好像是有点互斥的。因为我们刚才在讲质量的时候说，分布越集中代表质量越高，多样性的分布越平均。但是如果分布越平均，那质量就会越低，因为分布越平均，代表图片都不太像，所以质量就会越低。这里要强调一下质量和多样性的评估范围不一样，质量是只看一张图片，一张图片丢到分类器的时候，分布有没有非常地集中。而多样性看的是一堆图片分布的平均，一堆图片中图像分类器输出的越平均，那就代表现在的多样性越大。

过去有一个非常常被使用的分数，叫做 Inception 分数。其顾名思义就是用 Inception 网络来做评估，用 Inception 网络度量质量和多样性。如果质量高并且多样性又大，那 Inception 分数就会比较大。目前研究人员通常会采取另外一个评估方式，叫 Fréchet Inception distance (FID)。具体来讲，先把生成器产生出来的人脸图片，丢到 InceptionNet 里面，让 Inception 网络输出它的类别。这里我们需要的不是最终的类别，而是进入 Softmax 之前的隐藏层的输

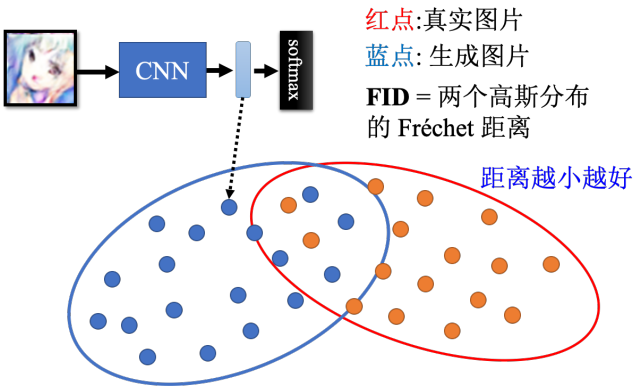


图 8.28 FID 的计算过程

出向量，这个向量的维度是上千维的，代表这个图片，如图 8.28 所示。图中所有红色点代表把真正的图片丢到 Inception 网络以后，拿出来的向量。这个向量其实非常高维度，甚至是上千维的，我们就把它降维后画在二维的平面上。蓝色点是 GAN 的生成器产生出来的图片，它丢到 Inception 网络以后进入 Softmax 之前的向量。接下来，我们假设真实的图片和生成的图片都服从高斯分布，然后去计算这两个分布之间的 Fréchet 的距离。两个分布间的距离越小越好，距离越小越代表这两组图片越接近，也就是产生出来的品质越高。这里还有几个细节问题，首先，假设为高斯分布没问题吗？另外一个问题是如果要准确的得到网络的分布，那需要产生大量的采样样本才能做到，这需要一点运算量，也是做 FID 不可避免的问题。

FID 算是目前比较常用的一种度量方式，那有一篇文章叫做“Are GANs Created Equal? A Large-Scale Study”，这个 Google 完成的论文里面尝试了不同的 GAN。每一个 GAN 的训练的分类，训练的损失都有点不太一样，并且每一种 GAN，它都用不同的随机种子，去跑过很多次以后，取结果的平均值等等。从文章的结果来看所有的 GAN 都差不多，那所以与 GAN 有关的研究都是白忙一场吗？事实上也未必是如此，这篇文章做实验的时候不同的 GAN 用的网络架构都是同一个，只是疯狂调参而已，调随机种子和学习率而已。网络架构还是同一个，所以是不是有某些网络架构，某些种类的 GAN 会不会在不同的网络架构上表现得比较稳定。这些都有待研究。

此外，还有一个状况。假设 GAN 产生出来的图片，跟真实的图片长得一模一样，那此时 FID 会是零，因为两个分布是一模一样的。如果你不知道真实数据长什么样子，光看这个生成器的输出可能会觉得太棒了，那 FID 算出来一定是非常小的，但是如果它产生出来的图片都跟数据库里面的训练数据的一模一样的话，那干脆直接从训练数据集里面采样一些图像出来不是更好，也就不需要训练生成器了。我们训练生成器其实是希望它产生新的图片，也就是训练集里面没有的人脸。

对于这种问题，就不是普通的度量标准可以侦测的。那怎么解决呢？其实有一些方法，例如可以用一个分类器，这个分类器是用来判断这张图片是不是真实的，是不是来自于你的训练集的。这个分类器的输入是一张图片，输出是一个概率，这个概率代表说这张图片是不是来自于你的训练集。如果这个概率是 1，那就代表说这张图片是来自于你的训练集，如果这个概率是 0，那就代表说这张图片不是来自于你的训练集。但是另外一个问题，假设生成器学到的是把所有训练数据里面的图片都左右反转呢，那它也是什么事都没有做。但是你的分类器会觉得说，这张图片是来自于你的训练集，因为它是来自于你的训练集的图片，只不过是左右反

转而已。进行分类时或者进行相似度的比较时，又比不出来。所以 GAN 的评估是非常地困难的，还甚至如何评估一个生成器做得好不好都是一个可以研究的题目。

8.8 条件型生成

下面，我们要介绍**条件型生成（conditional generation）**。我们之前讲的 GAN 中的生成器，它没有输入任何的条件，它只是输入一个随机的分布，然后产生出来一张图片。我们现在想要更进一步的是希望可以操控生成器的输出，我们给它一个条件 x，让他根据条件 x 跟输入 z 来产生输出 y。那这样的条件型生成器有什么样的应用呢？比如可以做文字对图片的生成，那如果要做文字对图片的生成，它其实是一个监督学习的问题，我们需要一些有标签的数据，需要去搜集一些人脸图片，然后这些人脸都要有文字的描述。比如这个样本是红眼睛、黑头发，另一个是黄头发、有黑眼圈等等标签的样本，才能够训练这种条件型生成器。所以在文字变图像这样的任务里面，我们的条件 x 就是一段文字。我们希望输入一段文字，然后生成器就可以产生出来一张图片，这张图片就是这段文字所描述的图片。那一段文字怎么输入给生成器呢，其实依赖于你自己。以前会用 RNN 把它读过去，然后得到一个向量，再丢到生成器中。今天也许你可以把它丢到一个 Transformer 的编码器中，然后得到一个向量，再丢到生成器中，只要能够让生成器读一段文字就行。我们期待的模型是输入“红眼睛”，然后机器就可以画一个红眼睛的角色，而且每次画出来的角色都不一样。那这个画出来什么样的角色，取决于你采样到到什么样的 z。采样到不一样的 z，就会画出不一样的角色，但是这个角色都是红眼睛的。那这个就是条件型生成的应用，文字变图像。

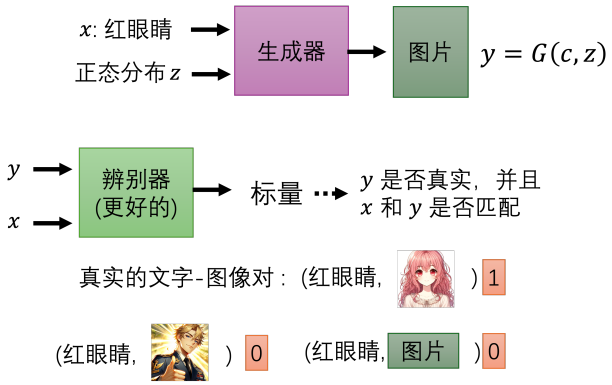


图 8.29 条件型 GAN

具体怎么做条件型的 GAN 呢？我们现在的生成器有两个输入，一个是正态分布中采样出来的 z，另一个是条件 x 也就是一段文字，然后它会产生出来一个 y，也就是一张图片。同时，我们需要一个判别器。如果按照我们过去介绍的知识，判别器就是使用一张图片 y 当作输入，输出一个数值，这个数值代表输入的图片多像真实的图片。训练这个判别器的方法就是，如果看到真实的图片就输出 1，如果看到生成的图片就输出 0。这样就可以训练判别器，然后判别器跟生成器反复训练。

但是这样的方法没办法真的解条件型的 GAN 的问题，因为如果我们只有训练这个判别器，只会将 y 当做输入的话，那生成器会学到的东西就是，只要产生出来的图片 y 好，但是跟输入完全没有任何关系，因为对生成器来说它只要产生清晰的图片就可以骗过判别器了。它何必要去管输入的文字叙述是什么呢，所以直接就无视这个条件 x，直接产生一个图片骗过

判别器就结束了。但这显然不是我们要的，所以在条件型的 GAN 里面，就要做有点不一样的设计，也就是判别器不是只吃图片 y，同时还要吃条件 x。同时判别器输出的数值不只是看 y 好不好，还要看 y 跟 x 配不配。如果 y 跟 x 配不上，那就要给一个很低的分数，如果 y 跟 x 配上，那就要给一个很高的分数。我们需要文字和图像成对的数据来训练判别器，所以条件型的 GAN，一般的训练是需要这个成对的标注数据的。所以当看到这些真正的成对数据，就给它 1 分，看到红色眼睛但是文字叙述是黑色头发，就给它 0 分，看到黑色头发但是文字叙述是红色眼睛，也给它 0 分，这样就可以训练判别器了，如图 8.29 所示。

那其实在实际操作中，只是拿这样的负样本对和正样本对来训练判别器，得到的结果往往不够好。往往还需要加上一种不好的状况：已经产生好的图片但是文字叙述配不上的状况。所以通常会把我们的训练数据拿出来，然后故意把文字跟图片乱配，或者故意配一些错的，然后告诉判别器看到这种状况，也是要输出不匹配。用这样子的数据，才有办法把判别器训练好。然后生成器跟判别器反复的训练，最后才会得到好的结果，这个就是条件型的 GAN。

在目前的例子里面都是看一段文字产生图片，当然条件型的 GAN 的应用不只看一段文字产生图片，比如也可以看一张图片产生其他图片。比如，给 GAN 房屋的设计图，然后让生成器直接把房屋产生出来，或者给它黑白的图片然后让它把颜色着上，或者给它素描的图，让它把它变成实景实物，再或者给它白天的图片，让它变成晚上的图片，给它起雾的图片，让它变成没有雾的图片等等。那像这样子的应用，叫做图像翻译，也就是输入一张图片，然后产生出来另外一张图片，也叫做 Pix2pix。那跟刚才讲的从文字产生图像也没有什么不同，现在只是从图像产生图像，把文字的部分用图像取代掉而已。所以其中同样要产生生成器，产生一张图片，然后要产生判别器，判别器要输入两张图片，然后输出一个数值。其中可以用监督学习的方法，训练一个图片生成图片的生成器，但是可能生成的结果图片非常地模糊，原因在于同样的输入可能对应到不一样的输出。生成器学到的就是把不同的可能平均起来，结果变成一个模糊的结果。所以这个时候我们的判别器它是输入是一张照片和输入条件，同时看图片和条件看有没有匹配来决定它的输出。另外，如果单纯用 GAN 的话它有一个小问题，它产生出来的图片比较真实，但是与此同时它的创造力、想像力过度丰富，会产生一些输入没有的东西。所以如果要做到最好往往就是 GAN 跟监督学习同时使用，就是说你的生成器不只要骗过判别器，同时你的生成器还要产生出来的图片跟标准答案越像越好。

条件型 GAN 还有很多应用，比如给 GAN 听一段声音，然后产生一个对应的图片。比如说给它听一段狗叫声，GAN 可以画出一只狗。这个应用的原理跟刚才讲的文字变图像是一样的，只是输入的条件变成声音而已。对于标签的成对数据就是声音和图像成对的数据，这个并没有很难搜集，因为可以爬到大量的影片，影片里面有图像有画面也有声音，并且每一帧是一一对应的，所以可以用这样的数据来训练。另外条件型 GAN 还可以产生会动的图片，比如给 GAN 一张蒙娜丽莎的画像，然后就可以让蒙娜丽莎开始讲话等等。

8.9 Cycle GAN

这一小节，我们要介绍一个 GAN 的另一个有趣的应用，就是把 GAN 用在无监督学习上。到目前为止呢，我们介绍的几乎都是监督学习，即我们要训练一个网络，其输入是 x，输出为 y，并且我们需要成对的数据才有办法训练网络。但是我们可能会遇到一个状况是，我们有一系列的输入和输出，但是 x 和 y 之间并没有成对的关系，也就是说我们没有成对的数据。举一个例子，比如图像风格转换的情况我们就没有成对的数据。我们有一系列真实的照片，然后有一些动漫的头像，我们希望可以把真实的照片转换成动漫的头像。具体来讲，假设我们要

训练一个深度学习的网络，它要做的事情是把 x 域的真人照片，转换为 y 域的动漫人物的头像。在这个例子里面我们或许就没有任何的成对的数据，因为我们有一堆真人的照片，但是我们没有这些真人的动漫头像。除非我们将动漫的头像先自己画出来，我们才有办法训练网络，不过这个做法显然实在是太昂贵了。所以对于图像风格转换，我们可能一点成对的数据都没有。那么在这种状况下，还有没有办法训练一个网络输入一个 x 产生一个 y 呢？这个时候就可以用到 GAN，在这种完全没有成对数据的情况下进行学习。

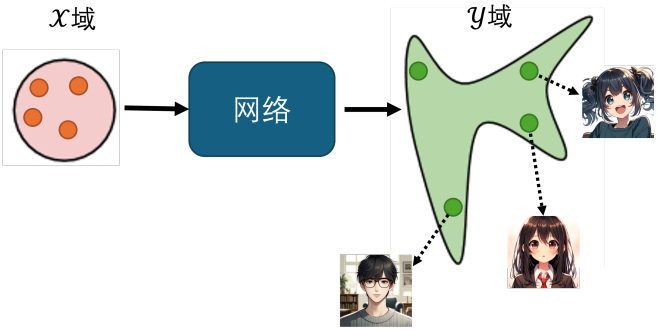


图 8.30　从无成对数据中学习的 GAN

如图 8.30 ，这个是我们之前在介绍无条件生成的时候所使用的生成器架构，输入是一个高斯的分布，输出可能是一个复杂的分布。现在稍微转换一下我们的想法，输入不是高斯分布，而是 x 域的图片的分布，y 域是图片的分布。我们完全可以套用原来的 GAN 的想法，即在原来的 GAN 中从高斯分布中采样一个向量输入到生成器里面，之前输入是 x 域的分布，我们只要改成可以从 x 域中采样就可以了。其实都不一定要从高斯分布采样，只要是一个分布，我们都可以从这个分布中采样一个向量丢到生成器里面，我们选择高斯分布只是因为它的公式我们是知道的。这个采样过程可以理解为就从真实的人脸里面随便挑一张出来，然后把这个照片丢到生成器里面，让它产生另外一张图片（分布）。这个时候我们的判别器就要改一下，判别器不再是只输入 y 域的图片，而是同时输入 x 域的图片和 y 域的图片，然后输出一个数值，这个数值代表这两张图片是不是一对的。

这整个过程与之前的 GAN 没有什么区别，但是仔细想想看只是套用原来的 GAN 训练，好像是不够的。因为我们要做的事情是要让这个生成器输出一张 y 域的图，但是它输出的 y 域的图一定要跟输入有关系吗？此处我们没有做任何的限制，所以生成器也许就把这张图片当作一个符合高斯分布的噪音，然后不管你输入什么它都无视它，只要判别器觉得它做得很好就可以了。所以如果我们完全只套用一般 GAN 的做法，只训练一个生成器，这个生成器输入的分布从高斯分布变成 x 域的图片，然后训练一个判别器，这样显然是不够的。因为训练出来的生成器，它可以产生的人物头像跟输入的真实的照片没有什么特别的关系，这个也不是我们要的。

那怎么解决这个问题呢？怎么强化输入与输出的关系呢？我们在介绍条件型 GAN 的时候，提到说假设判别器只看 y，那它可能会无视生成器的输入，所以产生出来的结果不是我们要的。所以我们要让判别器看 x 和 y，这样才可以让生成器学到 x 和 y 之间的关系。但是目前如果我们要从没有样本对的数据中学习，我们也没有办法直接套用条件型 GAN 的想法，因为在条件型 GAN 里面我们是有成对的数据的，可以用这些成对的数据来训练的判别器。但目前我们没有成对的数据来告诉判别器，怎么样的 x 和 y 的组合才是对的。为了解决这个问

题，我们可以使用**循环生成对抗网络（Cycle GAN）**。

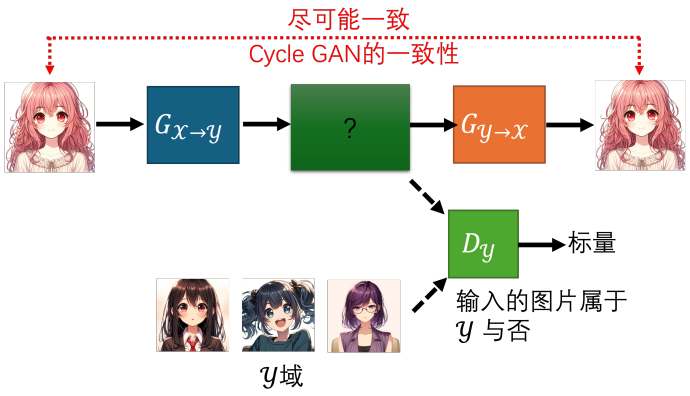


图 8.31　Cycle GAN 的基本架构

具体来说，在 Cycle GAN 中，我们会训练两个生成器。第一个生成器是把 x 域的图变成 y 域的图，第二个生成器它的工作是看到一张 y 域的图，把它还原回 x 域的图。在训练的时候，我们会增加一个额外的目标，就是我们希望输入一张图片，其从 x 域转成 y 域以后，要从 y 域转回原来一模一样的 x 域的图片。就这样经过两次转换以后，输入跟输出要越接近越好，或者说两张图片对应的两个向量之间的距离越接近越好。因为这边有一个循环，从 x 到 y 再从 y 回到 x，所以它是一个循环，所以被称为 Cycle GAN。而因为输入经过两次转换以后变成输出，并且需要保证输入跟输出越接近越好，所以这个过程也被称为 Cycle GAN 的一致性。所以在 Cycle GAN 中我们有三个网络，第一个生成器的工作是把 x 转成 y，第二个生成器的工作是要把 y 还原回原来的 x，另一个判别器的工作仍然是要看第一个生成器的输出像不像是 y 域的图，如图 8.31 所示。

那加入了第二个生成器以后，对于前面这个第一个的生成器来说，它就不能够随便产生与输入没有关系的人脸了。因为如果它产生出来的人脸跟输入的人脸没有关系，那第二个生成器就无法把它还原回原来的 x 域的图片。所以对第一个生成器来说，为了要让第二个生成器能够成功还原原来的图片，它产生出来的图片就不能跟输入差太多，然后第二个生成器才能够还原回原来的输入。

另外有一个问题，我们需要只保证第一个生成器的输出和输入有一定的关系，但是我们怎么知道这个关系是所需要的呢？机器自己有没有可能学到很奇怪的转换并且满足 cycle 的一致性呢？比如一个很极端的例子，假设第一个生成器学到的是把图片左右翻转，那只要第二个生成器学到把图片左右翻转就可以还原了啊。这样的话，第一个生成器学到的东西跟输入的图片完全没有关系，但是第二个生成器还是可以还原回原来的图片。所以如果我们做 Cycle GAN，用 cycle 的一致性的话似乎没有办法保证我们输入跟输出的人脸看起来很像，因为也许机器会学到很奇怪的转换，反正只要第二个生成器可以转得回来就好了。面对这个问题目前确实没有什么特别好的解法，但实际上使用 Cycle GAN 的时候，这种状况没有那么容易出现。输入和输出往往真的就会看起来非常像，甚至实际应用时就算没有第二个生成器，不用 cycle GAN，使用一般的 GAN 替代这种图片风格转换的任务，往往效果也很好。因为网络其实非常“懒惰”，输入一个图片它往往就想输出默认的与输入很像的东西，它不太想把输入的图片做太复杂的转换。所以在实际应用中，Cycle GAN 的效果往往非常好，而且输入和输出往往真的就会看起来非常像，或许只是改变了风格而已。

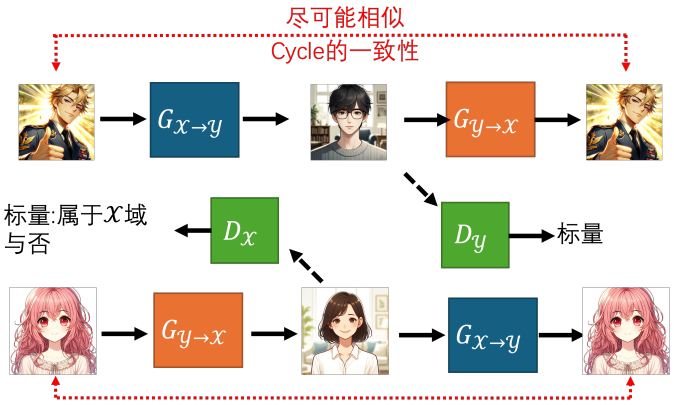


图 8.32　Cycle GAN 的双向架构

另一个角度，Cycle GAN 可以是双向的，如图 8.32 所示。我们刚才有一个生成器，输入 y 域的图片，输出 x 域的图片，是先把 x 域的图片转成 y，在把 y 转回 x。在训练 cycle GAN 的时候，其实可以同时做另外一个方向的训练，也就是把橙色的生成器给它 y 域的图片，让它产生 x 域的图片。然后在让蓝色的生成器把 x 域的图片还原回原来 y 域的图片。同时我们依然希望输入跟输出越接近越好，所以一样要训练一个判别器，这个判别器是 x 域的判别器，它是要看橙色生成器输出的图片像不像是真实人脸的图片。这个橙色的生成器它要去骗过这个 D_x 判别器。这个合起来就是 Cycle GAN。

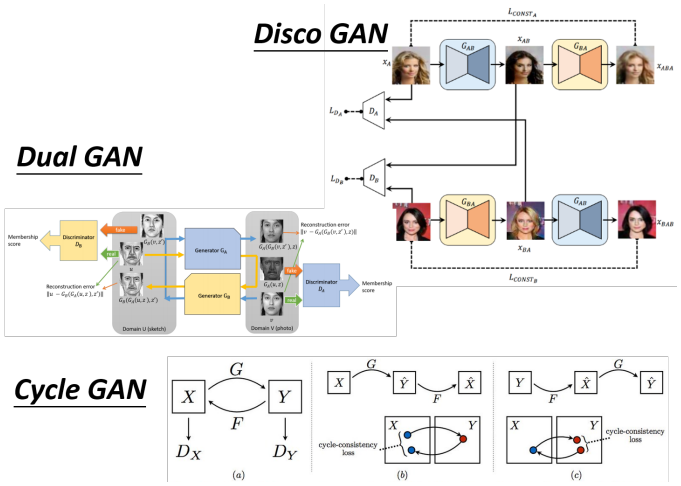


图 8.33　其他可以做风格转换的 GAN

除了 Cycle GAN 以外，还有很多其他的可以做风格转换的 GAN，比如 Disco GAN、Dual GAN 等等，如图 8.33 所示。这些 GAN 的架构都是类似的，一样的想法。此外还有另外一个更进阶的可以做图像风格转换的版本，叫做 StarGAN。Cycle GAN 只能在两种风格间做转换，那 StarGAN 厉害的地方是它可以在多种风格间做转换。这些文章就等待大家去自己研究。

同样的 GAN 的应用不仅限于图像风格的转换，你也可以做文字风格的转换。比如说，把一句负面的句子转成正面的句子，这个也是可以用 GAN 来做的，只是输入变成文字，输出也变成文字而已。也就等于是输入一个序列，输出一个序列，这里可能会使用 Transformer 架构来做这个文字风格转换的问题。那具体怎么做文字的风格转换呢？其实和 Cycle GAN 是一模

一样的。首先要有训练数据，收集一些负面的句子和一些正面的句子，可以从网络上直接爬虫得到。接下来就完全套用 Cycle GAN 的方法，假设我们是要负面的句子转正面的句子，那么判别器就要看现在生成器的输出像不像是真正的正面的句子。然后我们还要有另外一个生成器要学会把正面的句子转回原来负面的句子，要用 Cycle 的一致性。负面的句子转成正面的以后还可以转回原来负面的句子。两个句子的相似度也可以编码为向量来计算。

其实像这种文字风格转换还有很多其他的应用，不是只有正面句子转负面句子。举例来说，有很多长的文章想让机器学习文字风格的转换，让机器学会把长的文章变成简短的摘要。让它学会怎么精简的写作，让它学会把长的文章变成短的句子。另外，同样的想法可以做无监督的翻译，例如收集一堆英文的句子，同时收集一堆中文的句子，没有任何成对的数据，用 Cycle GAN 硬做，机器就可以学会把中文翻成英文了。另外还有，无监督式的语音识别，也就是让机器听一些声音，然后学会把声音转成文字。这个也是可以用 Cycle GAN 来做的，只是输入变成声音，输出变成文字而已。当然还有很多很多的有趣的应用等待大家去探索。

这一节有关 GAN 的内容就介绍完了，主要为大家介绍了生成式模型、GAN、GAN 的理论、GAN 的训练小技巧、评估 GAN 的方法、条件型 GAN 以及 Cycle GAN 这种不需要数据对的 GAN 网络。如果大家想继续深入研究 GAN 的理论和应用，可以多去看一些综述类型的文章以及最新的论文。

第 9 章 扩散模型

扩散模型 (diffusion model) 是一种运用了物理热力学扩散思想的生成模型。扩散模型有很多不同的变形，本章主要介绍最知名的去噪扩散概率模型（Denoising Diffusion Probabilistic Model，DDPM）。如今比较成功的用扩散模型做的图像生成的系统，比如 DALL-E、谷歌的 Imagen、Stable Diffusion 基本上都是用类似的方法来作为其扩散模型。

接下来介绍下扩散模型运作方法。我们来看它是怎么生成一张图片的。如图 9.1 所示，在生成图片的第一步，要去采样一个都是噪声的图片，就是从高斯分布里面采样出一个向量。这个向量里面有的数字，这个向量的维度跟要生成的图片大小是一模一样的。假设要生成一张 256×256 的图片，从正态分布采样出来的向量的维度就要是 256×256，就把采样到的 256×256 的向量排成一张图片。接下来就有一个去噪的模块，去噪的网络。输入一张都是噪声的图，输出它就会把噪声滤掉一点，就可能看到有一个猫的形状，再做去噪，猫的形状就逐渐出来。去噪越做越多，期待最终就看到一张清晰的图片，去噪的次数是事先定好的。通常会给每一个去噪的步骤一个编号，产生最终图片的那个编号比较小。一开始从完全都是噪声的输入开始，做去噪的编号比较大，所以这边就从 1999 一直排到 2，排到 1，这个从噪声到图片的步骤称为逆过程（reverse process）。在概念上，其实就像是米开朗基罗说的：“塑像就在石头里，我只是把不需要的部分去掉”，扩散模型做的事情是相同的。

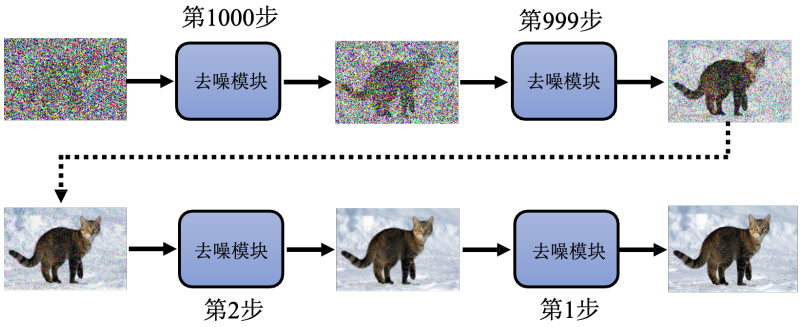


图 9.1 扩散模型生成图片的过程

接下来呢就要讲这个去噪的模型。这边是把同一个去噪的模型反复进行使用，但是因为在这边每一个状况，输入的图片差异非常大，在这个状况输入的东西就是一个纯噪声。在这个状况输入的东西噪声非常小，它已经非常接近完整的图。所以如果是同一个模型，它可能不一定能够真的做得很好。所以这个去噪的模型，除了“吃”要被去噪的那张图片以外，还会多“吃”一个输入，该输入代表现在噪声严重的程度，如图 9.2 所示。1000 代表刚开始去噪的时候，这个时候噪声的严重程度很大。1 代表说现在去噪的步骤快结束了，这个是最后一步去噪的步骤。显然噪声很小，这个去噪的模型希望它可以根据我们现在输入在第几个步骤的信息做出不同的回应。这个是去噪的模型，所以确实只有用一个去噪的模型。但是这个去噪的模型会“吃”一个额外的数字，告诉它说现在是在去噪的哪一个步骤。

去噪的模型里面实际内部做的事情是什么呢？如图 9.3 所示，在去噪的模组里面有一个噪声预测器（noise predictor），其会预测图片里面的噪声。这个噪声预测器就“吃”这个要被去噪的图片，跟“吃”一个噪声现在严重的程度，也就是我们现在进行到去噪的第几个步骤的代号，就输出一张噪声的图。它就是预测说在这张图片里面噪声应该长什么样子，再把它输出的

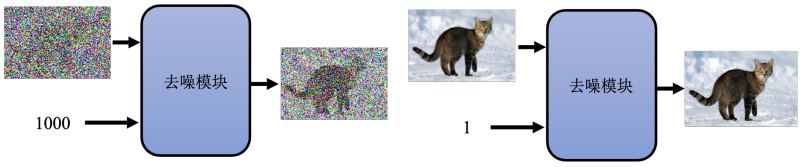


图 9.2　扩散模型进行去噪时需要结合具体步数

噪声去剪掉这个要被去噪的图片，就产生去噪以后的结果，所以这边去噪的模型并不是输入一张有噪声的图片。输出就直接是去噪后的图片，它其实是产生一个这个输入的图片的噪声，再把噪声扣掉输入的图片来达到去噪的效果。

Q：为什么不直接使用一个端到端的模型，输入是要被去噪的图片，输出就直接是去噪的结果呢？
A：可以这么做。不过如今多数的论文还是选择使用一个噪声的预测器，因为产生一张图片跟产生噪声的难度是不一样的。如果去噪的模型可以产生一只带噪声的猫，它几乎就可以说它已经会画一只猫了。因此要产生一个带噪声的猫跟产生一张图片里面的噪声的难度是不一样的，所以直接使用一个噪声预测器可能是比较简单的，使用一个端到端的模型要直接产生去噪的结果是比较困难的。

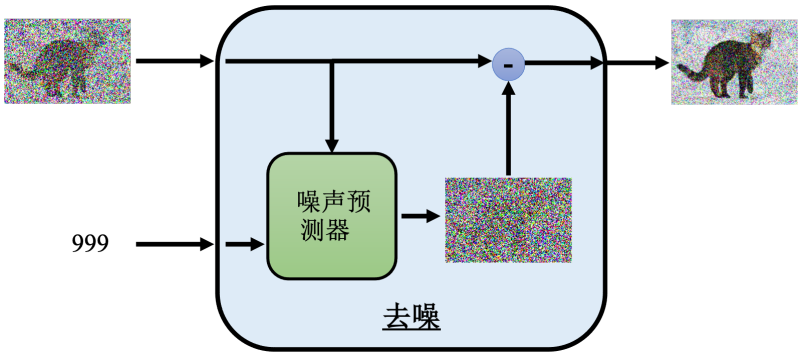


图 9.3　扩散模型去噪模块内部结构

接下来的问题就是怎么训练这个噪声预测器呢？一个去噪的模型是根据一个噪声的图片和去噪的步数的 ID，产生去噪的结果。去噪的模型里面是一个噪声预测器，它是要“吃”这张图片“吃”一个 ID，产生一个预测出来的噪声。但是要产生出一个预测出来的噪声，需要有标准答案。在训练网络的时候，要有成对的数据才能够训练。需要告诉噪声预测器这张图片里面的噪声长什么样子，它才能够学习怎么把噪声输出来。如何制造出这样子的数据呢。噪声预测器的训练数据是人为创造的，怎么创造呢？创造方法如图 9.4 所示，从数据集里面拿一张图片出来，随机从高斯分布里面采样一组噪声出来加上去，产生有点噪声的图像，能再采样一次再得到更噪声的图片，以此类推，最后整张图片就看不出来原来是什么东西。加噪音的过程称为前向过程，也称为扩散过程。做完这个扩散过程以后，就有噪声预测器的训练数据了。对噪声预测器，其训练数据就是这一张加完噪声的图片跟现在是第几次加噪声，是网络的输入，而加入的这个噪声就是网络应该要预测的输出，就是网络输出的标准答案。

所以在做完这个扩散过程以后，就有训练数据了。噪声预测器看到这张图，看到第二个

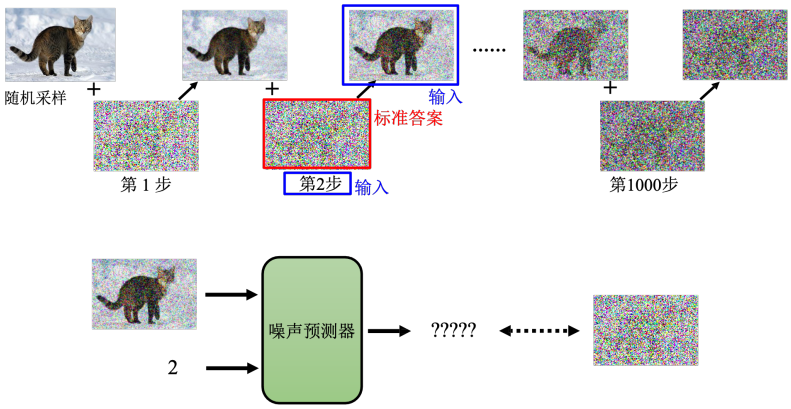


图 9.4　扩散模型的前向过程

步骤，输入 2 这个数字，要输出是什么，标准答案就是一个长这个样子的噪声。接下来就跟训练一般的网络一样训练下去就结束了。但是我们要的不只是生图而已。刚才只是从一个噪声里面生出图，还没有考虑文字。如果要训练一个图像生成的模型，他是“吃”文字产生图片，如图 9.5 所示。其实还是需要图片跟文字成对的数据。ImageNet 中每一张图片有一个类别的标记，还不是那个图片的描述，不是像“一只在猫在雪里”这样图片的描述，它只是每一张图有一个标记，它有 100 万张图片。Midjourney、Stable Diffusion 或 DALL-E 的数据往往来自于 LAION，LAION 有 58.5 亿张的图片，所以难怪今天这一些模型可以产生这么好的结果，LAION 有一个搜索的 Demo 的平台，里面内容很全面，比如猫的图片，它不是只有猫的图片跟英文文字的对应，还有跟中文、英文的对应。所以之所以图像生成模型不是只看得懂英文、中文它大多数都看得懂，是因为它的训练数据里面也有中文跟其他的语言，里面有很多名人的照片，随便一找就一大堆。因此 Midjourney 能画得出很多名人，是因为他就是知道名人的模样。所以这个是需要准备的训练数据。有了这个文字跟图像成对的数据以后。

图 9.5　文生图示例

如图 9.6 所示，把文字加到去噪的模组就结束了。所以去噪的模组不是只看输入的图片做去噪，它是根据输入的图片加上一段文字的描述去把噪声拿掉，所以在每一个步骤去噪的模组都会有一个额外的输入。这个额外的输入就是，要它根据这段文字的描述产生什么样的图片。这个去噪模块里面的噪声预测器要怎么改呢？直接把这段文字给噪声预测器就结束了，要让噪声预测器多一个额外的输入也就是这段文字，如图 9.7 所示，就结束了。

训练的部分要怎么改呢，如图 9.8 所示，每一张图片都有一段文字，所以先把这张图片做完扩散过程以后，在训练的时候，不只要给噪声预测器加入噪声后的图片，还有现在步骤的 ID，多给一个就是文字的输入。噪声预测器会根据这 3 样东西产生适当的噪声，产生要去消掉的噪声，产生这个标准答案。

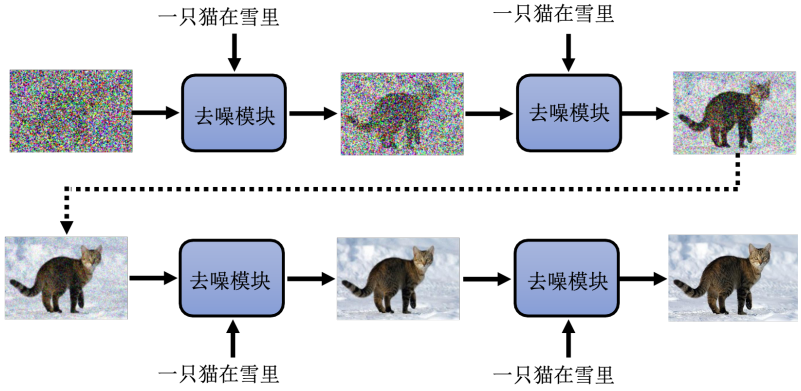


图 9.6　文生图的去噪过程

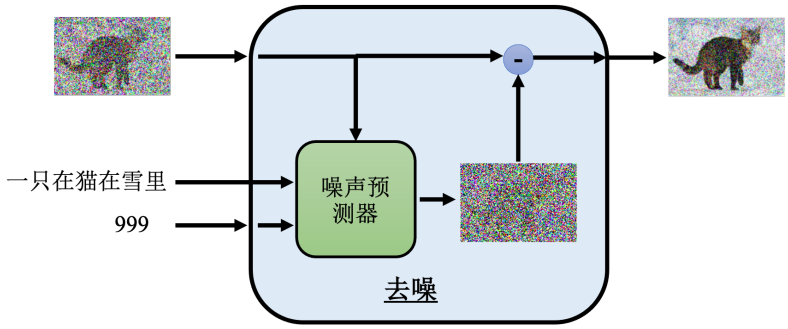


图 9.7　去噪模块加文字描述

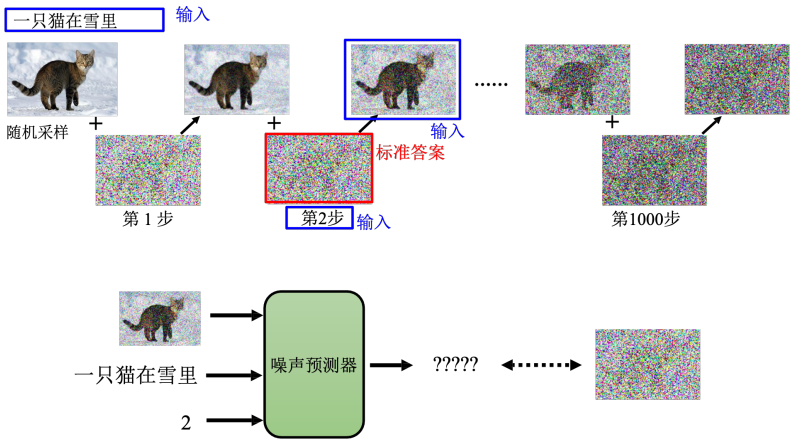
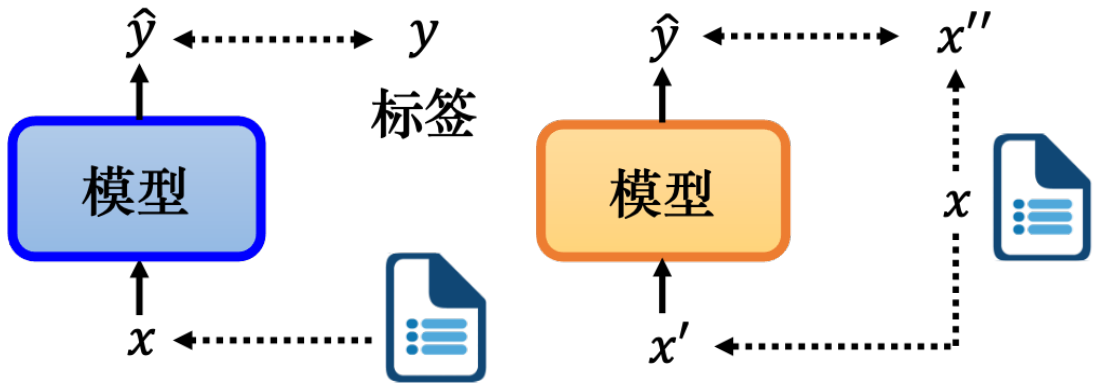


图 9.8　加入文字描述的前向过程

第 10 章 自监督学习

自监督学习 (Self-Supervised Learning, SSL) 是一种无标注的学习方式，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。监督学习与无监督学习是两种常见的学习方式，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 10.1(a) 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，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无标注的学习方式. 如图 10.1(b) 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.



(a) 监督学习 (b) 自监督学习

图 10.1 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**知识增强的语义表示模型 (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 10.1 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 10.1　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.

10.1　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 10.2 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、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.

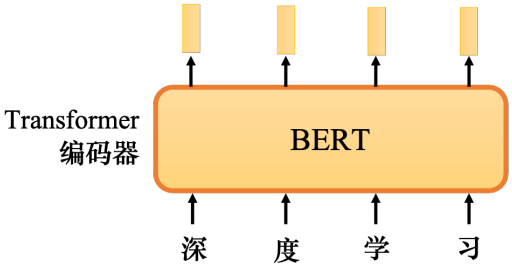


图 10.2　BERT 的架构

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. 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 10.3 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.

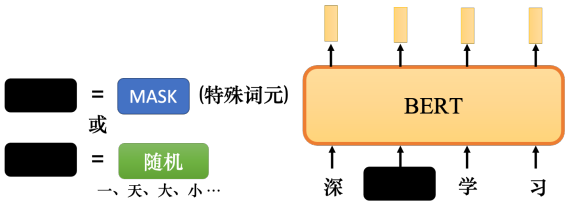


图 10.3　掩码的两种方法

如图 10.4 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.

如何训练 BERT 模型？如图 10.5 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. 独热编码可以用来表示字符，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.

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 10.6 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

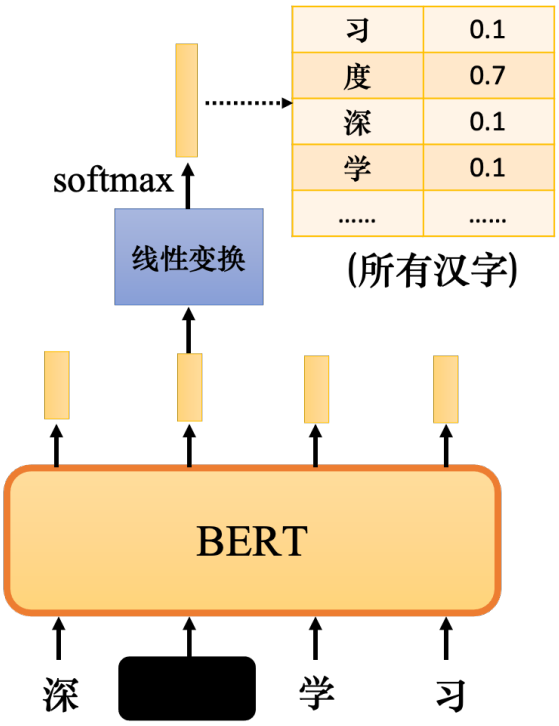


图 10.4 BERT 预测过程

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.

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.

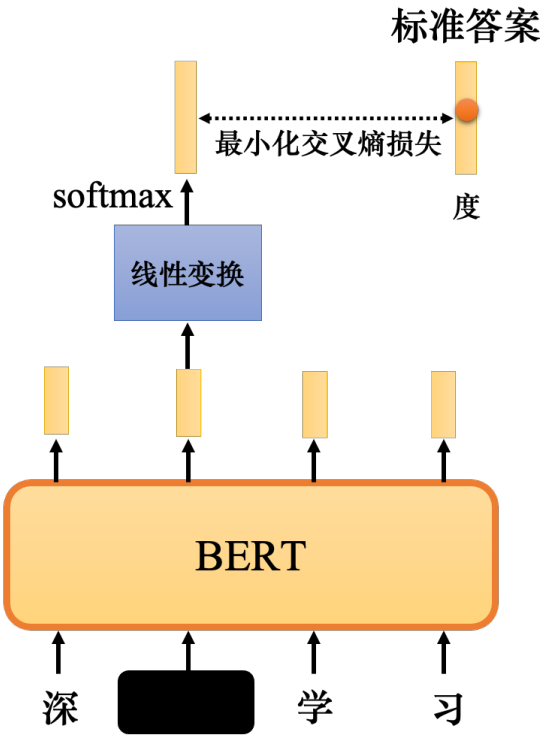


图 10.5 BERT 的训练过程

10.1.1 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.

通过这两个任务，BERT 学会了如何填空。在训练模型填空后，也可以用于其他任务。如图 10.7 所示，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，可称为**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.

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-tuning）**. 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. 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

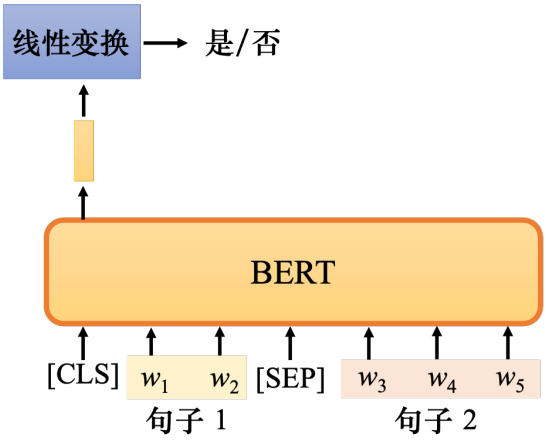


图 10.6　下一句预测

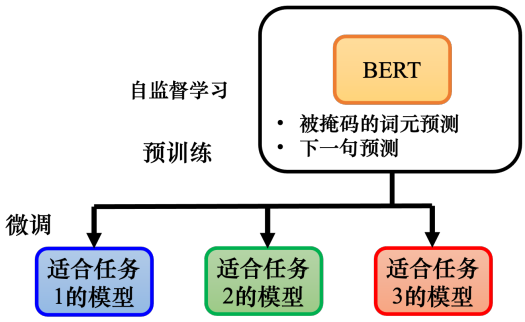


图 10.7　使用 BERT 解决下游任务

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 10.8 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 10.8 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. 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，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，但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. 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.

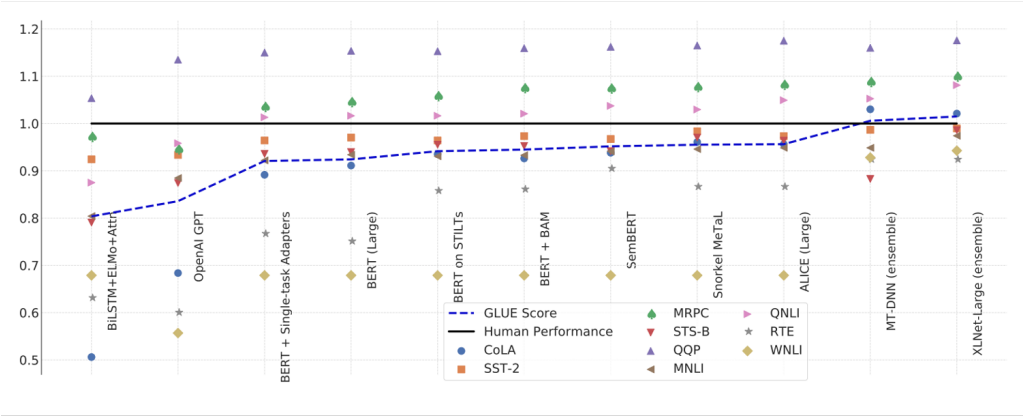


图 10.8　BERT 的训练过程

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 10.9 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。

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.

但在 BERT 中不必随机初始化所有参数，随机初始化的参数只是线性变换的参数. 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 10.10 所示，横轴是训练的回合，纵轴是训练损失. 随着训练的进行，损失会越来越

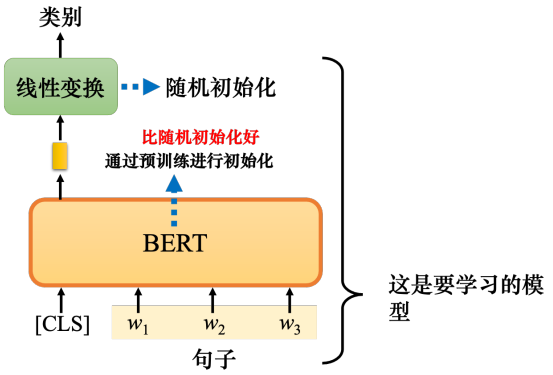


图 10.9　用 BERT 做情感分析

低. 图 10.10 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.

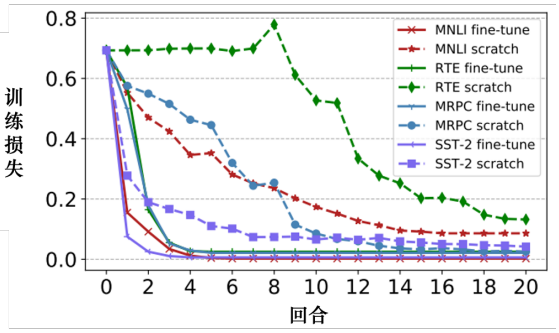


图 10.10　预训练模型的初始化结果对比

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 10.11 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.

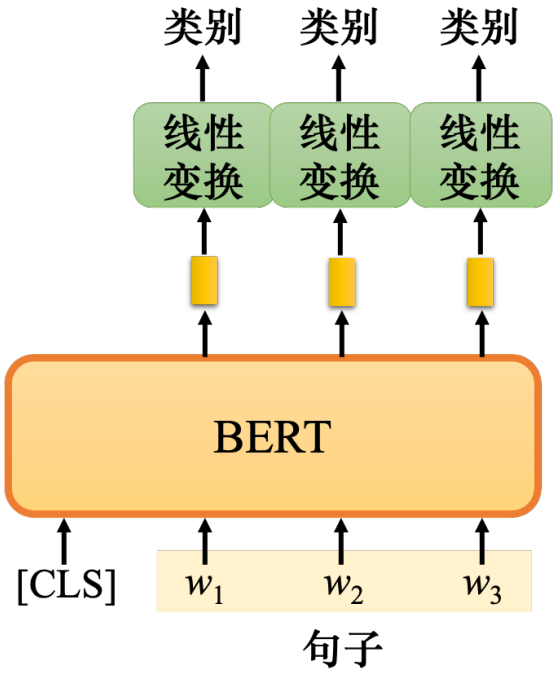


图 10.11 用 BERT 做词性标注

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）. 机器所做的是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 10.12 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.

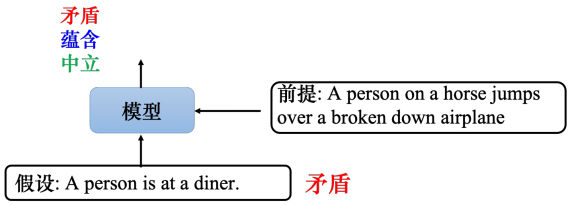


图 10.12 自然语言推理

BERT 如何解决这个问题? 如图 10.13 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.

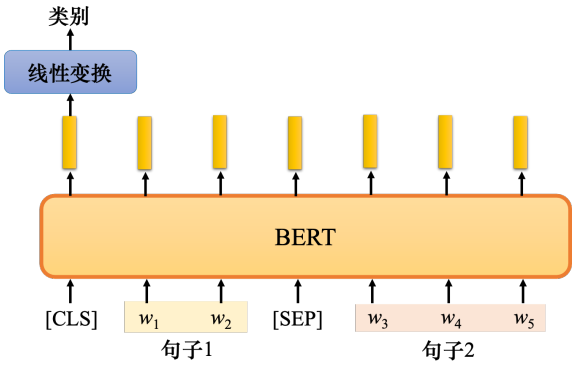


图 10.13 使用 BERT 进行自然语言推理

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：

$$\begin{aligned} D &= \{d_1, d_2, \cdots, d_N\} \\ Q &= \{q_1, q_2, \cdots, q_M\} \end{aligned} \tag{10.1}$$

对于中文，式 (10.1) 中每个 d 代表一个汉字，每个 q 代表一个汉字.

如图 10.14 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 10.14　问答模型

例如，如图 10.15 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.

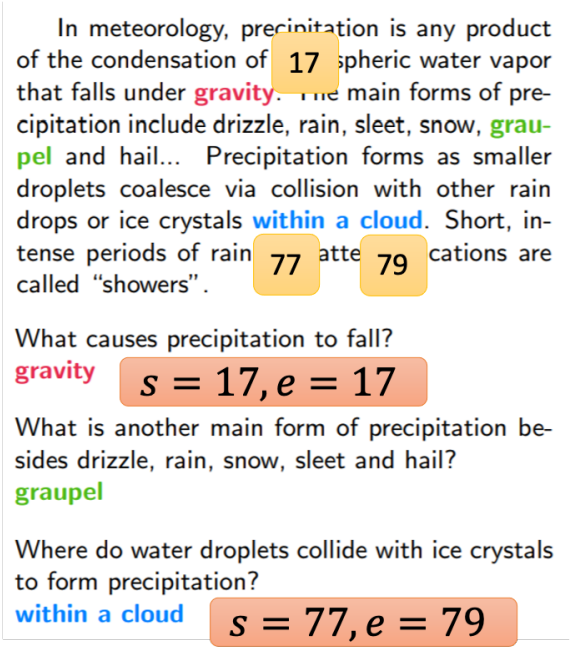


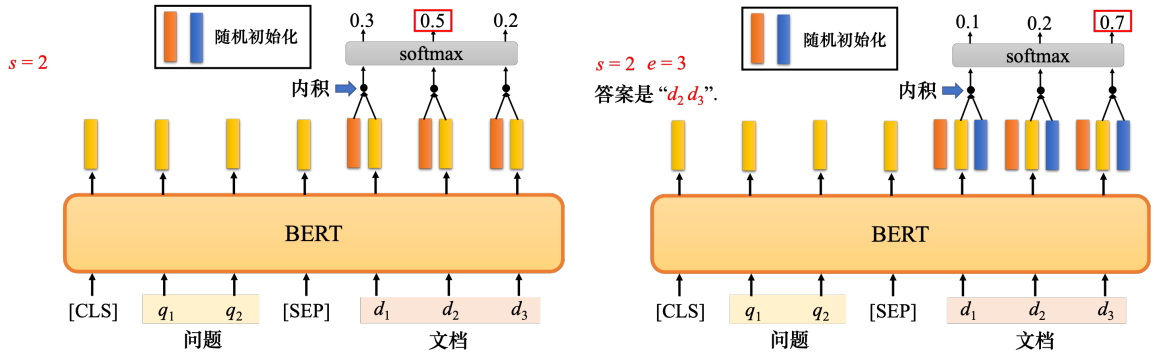
图 10.15　基于提取的问答

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 10.16 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 10.16(a) 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.

如图 10.16(b) 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.



(a) 找起始位置　　(b) 找结束位置

图 10.16　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 10.17 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.

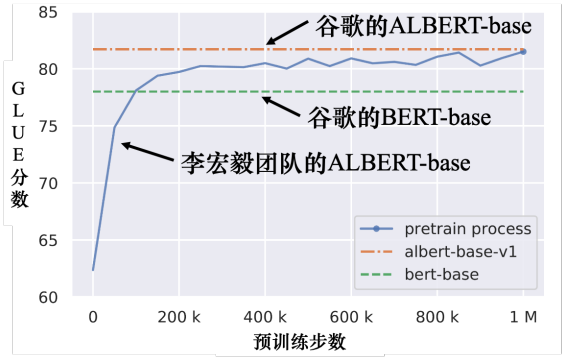


图 10.17　使用 ALBERT 训练 GLUE

30 亿数据看起来很多，但它是未标注的数据. 从网络上随便爬取一堆文字就可以有这么多数据，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.

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 10.18 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.

损坏句子的方式有多种，如图 10.19 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.

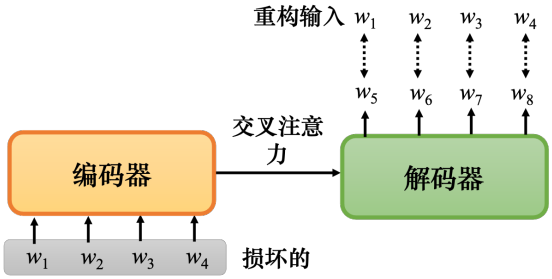


图 10.18　预训练一个 Seq2Seq 模型

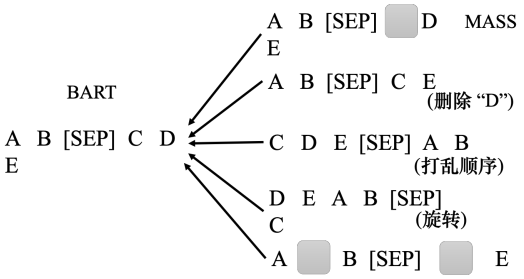


图 10.19　损坏句子的方法

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，即使下载完成，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.

10.1.2　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 10.20 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 10.21 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“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”.

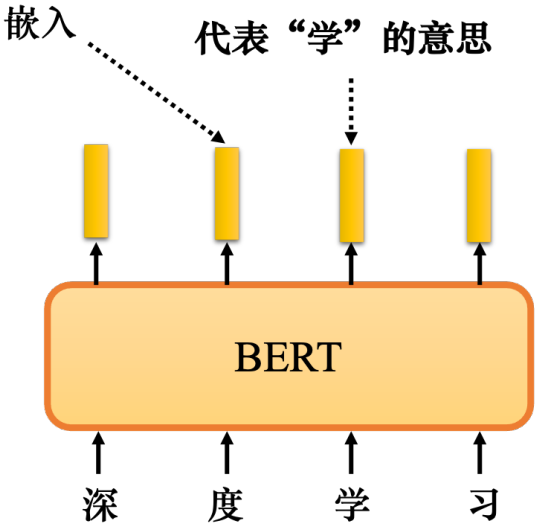


图 10.20　BERT 输出的嵌入代表了输入的字的意思

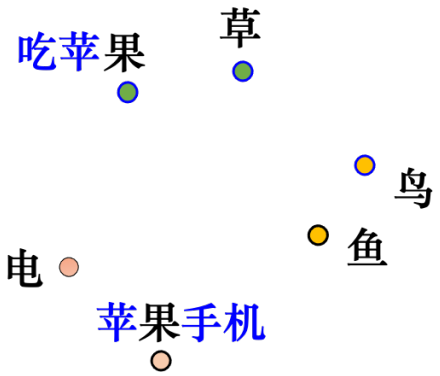


图 10.21　意思相近的字，嵌入更接近

如图 10.22 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 10.23 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 10×10 的矩阵.

图 10.23 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

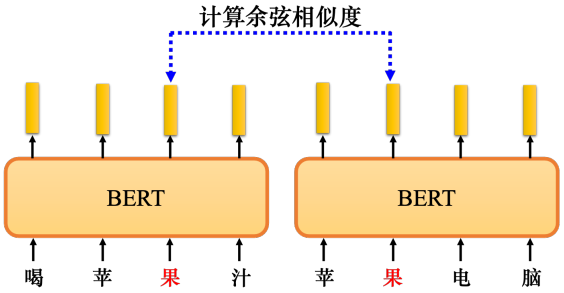


图 10.22 计算余弦相似度

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 10.24 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 10.25 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，词嵌入中有一种技术称为**连续词袋（Continuous Bag Of Words，CBOW）**. 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.

> Q：为什么 CBOW 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?
> A：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 10.26 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.

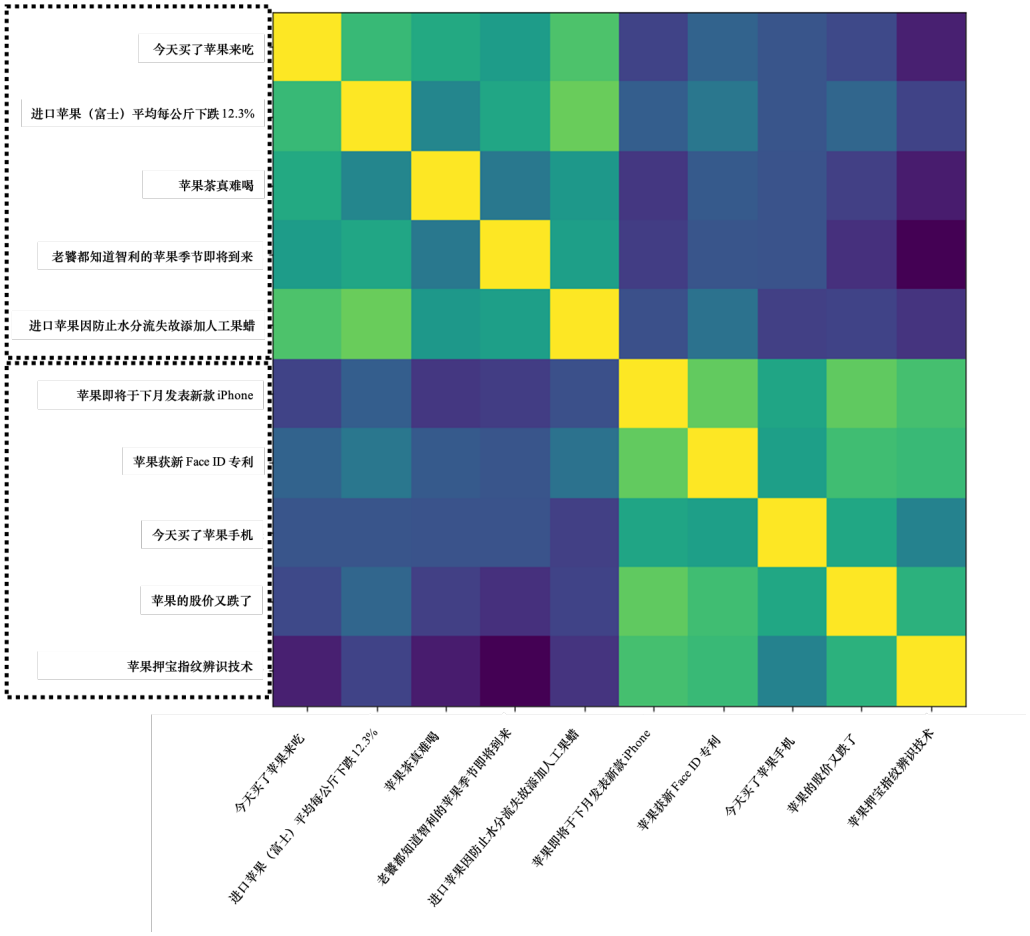


图 10.23　余弦相似度的计算结果

如图 10.27 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？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，使用 BERT 的结果实际上更好[1].

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

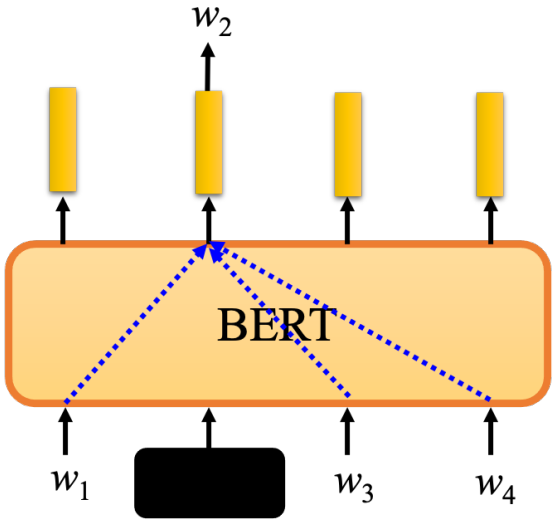


图 10.24 通过上下文信息预测掩码部分

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.

10.1.3 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 10.29 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. 表 10.2 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.

如图 10.30 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. 也许多语言 BERT 在看过大量语言的过程中自动学会了这件事情.

如图 10.31 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

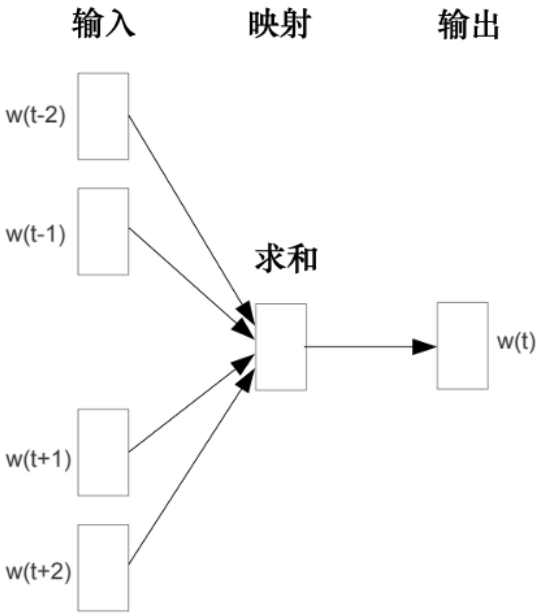


图 10.25　连续词袋模型

类别	DNA 序列
EI	CCAGCTGCATCACAGGAGGCCAGCGAGCAGGTCTGTTCCAAGGGCCTTCGAGCCAGTCTG
EI	AGACCCGCCGGGAGGCGGAGGACCTGCAGGGTGAGCCCCACCGCCCCTCCGTGCCCCCGC
IE	AACGTGGCCTCCTTGTGCCCTTCCCCACAGTGCCCTCTTCCAGGACAAACTTGGAGAAGT
IE	CCACTCAGCCAGGCCCTTCTTCTCCTCCAGGTCCCCCACGGCCCTTCAGGATGAAAGCTG
IE	CCTGATCTGGGTCTCCCCTCCCACCCTCAGGGAGCCAGGCTCGGCATTTCTGGCAGCAAG
IE	AGCCCTCAACCCTTCTGTCTCACCCTCCAGCCTAAAGCTCCTTGACAACTGGGACAGCGT
IE	CCACTCAGCCAGGCCCTTCTTCTCCTCCAGGTCCCCCACGGCCCTTCAGGATGAAAGCTG
N	CTGTGTTCACCACATCAAGCGCCGGGACATCGTGCTCAAGTGGGAGCTGGGGGAGGGCGC
N	GTGTTACCGAGGGCATTTCTAACAGTCTTCTTACTACGGCCTCCGCCGACCGCGCGCTCG
N	TCTGAGCTCTGCATTTGTCTATTCTCCAGCTGACCCTGGTTCTCTCTCTTAGCTACCTGC

图 10.26　DAN 分类问题

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 10.31 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关键因素，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，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。

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

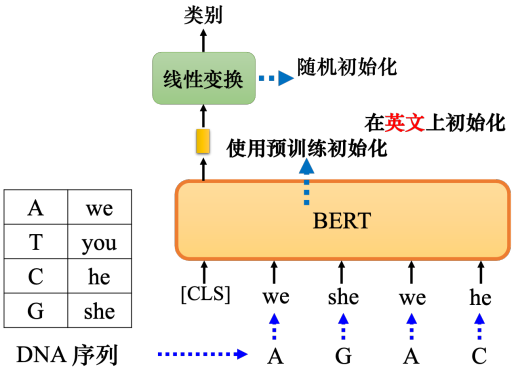


图 10.27　使用 BERT 进行 DNA 分类

	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 10.28　使用 BERT 处理不同任务

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 10.32 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.

多语种 BERT 可以做一个很棒的无监督翻译，如图 10.33 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 10.33 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.

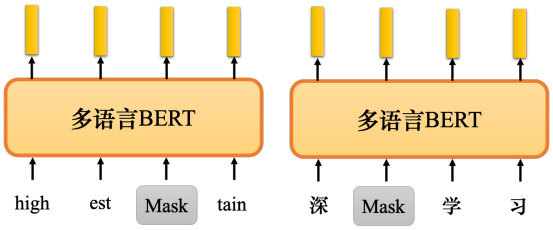


图 10.29　多语言 BERT

表 10.2 使用多语言 BERT 进行问答[2]

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

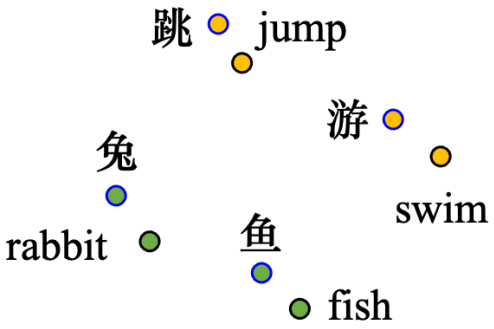


图 10.30 多语言 BERT 对比

10.2 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 10.34 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 <BOS>（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 <BOS> 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

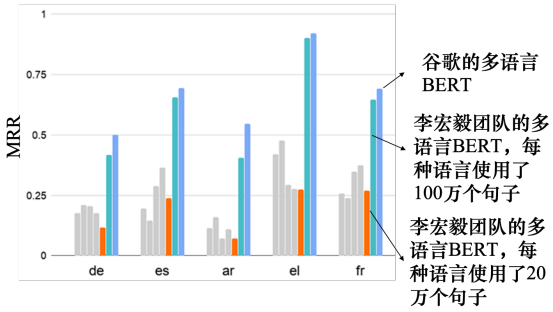


图 10.31 多语言 BERT 对比[3]

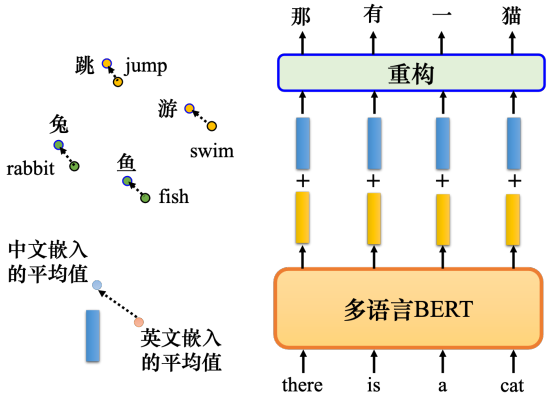


图 10.32　中英文之间的差距

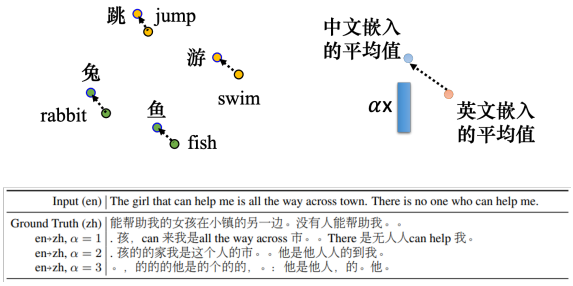


图 10.33　无监督词元级翻译

(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 <BOS> 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 <BOS>、“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 <BOS>、“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”.

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 用了很多数据训练了一个非常大的模型. 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 <BOS> 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假新闻，假新闻里面说在安第斯山脉发现了独角兽，

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 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. 而 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. 因此 GPT

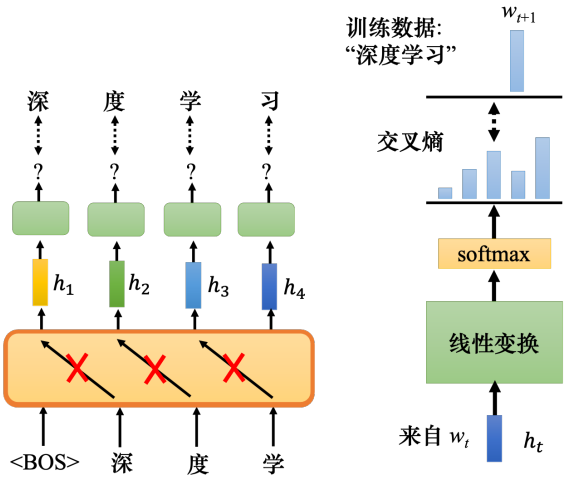


图 10.34　使用 GPT 预测下一个词

系列有一个和人类更接近的使用方式。

如图 10.35 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）**.

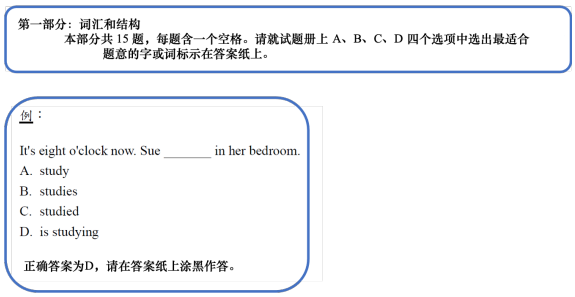


图 10.35　托福听力测验

假设要 GPT 做翻译，如图 10.36(a) 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 10.36(b) 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 10.36(c)，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来做翻译. 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.

(a) 小样本学习 (b) 单样本学习 (c) 零样本学习

图 10.36 语境学习

如图 10.37 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% 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.

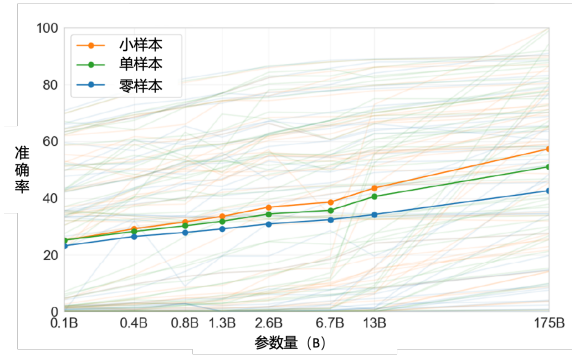


图 10.37 使用 GPT-3 进行语境学习

如图 10.38 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.

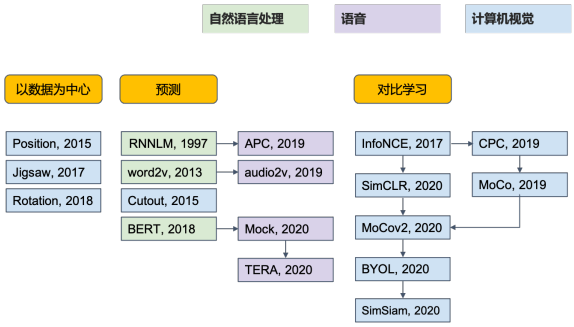


图 10.38　其他领域的自监督学习

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 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。

语音有非常多不同的方向，语音相关的技术不只是把语音识别把声音转成文字. 语音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.

参考文献

[1] KAO W T, LEE H Y. Is bert a cross-disciplinary knowledge learner? a surprising finding of pre-trained models' transferability[J]. arXiv preprint arXiv:2103.07162, 2021.

[2] HSU T Y, LIU C L, LEE H Y. Zero-shot reading comprehension by cross-lingual transfer learning with multi-lingual language representation model[J]. arXiv preprint arXiv:1909.09587, 2019.

[3] LIU C L, HSU T Y, CHUANG Y S, et al. What makes multilingual bert multilingual? [J]. arXiv preprint arXiv:2010.10938, 2020.

第 11 章 自编码器

在讲**自编码器（autoencoder）**之前，其实自编码器也可以算是自监督学习的一环，因此我们可以再简单回顾一下自监督学习的框架。如图 11.1 所示，首先你有大量的没有标注的数据，用这些没有标注的数据，你可以去训练一个模型，你必须设计一些不需要标注数据的任务，比如说做填空题或者预测下一个词元等等，这个过程就是自监督学习，有时也叫做预训练。用这些不用标注数据的任务学完一个模型以后，它可能本身没有什么作用，比如 BERT 模型只能做填空题，GPT 模型只能够把一句话补完，但是你可以把它用在其他下游的任务里面。

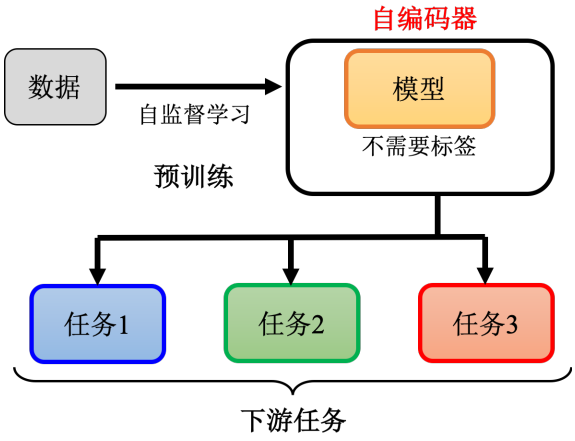


图 11.1 自监督学习框架

在有 BERT 或者 GPT 模型之前，其实有一个更古老的，不需要用标注数据的任务，就叫做自编码器，所以你也可以把自编码器看作是一种自监督学习的预训练方法。当然可能不是所有人都会同意这个观点，有人可能会说这个自编码器，不算是自监督学习。因为这个自编码器是早在 2006 年就有的概念，然后自监督学习是 2019 年才有这个词汇，所以他们认为自编码器不算是自监督学习的一环。这个都是见仁见智的问题，这种名词定义的问题，我们就不用太纠结在这个地方，从自监督学习，即不需要用标注数据来训练这个角度来看，自编码器我们可以认为它算是自监督学习中的一种方法，它就跟填空或者预测接下来的词元是很类似的概念，只是用的是另外一种不一样的思路。

11.1 自编码器的概念

自编码器的原理，以图像为例，如图 11.2 所示，假设我们有非常大量的图片，在自编码器里面有两个网络，一个叫做编码器，另外一个叫做解码器，它们是不同的两个网络。编码器把一张图片读进来，它把这张图片变成一个向量，编码器可能是很多层的卷积神经网络（CNN），把一张图片读进来，它的输出是一个向量，接下来这个向量会变成解码器的输入。而解码器会产生一张图片，所以解码器的网络架构可能会像是 GAN 里面的生成器，它是比如 11 个向量输出一张图片。

训练的目标是希望编码器的输入跟解码器的输出越接近越好。换句话说，假设你把图片看作是一个很长的向量的话，我们就希望这个向量跟解码的输出，这个向量，这两个向量他

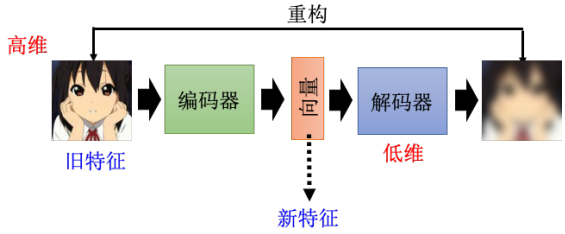


图 11.2 自编码器的流程

们的距离越接近越好，也有人把这件事情叫做**重构（reconstruction）**。因为我们就是把一张图片，压缩成一个向量，接下来解码器要根据这个向量，重建出原来的图片，希望原输入的结果跟重建后的结果越接近越好。讲到这里读者可能会发现说，这个概念其实跟前面讲的 Cycle GAN 模型是类似的。

在做 Cycle GAN 的时候，我们会需要两个生成器，第一个生成器把 X 域的图片转到 Y 域，另外一个生成器把 Y 域的图片转回来，然后希望最原先的图片跟转完两次后的图片越接近越好。那这边编码器和解码器，也就是这个自编码器的概念，跟 Cycle GAN 其实是一模一样的，都是希望所有的图片经过两次转换以后，要跟原来的输出越接近越好，而这个训练的过程，完全不需要任何的标注数据，你只需要收集到大量的图片，你就可以做这个训练。因此它是一个无监督学习的方法，跟自监督学习系列中预训练的做法一样，你完全不需要任何的标注数据。那像这样子这个编码器的输出，有时候我们叫它嵌入。嵌入也称为表示或编码，因为编码器是一个编码，所以这个有人把这个向量叫做编码，这些其实指的都是同一件事情。

怎么把训练好的自编码器用在下游的任务里面呢？常见的用法就是把原来的图片可以看成是一个很长的向量，但这个向量太长了不好处理，这是把这个图片丢到编码器以后，输出另外一个向量，这个向量我们会让它比较短，比如说只有 10 维或者 100 维。接着拿这个新的向量来做接下来的任务，也就是图片不再是一个很高维度的向量，它通过编码器的压缩以后，变成了一个低维度的向量，我们再拿这个低维度的向量，来做接下来想做的事情，这就是自编码器用在下游任务的常见做法。

由于通常编码器的输入是一个维度非常高的向量，而其输出也就是我们的嵌入（也称为表示或编码），其是一个非常低维度的向量。比如输入是 100×100 的图片，100×100 那就是 1 万维的向量。如果是 RGB 那就是 3 万维的向量，但是通常编码器我们会设得很小，比如说 10、100 这样的量级，所以这个这边会有一个特别窄的部分，本来输入是很宽的，输出也是很宽的，但是中间特别窄，因此这一段就叫做瓶颈。而编码器做的事情，是把本来很高维度的东西，转成低维度的东西，把高维度的东西转成低维度的东西又叫做降维。

11.2 为什么需要自编码器?

自编码器到底好在哪里？当我们把一个高维度的图片，变成一个低维度的向量的时候，到底带来什么样的帮助呢？我们来设想一下，自编码器这件事情它要做的，是把一张图片压缩又还原回来，但是还原这件事情为什么能成功呢？如图 11.3 所示，假设本来图片是 3×3 的维度，此时我们要用 9 个数值来描述一张 3×3 的图片，假设编码器输出的这个向量是二维的，我们怎么才可能从二维的向量去还原 3×3 的图片，即还原这 9 个数值呢？怎么有办法把 9 个数值变成 2 个数值，又还原成 3，又还原回 9 个数值呢？

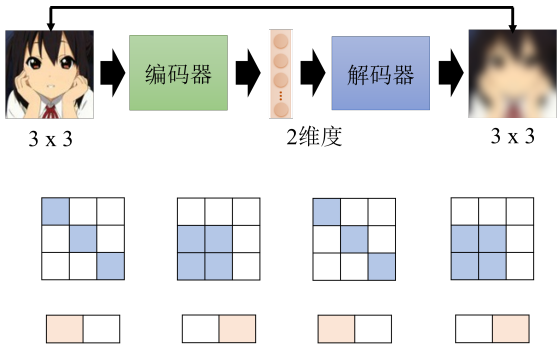


图 11.3　自编码器的原理

能够做到这件事情是因为，对于图像来说，并不是所有 3×3 的矩阵都是图片，图片的变化其实是有限的，你随便采样一个随机的噪声，随便采样一个矩阵出来，它通常都不是你会看到的图片。举例来说，假设图片是 3×3 的，那它的变化，虽然表面上应该要有 3×3 个数值，才能够描述 3×3 的图片，但是也许它的变化实际上是有限的。也许我们把图片收集起来发现，它实际上只有如图 11.3 所示的白色和橙色方块两种类型。一般在训练的时候就会看到这种状况，就是因为图片的变化还是有限的。因此我们在做编码器的时候，有时只用两个维度就可以描述一张图片，虽然图片是 3×3，应该用 9 个数值才能够储存，但是实际上它的变化也许只有两种类型，那你就可以说看到这种类型，我就左边这个维度是 1 ，右边是 0，看到这种类型就左边这个维度是 0，右边这个维度是 1。

而编码器做的事情就是化繁为简，有时本来比较复杂的东西，它实际上只是表面上看起来复杂，而本身的变化是有限的。我们只需要找出其中有限的变化，就可以将它本来比较复杂的东西用更简单的方法来表示。如果我们可以把复杂的图片，用比较简单的方法来表示它，那我们就只需要比较少的训练数据，在下游的任务里面，我们可能就只需要比较少的训练数据，就可以让机器学到，这就是自编码器的概念。

自编码器从来就不是一个新的概念，它具有悠长的历史。举例来说，深度学习之父 Hinton 早在 2006 年的 Science 论文里面就有提到自编码器这个概念，只是那个时候用的网络跟今天我们用的有很多不一样的地方。那个时候自编码器的结构还不太成熟，当时人们还并不认为深度的神经网络是能够训练起来的，换句话说就是把网络叠很多层，然后每一层一起训练的方法是不太可能成功的，因此人们普遍认为每一层应该分开训练，所以 Hinton 用的是一个叫做**受限玻尔兹曼机（Restricted Boltzmann Machine，RBM）**的技术。

这里我们特别把 Hinton 教授 2006 年的文章里面的图拿出来给大家展示一下过去人们是怎么看待深度学习这个问题的。那个时候要训练一个很深的网络不太可能，每一层分开要训练，虽然这个在现在看来很深也没有很深，只是三层，但是在 2006 年这个已经是很深的了。那这个三层要分开来训练才可以，那这边说分开来训练这件事情叫做预训练。但它跟自监督学习中的预训练又不太一样，假设自编码器是预训练，那么这里的预训练就相当于是预训练中的预训练，这里预训练倾向于训练流程的概念，自编码器的预训练倾向于算法流程的概念。这里预训练是要先训练自编码器，每一层用 RBM 的技术分开来训练。先把每一层都训练好，再全部接起来做微调这件事情，这边的微调也并不是 BERT 模型的微调，它是微调那个预训练的模型。这个 RBM 技术我们会发现今天很少有人在提到它了，它其实不是一个深度学习的技术，它有点复杂，我们也没有打算要深入细讲，至于为什么现在都很少人用它，就是因为

它没有什么用。但是在 2006 年，还是有必要用到这个技术的。其实 Hinton 后来在 2012 年的时候，有一篇论文呢偷偷在结尾下一个结论说其实 RBM 技术也没有什么必要，所以后来就没有什么人再用这个技术了。而且那时候还有一个神奇的信念，即编码器和解码器，它们的结构必须是对称，所以编码器的第一层，跟解码器的最后一层，它们必须对应，不过现在已经没有或者说比较少这样的限制。总之，自编码器从来就不是一个新的概念。

11.3 去噪自编码器

自编码器有一个常见的变体，叫做**去噪自编码器（denoising autoencoder）**。如图 11.4 所示，去噪自编码器就是把原来需要输入到编码器的图片，加上一些噪声，然后一样地通过编码器，再通过解码器，试图还原原来的图片。

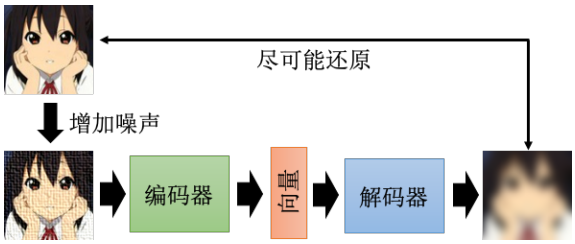


图 11.4 去噪自编码器的结构

我们现在还原的，不是编码器的输入，编码器的输入的图片是有加噪声的，我们要还原的不是加入噪声之前的结果。所以我们会发现现在编码器跟解码器，除了还原原来的图片这个任务以外，它还多了一个任务，这个任务就是它必须要自己学会把噪声去掉。编码器看到的是有加噪声的图片，但解码器要还原的目标是，没有加噪声的图片，所以编码器加上解码器，他们合起来必须要联手能够把噪声去掉，这样你才能够把去噪的自编码器训练出来。

其实去噪自编码器也不算是太新的技术，至少在 2008 年的时候，就已经有相关的论文了。如果读者看 BERT 模型的话，其实也可以把它看成一个去噪自编码器。输入我们会加掩码，掩码其实就是噪声，BERT 的模型就是编码器，它的输出就是嵌入。在讲 BERT 的技术的时候，我们说这个输出就叫做嵌入，接下来有一个线性的模型，就是解码器，解码器要做的事情，就是还原原来的句子，也就是把填空题被盖住的地方，把它还原回来，所以我们可以说，BERT 其实就是一个去噪的自编码器。有读者可能会问，为什么这个解码器一定要是线性的呢，其实它不一定要是线性模型。或者换一个说法，如图 11.5 所示，这个 BERT 它有 12 层，最小的那个 BERT 有 12 层，比较大的有 24 层或者是 48 层，那最小的 BERT 是 12 层，如果我们说这个 12 层中间，第 6 层的输出是嵌入，那其实也可以说剩下的 6 层，就是解码器。可以说 BERT，就假设在用 BERT 的时候，我们用的不是第 12 层的输出，而是第 6 层的输出，那完全可以说，BERT 的前 6 层就是编码器，后面 6 层就是解码器，总之这个解码器没有一定要是线性的。

11.4 自编码器应用之特征解耦

自编码器可应用于在**特征解耦（feature disentanglement）**。解耦是指把一堆本来纠缠在一起的东西把它解开。为什么需要解耦？我们先看一下自编码器做的事情，如图 11.6 所示，如果是图片的话，就是把一张图片变成一个编码，再把编码变回图片，既然这个编码可以变

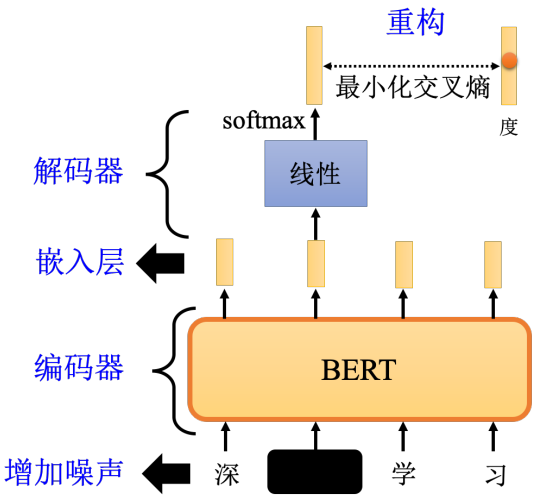


图 11.5　BERT 模型回顾

回图片，代表说这个编码里面有很多的信息，包含图片里面所有的信息。举例来说，图片里面的色泽、纹理等等。自编码器这个概念也不是只能用在图像上，如果用在语音上，可以把一段声音丢到编码器里面，变成向量再丢回解码器，变回原来的声音，代表这个向量包含了语音里面所有重要的信息，包括这句话的内容是什么，就是编码器的信息，还有这句话是谁说的，就是语者的信息。如果是一篇文章，丢到编码器里面变成向量，这个向量通过解码器会变回原来的文章，那这个向量里面有什么，它可能包含文章里面，文句的句法的信息，也包含了语义的信息，但是这些信息是全部纠缠在一个向量里面，我们并不知道一个向量的哪些维度代表了哪些信息。

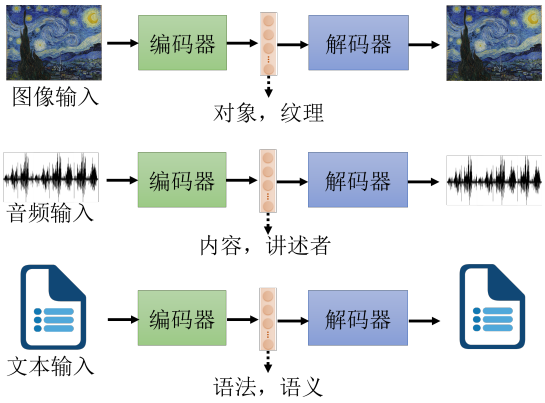


图 11.6　自编码器模型回顾

而特征解耦想要做到的事情就是，我们有没有可能想办法，在训练一个自编码器的时候，同时有办法知道嵌入（又称为表征或编码）的哪些维度代表了哪些信息。比如 100 维的向量，知道前 50 维就代表了这句话的内容，后 50 维就代表了这句话说话人的特征，这种对应的技术称为特征解耦。

再举一个特征解耦方面的应用，叫做语音转换，如图 11.7 所示。也许读者们没有听过语音转换这个词汇，但是一定看过它的应用，它就相当于是柯南的领结变身器。阿笠博士在做

这个变声期也就是语音转换的时候，需要成对的声音信号，也就是假设要把 A 的声音转成 B 的声音，就必须把 A 跟 B 都找来，叫他念一模一样的句子。

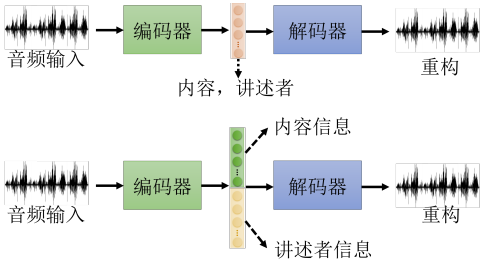


图 11.7　特征解耦应用之语音转换

如图 11.8 所示，A 说好“How are you”，B 也说好“How are you”，A 说“Good morning”，B 也说“Good morning”，他们两个各说一样的句子，说个 1000 句，接下来就交给自监督学习去训练了。即现在有成对的数据，训练一个自监督模型，把 A 的声音丢进去，输出就变成 B 的声音。但是如果 A 跟 B 都需要念一模一样的句子，念个 500 或者 1000 句，显然是不切实际的。

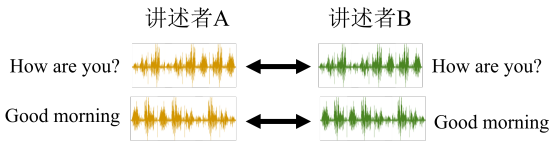


图 11.8　语音转换的挑战

有了特征解耦的技术以后，我们可以期待机器做到，给它 A 的声音和 B 的声音，A 跟 B 不需要念同样的句子，甚至不需要讲同样的语言，机器也有可能学会把 A 的声音转成 B 的声音。实际的做法如图 11.9 所示，假设收集到一大堆人类的声音信号，使用这堆声音信号训练一个自编码器，同时又做了特征解耦，所以我们就知道了在编码器的输出里面，哪些维度代表了语音的内容，哪些维度代表了讲述者的特征，这样就可以把两句话的声音跟内容的部分互换。

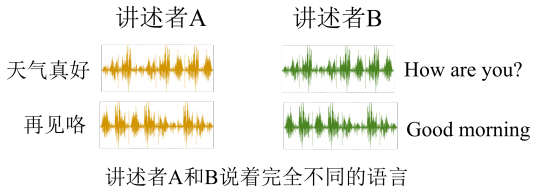


图 11.9　语音转换中特征解耦的作用

举例来说，如图 11.10 所示，讲述者 A 的声音（“How are you?”）丢进编码器以后，就可以知道这个编码器的输出里面，某些维度代表“How are you?”的内容，某些维度代表讲述者 A 的声音。把讲述者 B 的声音丢进编码器里面，它就知道某一些维度代表讲述者 B 说的话的内容，某一些维度代表讲述者 B 声音的特征。接下来只要把讲述者 A 说话的内容的部分取出来，把讲述者 B 说话的声音特征的部分取出来，把它拼起来，丢到解码器里面，就可以用讲

述者 A 的声音，讲讲述者 B 说的话的内容。

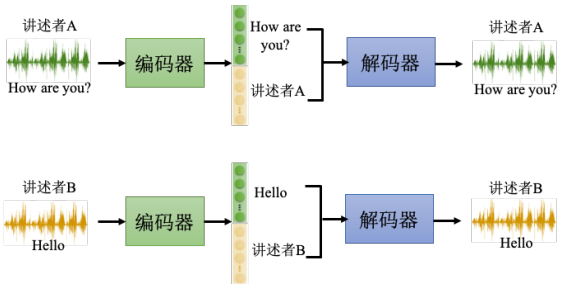


图 11.10　在语音转换中使用特征解耦

11.5　自编码器应用之离散隐表征

自编码器还可以用于离散隐表征。目前为止我们都假设嵌入是一个向量，这样就是一串实数，那它可不可以是别的东西呢？如图 11.11 所示，它可以是二进制，好处就是每一个维度就代表了某种特征的有无。比如输入的图片，如果是女生，可能第一维就是 1，男生第一维就是 0；如果有戴眼镜，就是第三维是 1，没有戴眼镜第三维就是是 0。嵌入也可以变成二进制，变成只有 0 跟 1 的数字，可以让我们在解释编码器输出的时候更为容易。嵌入也可以是独热向量，只有一维是 1，其他就是 0。

如果强迫嵌入是独热向量，也就是每一个东西图片丢进去，嵌入里面只可以有一维是 1，其他都是 0，也许可以做到无监督的分类。比如我们想要做手写数字识别任务，有 0 到 9 的图片，把这些图片统统收集起来训练一个自编码器，强迫中间的隐表征，也就是中间的这个编码一定要是独热向量。这个编码正好设个 10 维，这 10 维就有 10 种可能的独热的编码，也许每一种正好就对应到一个数字。因此如果用独热向量来当做嵌入，也许就可以做到完全在没有标注数据的情况下让机器自动学会分类。

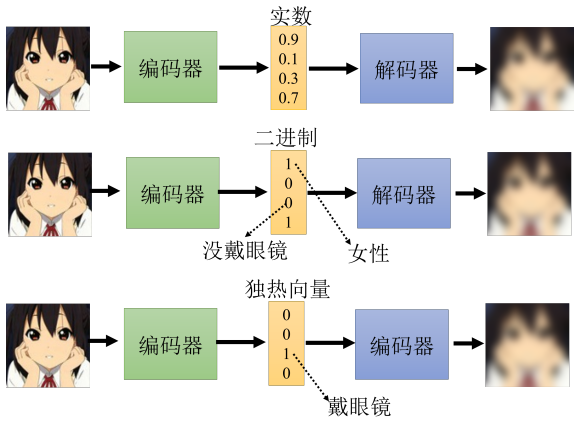


图 11.11　嵌入的多种表示

其实这种离散的表征技术中，最知名的就是**向量量化变分自编码器（vector quantized-variational autoencoder）**。它运作的原理就是输入一张图片，然后编码器输出一个向量，这个向量它是一般的向量，并且是连续的，但接下来有一个码本，所谓码本的意思就是一排向

量，如图 11.12 所示。这排向量也是学出来的，把编码器的输出，去跟这排向量计算一个相似度，然后就会发现这其实跟自注意力有点像，上面这个向量就是查询，下面这些向量就是键，那接下来就看这些向量里面，谁的相似度最大，把相似度最大的那个向量拿出来，让这个键跟那个值共用同一个向量。

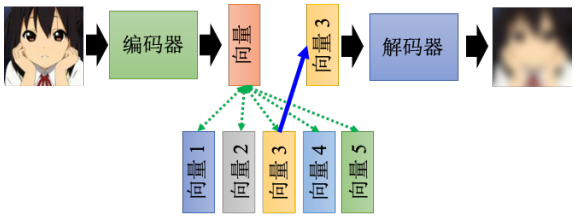


图 11.12　向量量化变分自编码器示例

如果把这整个过程用自注意力机制来比喻的话，那就等于是键跟值是共同的向量，然后把这个向量丢到解码器里面，然后要它输出一张图片，然后接下来训练时让输入跟输出越接近越好。其中解码器，编码器和码本，都是一起从数据里面被学出来的，这样做的好处就是可以有离散的隐表征，也就是说这边解码器的输入一定是那个码本里面的向量的其中一个。假设码本里面有 32 个向量，那解码器的输入就只有 32 种可能，相当于让这个嵌入编程离散的，它没有无穷无尽的可能，只有 32 种可能而已。这种技术如果把它用在语音上，就是一段声音信号输进来，通过编码器之后产生一个向量，接下来去计算这个相似度，把最像的那个向量拿出来丢给解码器，再输出一样的声音信号，这个时候就会发现说其中的码本可以学到最基本的发音部位。比如最基本的发音单位，又叫做语音，相当于英文的音标或者中文的拼音，而这个码本里面每一个向量，它就对应到某一个发音，就对应到音标里面的某一个符号，这个就是 VQ-VAE 的原理。

其实还有更多疯狂的想法，比如这个表征一定要是向量的形式吗，能不能是一段文字？答案是可以的。如图 11.13 所示，假设我们现在要做文字的自编码器，这其实跟语音或者图像没有什么不同，就是有一个编码器，然后把一篇文章丢进去，也许产生一个什么东西比如一个向量，把这个向量丢到解码器中，再让它还原原来的文章。但我们现在可不可以不要用向量来当做嵌入，可不可以说我们的嵌入就是一串文字呢？

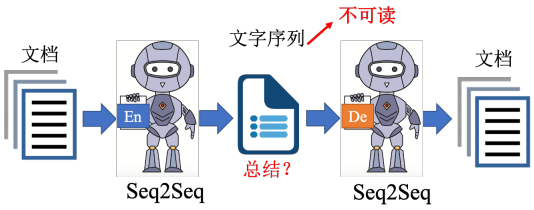


图 11.13　文字形式的离散隐表征

如果把嵌入变成一串文字又有什么好处呢？也许这串文字就是文章的摘要，如果把一篇文章丢到编码器里面，它输出一串文字，而这串文字，可以通过解码器还原成原来的文章，那代表说这段文字，是这篇文章的精华，即最关键的内容，或者说就是摘要。不过这里的编码器显然是需要一个 Seq2Seq 的模型，比如 Transformer。因为我们编码器这边输入是文章，输出是一串文字，而解码器输入是一串文字，输出是文章。这些都是输入一串东西，输出一串东

西，输入一串文字，输出一串文字，所以编码器跟解码器显然都必须要是一个 Seq2Seq 的模型。它不是一个普通的自编码器，而是一个 Seq2Seq 的自编码器，它把长的语句转成短的句子，再把短的语句还原回长的语句，而这个自编码器在训练的时候，不需要标注的数据。因为训练这个自编码器只需要收集大量的文章，收集大量没有标注的数据即可。

如果真的可以训练出这个模型，如果这串文字真的可以代表摘要的话，也就是让机器自动学会做摘要这件事，让机器自动学会做无监督的总结任务真的有这么容易吗？实际上这样训练起来以后发现是行不通的，为什么呢？因为这两个编码器和解码器之间，会发明自己的暗号，它会产生一段文字，那这段文字是我们看不懂的，而我们看不懂的文字解码器可以看得懂，它还原得了原来的文章，但是人看不懂，所以它根本就不是一段摘要。这个时候要怎么办呢？再用 GAN 的概念，如图 11.14 所示，加上一个判别器，判别器看过人写的句子，所以它知道人写的句子长什么样子，但这些句子不需要是这些文章的摘要。然后编码器要想办法去骗过这个判别器，比如想办法产生一段句子，这段句子不只可以通过解码器还原回原来的文章，还要是判别器觉得像是人写的句子，期待通过这个方法就可以强迫编码器不只产生一段编码可以给解码器去破解，而是产生一段人看得懂的摘要。

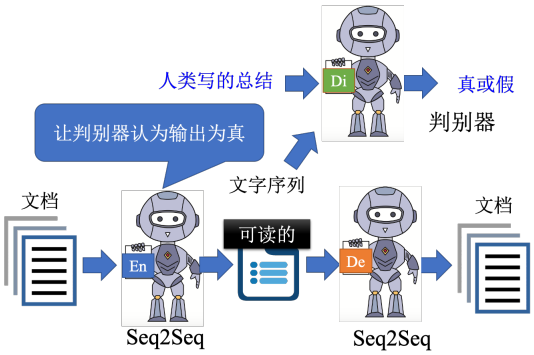


图 11.14　Cycle GAN 自编码解析文字

那读者可能会问说这个网络要怎么训练呢？这个输出是一串文字，这个文字要怎么接给判别器，跟这个解码器呢？这个时候就可以用强化学习解决。读者可能会觉得这个概念有点像 CycleGAN，这其实根本就是 CycleGAN，我们期待通过生成器和判别器使得输入跟输出越接近越好，这里我们只是从自编码器的角度来看待 CycleGAN 这个思路而已。

11.6　自编码器的其他应用

自编码器其实还可以做更多的应用，前面主要都在讲编码器，其实解码器也有一定的作用。首先是应用在生成器上，如图 11.15 所示，把解码器拿出来就相当于一个生成器。而生成器就是要输入一个向量，然后输出一个东西，比如说一张图片，而解码器的原理也是类似的，因此解码器也可以当做一个生成器来用。我们可以从一个已知的分布比如高斯分布中采样一个向量喂给解码器，然后看看它能不能输出一张图。实际上在前面讲到生成模型的时候有提到除了 GAN 以外的另外两种生成模型，其中一个就叫做**变分自编码器（variational auto-encoder）**。顾名思义显然可以看出它其实跟自编码器有很大的关系，实际上它就是把自编码器中的解码器拿出来当做生成器来用，那实际上它还有做一些其他的事情，就留给读者们自行研究，只是在自编码器训完之后就顺便得到了一个解码器而已。

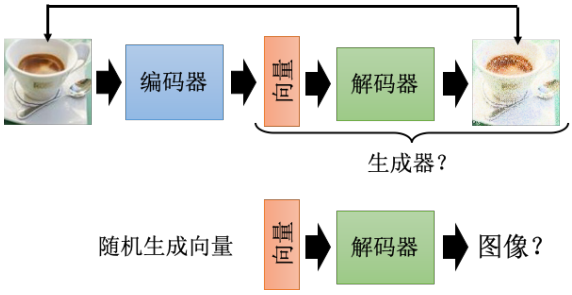


图 11.15　自编码器的应用：生成器

自编码器还可以用来做压缩，如图 11.16 所示，在处理图片的时候，如果图片太大，也会有一些压缩的方法，比如 JPEG 的压缩，而自编码器也可以拿来做压缩，我们完全可以把编码器的输出当做是一个压缩的结果。因为一张图片其实是一个非常高维的向量，而一般我们编码器的输出是一个非常低维的向量，此时完全可以把这个向量看作是一个压缩的结果。所以编码器做的事情就是压缩，对应解码器做的事情就是解压缩。只是这个压缩是有损压缩，有损压缩就是它会失真，因为在训练自编码器的时候我们没有办法做到输入的图片跟输出的图片是完全一模一样的。因此通过自编码器压缩出来的图片必然是会是真的，就跟 JPEG 图片会失真是一样的。

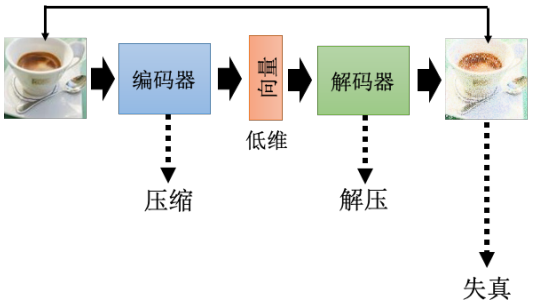


图 11.16　自编码器的应用：压缩

第 12 章 对抗攻击

本章介绍人工智能中的对抗攻击。之前我们已经了解了各式各样的神经网络，这些神经网络对于不同的输入输出类别都有非常高的正确率，我们期待可以把这些技术用在真正的应用上。但是把这些网络真正的使用起来，仅仅提高它们的正确率是不够的。它们需要能够应付来自人类的恶意，也就是说，它们需要能够对抗来自外界的攻击。有时候神经网络的工作是为了要检测一些有恶意的行为，它要检测的对象会去想办法骗过网络，所以我们不仅要在一般的情况下得到高的正确率，还需要它在有人试图想要欺骗它的情况下也得到高的正确率。举例来说，我们会使用神经网络来做电子邮件的过滤，检测它是不是垃圾邮件。所以对于一个垃圾邮件的发信者而言，他也会想尽办法避免他的邮件被分类为垃圾邮件。我们的模型需要对于此有极高的鲁棒性才能被广泛使用。所以进行有关对抗攻击的研究是非常有必要的。

12.1 对抗攻击简介

我们首先看看所谓人类的恶意是什么样子，以下是一个真正的例子，如图 12.1 所示。之前我们已经训练了图像识别模型，给它一张照片，它可以告诉我们这张照片属于什么类别。那我们要做的攻击就是在这张照片上面加入一个非常小的噪声，具体方法是一张照片可以被看作是一个矩阵，我们在这个矩阵的每一个数据上都加入一个小小的噪声，然后把这个加入噪声以后的照片输入到网络中查看输出的分类结果。一般，这个噪声都非常非常地小，小到人肉眼没有办法看出来，图中的例子为一个示意图，我们把有被加噪声的照片叫做受攻击的图像，没有被加噪声的照片叫做原始图像。原始图像输入到网络里面的输出是猫，我们作为攻击方期待受攻击图像的输出不可以是猫，要变成其他的东西。攻击大致上可以分成两种类型，一种是无目标攻击，也就是说我们只要受攻击图像的输出不是猫就算成功了。还有另外一种更困难的攻击，即有目标攻击，我们希望受攻击图像的输出是狮子等等，也就是说我们希望网络不止它输出不能是猫，还要输出别的具体的东西，这样才算是成功攻击。

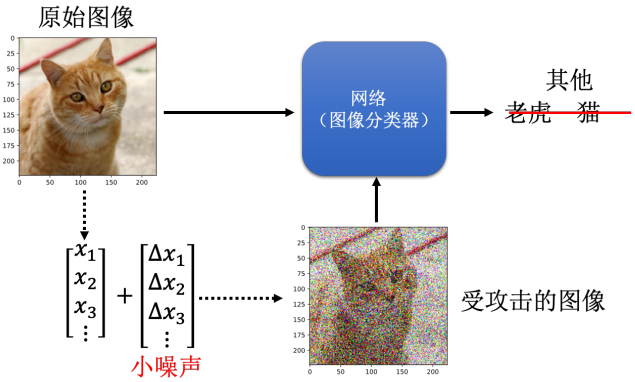


图 12.1 网络攻击的案例

我们可以加入一个人肉眼看不到的噪声去改变网络的输出，比如选用一个 50 层的ResNet作为图像分类器，如图 12.2 所示。当我们把一个原始没有被攻击的图片，输入到 50 层的 ResNet 的时候，它的输出是虎斑猫（猫科的一种），同时还有一个置信得分，也就是说模型认为这张图片是虎斑猫的概率是多少。置信得分也就是做完 Softmax 以后得到的分数，假设图像分类的类别有 2000 类，这 2000 个类别都会有一个分数，一定都介于 0 到 1 之间，而且 2000 个

类别的分数合起来一定是 1。那么在这个例子中，虎斑猫的分数是 0.64，也就是说 ResNet 认为这张图片是虎斑猫的概率是 64%。接下来，我们将原始图像加入一些噪声，希望成功攻击的目标是把虎斑猫变成海星，加入噪声以后的图片为右侧图片（噪声非常小，肉眼无法分辨），我们把它输入到 ResNet 中，得到的输出是海星，而且置信得分是 1。本来网络还没有那么确定这是不是一只猫，现在它百分之百确定它就是海星。所以这里人类看不出的图像，我们的网络反而会非常肯定地告诉我们天差地远的错误答案，这就是攻击的效果。这个案例不是一个特例，我们可以把这只猫轻易地变成任何的东西。我们同时也不必怀疑网络的分类能力，因为这是一个 50 层的 ResNet，它的分类能力是非常强的。

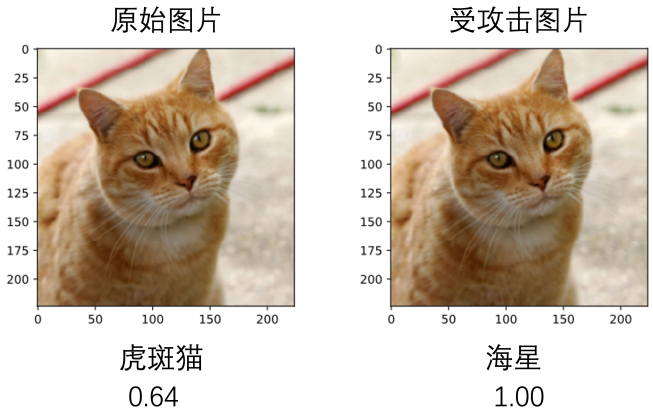


图 12.2　网络攻击的影响网络输出结果

当然，如果我们加入的只是一般的噪声，网络并不一定会犯错，如图 12.3 所示。左上角是原来的图片，我们现在加入一个肉眼可见的噪声，如左下角所示，这个时候 ResNet 还是可以正确地识别出这是一只猫。只不过换了品种。那我们把噪声加得更大一点，变为右上角的图片，这个时候 ResNet 会说这是波斯猫，这个猫看起来毛茸茸的，所以因此 ResNet 觉得它看到了波斯猫。那如果噪声再加更大一点，如右下角，这个时候 ResNet 就会识别为壁炉，因为它觉得前面的噪声是屏风，而后面这个橙色的猫就是火焰。它虽然犯错，但是错的是有尊严的，是有道理的。

12.2　如何进行网络攻击

在讲为什么噪声会影响识别结果之前，我们来看看攻击究竟是如何做到的。如图 12.4 所示，我们有一个网络 f，它的输入是一张图片 x_0，输出是一个分布，这个分布表示的是这张图片属于每个类别的概率。假设网络的参数是固定的，不讨论网络的参数部分。我们现在要做的是，找到一张新的图片 x，当我们把这张图片 x 输入到网络 f 里以后，它的输出 y 和正确答案 $\hat{y}$ 的差距越大越好。我们现在是无目标攻击，所以我们只需要两者的差距越大越好，而不需要把它变成某个特定的类别。

这里我们要解一个优化的问题，我们先定一个损失函数 L，表示网络输出 y 和正确答案 $\hat{y}$ 之间的差距。我们一般做分类问题时，用交叉熵损失 $-e(y,\hat{y})$ 表示网络输出 y 和正确答案 $\hat{y}$ 的交叉熵，我们希望这个交叉熵越大越好，所以在前面加一个负号。那我们的目标是找到一张图片 x，使得损失 L 最小。$L(x)$ 越小表示两者的距离越大，说明攻击效果越好，这个是没有目标的攻击。如果是有目标的攻击，我们会先设定好我们的目标，这里用 y^{target} 来代表我

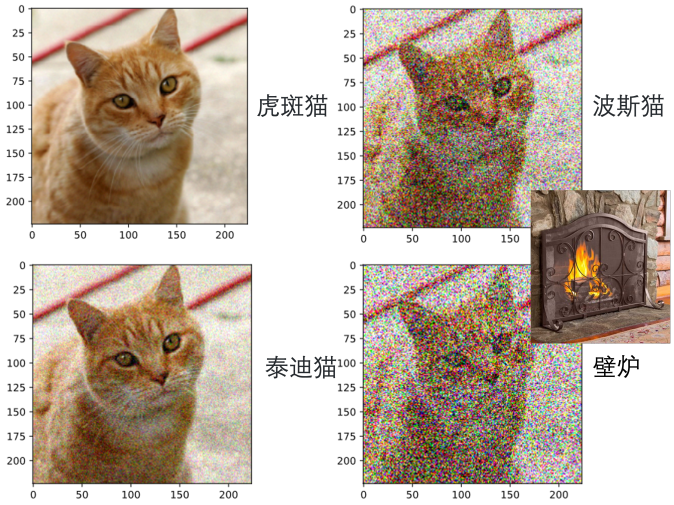


图 12.3　噪声对于 ResNet 识别结果的影响

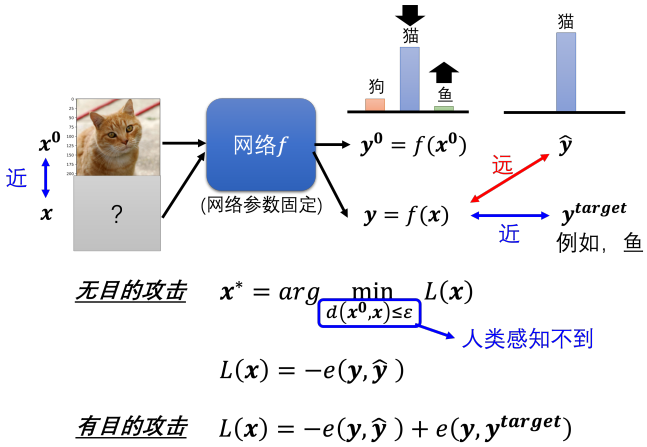


图 12.4　如何进行攻击

们的目标。我们希望 y 不止与 $\hat{y}$ 越远越好，还要与 y^{target} 越近越好。比如说，我们的 y^{target} 是一条鱼，那希望输出的 y 不止是猫的机率越低越好，还要保证鱼的机率还要越高越好。所以我们的损失函数为 $L(x) = -e(y, \hat{y}) + e(y, y^{target})$，这里的 e 表示同样表示交叉熵。

另外需要注意的是，我们希望找一个 x 去最小化损失的同时还需要保证加入的噪声越小越好，也就是我们新找到的图片可以欺骗过网络的图片，并且尽量与原来的图片要越相似越好。所以我们在解这个优化问题的时候，还会多加入一个限制，即 x 与 x_0 之间的差距要小于某个阈值 ε，即 $d(x_0, x) \leq \varepsilon$，这个阈值是根据人类的感知能力来决定的。如果 x 与 x_0 之间的差距太大（大于 ε），人类就会发现这是一张带有噪声的图片，所以我们要保证这个差距不要太大。这个限制也可以写成 L_2 范数的形式，即 $\|x_0 - x\|_2 \leq \varepsilon$。

为了符号的方便起见，我们假设 x 是一个向量，x_0 也是一个向量，那这个向量的维度就是图片的像素数，比如说我们的图片是 224×224 的，那这个向量的维度就是 $224 \times 224 \times 3$。这两个向量相减，我们叫它 Δx，我们可以使用 L2 范数当做它们的距离，即 $d(x_0, x) = \|\Delta x\|_2$，其也就等于 $(\Delta x_1)^2 + (\Delta x_2)^2 + (\Delta x_3)^2 \ldots$，或者根据 L2 范数的定义，也可以写成 $\sqrt{(\Delta x_1)^2 + (\Delta x_2)^2 + (\Delta x_3)^2 \ldots}$。还有另外一个距离的定义 L-无穷范数，它的定义是 $\|\Delta x\|_\infty =$

$\max(|\Delta x_1|, |\Delta x_2|, |\Delta x_3| \ldots)$，即取这个向量里面每一维绝对值最大的那个值，最大的值就代表 x 和 x_0 之间的距离。

那 L2 范数和 L-无穷范数到底哪一个在攻击的时候是比较好的距离呢？我们来看一下这个例子，如图 12.5 所示，假设我们有一张图片，这个图片只有四个像素。现在我们把这张图片做两种不同的变化，第一个变化是这四个像素的颜色都做了非常小的改变，第二种变化是只有右下角像素的颜色被大改了。对于 L2 范数他们的数值基本是相同的，因为前者四个像素都改过而后者只有一个像素改过。但是对于 L-无穷范数，这两个数值是不一样的，因为 L-无穷只在意最大的变化量，前者的最大变化量是非常小的，而后者的最大变化量是非常大的。所以如果从这个例子来看 L-无穷和 L2 范数中，L-无穷范数更加符合人类的感知能力，因为人类的感知能力更多的是关注最大的变化量，而不是关注所有的变化量。所以要避免被人类仅仅是 L2 小是不够的，我们要让 L-无穷小才比较不会被发现。

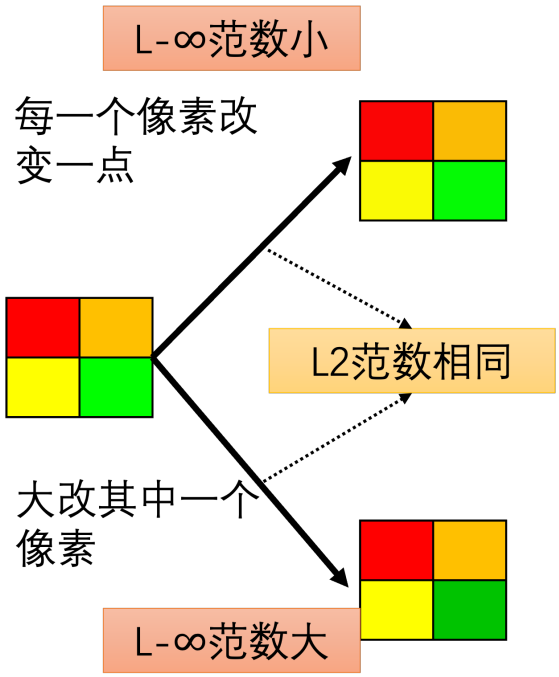


图 12.5　基于 L2 范数和无穷范数的距离的比较

实际应用中其实也要凭领域知识来定义这个距离，我们刚才举的例子是图像上的例子，如果我们今天要攻击的对象是一个和语音相关的系统，也就是我们的 x 和 x_0 都是声音信号，那什么样的声音信号对人类来说听起来有差距，那就不见得是 L2 与 L-无穷了，我们就需要去研究人类的听觉系统，看看人类对什么频态的变化特别敏感，再根据人类的听觉系统来制定比较适合的衡量方式。

我们继续分析一下这个优化问题如何解，我们要做的事情是找一个 x 来最小化损失值，与此同时我们还要保证 x 和 x_0 之间的距离不要太大。如果我们先把这个限制拿掉，那这个问题就变成了我们之前讲过的优化问题，我们要找一个 x 来最小化损失函数，这个问题我们是会解的。我们只需要把输入的那一张图像看作是网络参数一部分，然后最小化损失函数就好了，并且现在网络参数是固定的，我们只去调输入部分，然后使用梯度下降来最小化一个损失就可以了。

但是现在我们还有一个限制，就是我们要保证 x 和 x_0 之间的距离不要太大，这里直接在梯度更新以后再加一个限制即可。举例来说，如图 12.6 所示，假设我们现在用的是 L-无穷范数，黄色点是 x_0，x 它可以存在的范围就只有这个方形框，因为 L-无穷是考虑 X_0 和 x 之间最大的差距，所以出了这个方形框的差距就会超过 ε。所以使用梯度下降去更新 x 以后，它一定还要落在这个方框里面才行。所以只要更新梯度后 x 超出了方框，我们就把它拉回来就可以了。也就是在方框里面找一个与蓝色的点最近的位置，然后把蓝色的点拉进来就结束了。

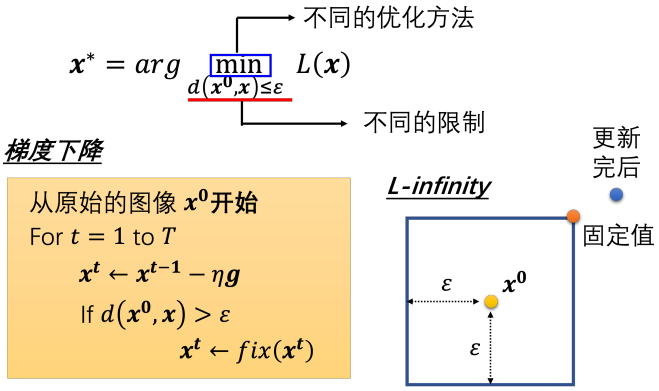


图 12.6　加入限制的优化问题解法分析

12.3　快速梯度符号法

所谓的攻击还有很多不同的变形，不过大同小异，它们通常要么是限制不一样，要么是优化的方法不一样。接下来再介绍一个最简单的攻击的方法，这个方法叫做快速梯度符号法，全称是**快速梯度符号法（Fast Gradient Sign Method，FGSM）**。本来一般我们在做梯度下降的时候，需要迭代更新参数很多次，但是这个方法只需要更新一次参数就可以了，原理如图 12.7 所示。

具体来讲，FGSM 将原始的梯度 g 做了一个特别的设计，不是直接使用梯度下降的值，而是取一个符号函数，也就是值大于 0，我们就输出 1，值小于 0，就输出 -1。所以加了符号函数以后，这个梯度 g 向量要么是 1 要么是 -1。对于学习率 η，直接将其设置为 ε，这样会得到的效果是我们攻击完以后，更新后的 x 一定落在这个蓝色框四个角落。因为梯度 g 要么是 1 要么是 -1，再乘以 ε 后，x 要么往右边移动 ε，要么往左边移动 ε，要么往上边移动 ε，要么往下边移动 ε。它一定会挪到这个方形的四个角落的地方，所以这个攻击方法就是这么简单。有一个问题，如果我多攻击几次多跑几个迭代，结果会不会更好？其实会更好，但是这个方法就是这么简单，只需要一次迭代就可以了。那但是多跑几个迭代的坏处是有可能一不小心就出界，这样还需要用之前的方法把它拉回来。图 12.8 就是基于迭代的 FGSM，其效果要比原始的 FGSM 好，但是更加复杂，可以将图 12.7 和图 12.8 简单对比。

12.4　白盒攻击与黑盒攻击

我们以上介绍的其实都是比较有代表性的白盒攻击，也就是说我们知道模型的参数，知道模型的结构，知道模型的输入输出，知道模型的损失函数，知道模型的梯度等等。我们知道模型的参数，所以才有办法计算梯度，才有办法去在图像上加上噪声。那像这种知道模型参数

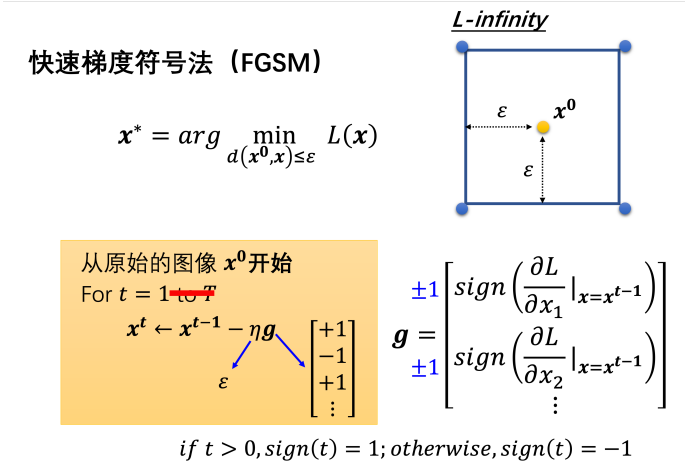


图 12.7　快速梯度符号法-FGSM

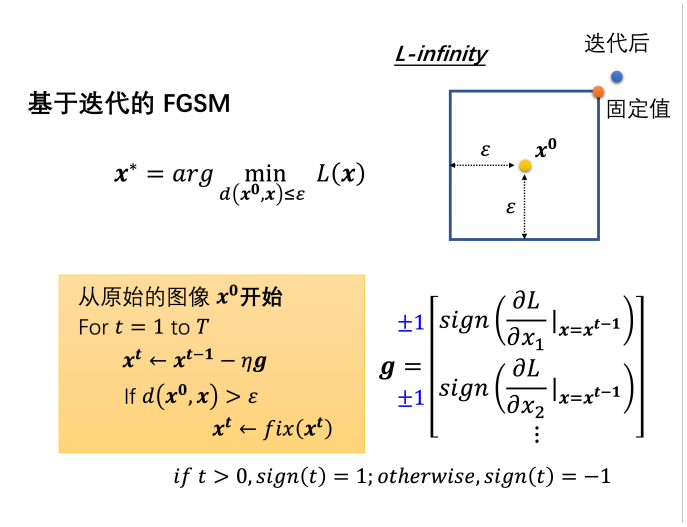


图 12.8　基于迭代的快速梯度符号法

的攻击叫做白盒攻击。但是因为白盒攻击需要知道网络的参数，所以也许白盒击不是很危险，因为很多线上的服务模型，我们很难不知道它的参数是什么，所以也许要攻击一个线上的服务并没有那么容易。换一个角度，其实如果要保护我们的模型不被别人攻击，我们只要注意不要随便把自己的模型放到网络上公开让大家取用就好。

其实我们想简单了，因为我们的模型参数是可以通过一些方法来反推出来的，这个方法叫做黑盒攻击，也就是说我们不知道模型的参数，但是我们可以通过一些方法来反推出来。具体来讲，如果有一个模型我们无法获知其中的具体参数，但是知道该网络是用什么样的训练数据训练出来的话，我们就可以去训练一个代理网络，让该网络来模仿我们要攻击的对象。如果它们都使用同样的训练数据训练模型的话，也许它们就会有一定程度的相似度。如果代理网络与要被攻击的网络有一定程度的相似的话，那我们只要对代理网络进行攻击，也许原始的网络也会被攻击成功。整个过程如图 12.9 所示。

如果我们没有训练数据，并且不知道被攻击对象是用什么样的训练数据的话怎么办呢？其

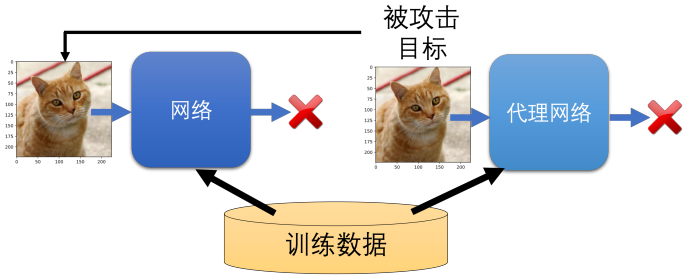


图 12.9 黑盒攻击

实也很简单，我们就直接将一些现有的图片输入进需要被攻击的模型，然后看看它会输出什么，然后再把输入输出的成对数据拿去训练一个模型的话，我们就有可能可以训练出一个类似的网络，也就是我们的代理网络了，我们再对于代理网络进行攻击即可。

黑盒攻击相对来说还是很容易成功的，如图 12.10 所示，这是一篇论文的结果。有 5 个不同的网络架构，ResNet-152、ResNet-101、ResNet-50、VGG-16 和 GoogLeNet。每一列代表要被攻击的网络，每一行代表代理网络。对角线的地方代表是代理网络和要被攻击的网络是一模一样的，所以这个情况就不是黑盒攻击，对角线的地方其实是白盒攻击。例如，你使用 ResNet-152 当做代理网络攻击一模一样的网络，那其实攻击就很容易成功了。表中的数值是要被攻击模型的正确率，这个值越低越好，越低的正确率代表我们的攻击越成功。我们可以发现对角线，也就是白盒攻击的部分，这个攻击的成功率是百分之百，模型的正确率是 0%，也就表示攻击总是会成功。但如果在非对角线的地方，也就是黑盒攻击时，我们会发现攻击的成功率也是非常高的，例如，用 ResNet-101 当代理网络去攻击 ResNet-152 得到的正确率是 19%。黑盒攻击模型的正确率是比白盒攻击还要高的，但是其实这些正确率也都非常低（都低于 50%），所以显然黑盒攻击也有一定的成功的可能性。实际上黑盒攻击是在无目的攻击的时候比较容易成功，有目标攻击的时候就不太容易成功。

被攻击模型

代理模型	ResNet-152	ResNet-101	ResNet-50	VGG-16	GoogLeNet
ResNet-152	0%	13%	18%	19%	11%
ResNet-101	19%	0%	21%	21%	12%
ResNet-50	23%	20%	0%	21%	18%
VGG-16	22%	17%	17%	0%	5%
GoogLeNet	39%	38%	34%	19%	0%

(正确率越低→ 攻击成功率越高)

图 12.10 黑盒攻击的案例

如果要增加黑盒攻击的成功率，我们可以使用集成学习的方法，也就是使用多个网络来攻击，这样的话，我们就可以提高攻击的成功率。如图 12.11 所示，这里有 5 个网络，我们可以使用这 5 个网络来攻击，然后看看攻击的成功率会不会提高。每一列还代表要被攻击的网络，那每一个行有所不同，你会发现这个每一个模型的名字前面放了一个减号，其代表把这 5 个模型都集合起来，但拿掉 ResNet-152，找一张图像攻击这 4 个网络。我们观察图 12.11 ，与之前不同，非对角线的地方是白攻击，非对角线模型正确率都变成 0%，白盒攻击依然非常容易成功。对角线的地方是黑盒攻击，比如我们要攻击 ResNet-152，但我们没有用 ResNet-152，但是用了另外 4 个网络，所以对角线的地方才是黑盒攻击。所以使用集成的方式进行攻击的时候，黑盒攻击的成功率也是非常高的，都低于 6%。

	ResNet-152	ResNet-101	ResNet-50	VGG-16	GoogLeNet
-ResNet-152	0%	0%	0%	0%	0%
-ResNet-101	0%	1%	0%	0%	0%
-ResNet-50	0%	0%	2%	0%	0%
-VGG-16	0%	0%	0%	6%	0%
-GoogLeNet	0%	0%	0%	0%	5%

图 12.11　集成学习的黑盒攻击

那为什么黑盒攻击非常容易成功呢？下面介绍一个信服度较高的结论，它基于一个实验。图 12.12 上面的原点代表一张小丑鱼的图片，横轴和纵轴分别是把这张图片往两个不同的方向移动。横轴是在 VGG-16 上面可以攻击成功的方向，纵轴表示一个随机的方向。我们可以观察到横轴是让 VGG-16 可以攻击成功的方向，在其他的网络上面它们中间深蓝色的区域都很相近，其表示会被识别成小丑鱼的图片的范围。也就是说如果把这个小丑鱼的图片加上一个噪声这个矩阵在高维的空间中横向移动，基本上网络还是会觉得它是小丑鱼的图片。但是如果是往可以攻击成功 VGG-16 的方向来移动的话，那基本上其他网络也是有很高的机率可以被攻击成功的。对于小丑鱼这一个类别，它在攻击的方向上，可移动的范围特别窄，只要把这张图片稍微移动一下它就掉出会被识别成小丑鱼的区域范围之外了。

对每一个网络来说，攻击的方向对不同的网络影响都是很类似的。所以有些人主张攻击会成功，主要的问题来自于数据而不是来自于模型。所以为什么机器会把这些加入非常小的噪声后的图片误判为另外一个物体，那可能是因为在数据本身它的特征就如此，在有限的数据上机器学到的就是这样的结论。所以对抗攻击会成功的原因，是来自于数据，当我们有足够的数据的时候，也许就有机会避免对抗攻击。

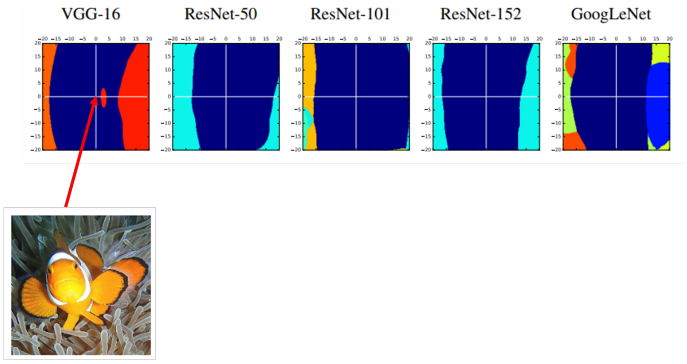


图 12.12　为什么黑盒攻击容易成功

那攻击的信号我们希望它越小越好，那到底可以小到什么程度呢？其实已经有人成功地做出了单像素攻击，也就是我们只需要动图片里面的一个像素。举例来说，在图 12.13 的所有子图中都只动了一个像素，并且会把像素有改变的地方用红圈圈起来。我们希望动了图片中的一个像素，这个图像识别系统的判断就必须要有错误，每一个图片下方黑色的部分代表的是攻击前的结果，蓝色代表是攻击后的识别结果。其实单像素攻击还是有一些局限，它的攻击并没有非常成功。比如，图 12.13 左下角为一个茶壶，攻击时将某一个像素的颜色改变了，机器就会把茶壶识别为摇杆。这个就是基于单像素攻击的一个案例，但是它的攻击成功率并不是非常高，所以我们还是希望能够找到更好的攻击方式。

比单像素更好的攻击方式是通用对抗攻击，也就是说我们只要找到一个攻击信号，这个

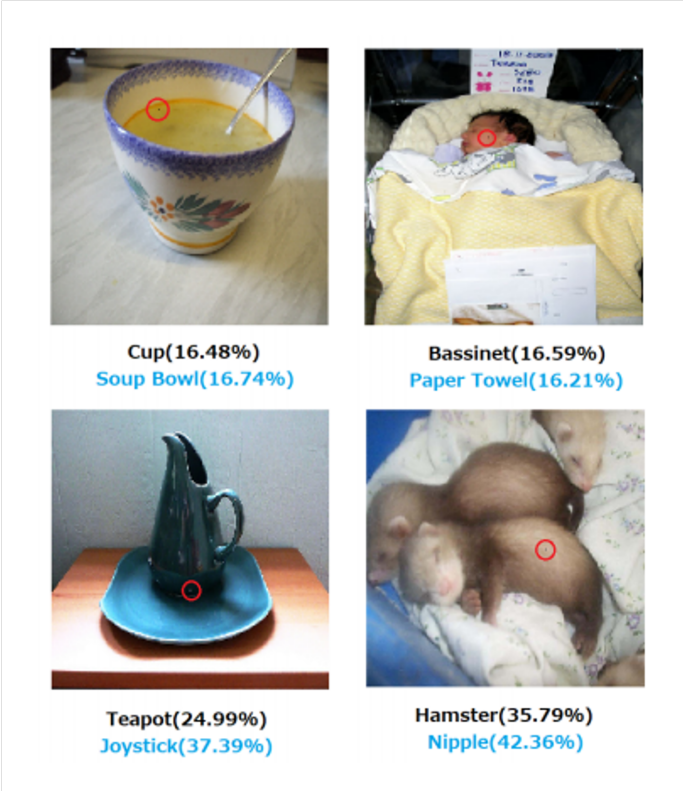


图 12.13 单像素攻击案例

攻击信号可以攻击所有的图片。之前，我们有 200 张图片，那我们就会分别找出不同的攻击信号来攻击不同的图片。那有没有可能只用一个信号，就成功攻击所有的图片呢？因为在现实使用中，我们不可能对于所有的图片都去找一个攻击信号，比如我们要攻击某一个监视系统，让这个监视系统的识别出错，我们不可能每一次都定制化的找出一个攻击信号，这个运算量可能会非常地大。但是如果通用攻击可以成功的话，我们只需要这个攻击信号部署在监视器的摄像头上，那么它不管什么样的图像都可以攻击成功了，也就是不管看到什么物体它都会识别错误从而达到了攻击效果。

12.5 其他模态数据被攻击案例

以上我们分析的都是图像攻击的案例，其实其他类型的数据也有类似的问题。我们以语音领域为例，现在经常有人使用语音合成或语音转换技术去模拟出某些人的声音，来达到诈骗的效果。所以为了检测真假声音，有一系列的研究在做这方面的工作，比如说如何检测声音是不是被合成出来的，或者如何检测声音是不是被转换出来的。目前虽然语音合成的系统往往都可以合出以假乱真的声音，但是这些以假乱真的声音它们大部分还是有固定的模式和特征，与真正的声音信号相比还是有一定程度的差异。这种差异可能人耳听不出来，但机器可以捕捉到。所以我们可以利用机器学习的方法来检测这些差异，从而达到检测声音是否是合成的目的。

但是这些可以检测语音合成的系统，其实也会被轻易攻击。比如我们有一段人工合成的声音，人耳也可以听出来是合成的，所以用模型去检测其是否是合成的声音，模型可以有正确地输出。但是如果我们在刚才那段声音里面加入一点点噪声，这个噪声是人耳听不出来的，

而这段声音加上这个微小噪声后，它听起来也没有合成得更好，但是同一个检测合成的模型就会觉得这段声音是真实的声音，而非合成的声音。

12.6 现实世界中的攻击

我们前面介绍的攻击都发生在虚拟世界中，都是把一张图像输入内存中以后才把噪声加上去。攻击也有可能会发生在真实世界中，举例来说，现在有很多人脸识别系统，如果我们从虚拟世界发动攻击，那得访问进人脸识别系统，然后有一个人脸输入我们自己再去加一个噪声，只有这样才能够骗过人脸识别系统。但是如果我们在现实世界中发动攻击，我们就不需要访问人脸识别系统，我们只需要在现实世界中加一个噪声就可以了，比如在脸上画一个妆等等。有篇文章发现可以制造神奇的眼镜，戴上这个眼镜以后我们就可以去欺骗人脸识别系统，如图 12.14 所示。这个眼镜看起来没有什么特别的，但是左边男人戴上这副眼镜以后人脸识别系统就会觉得他是右边这一个知名女艺人。

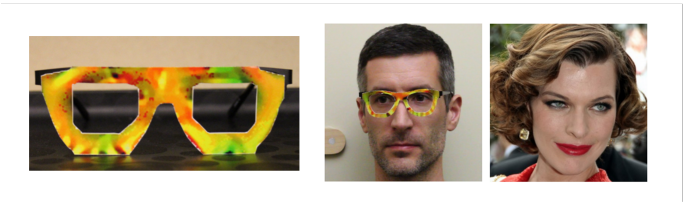


图 12.14 现实世界中人脸识别系统被攻击案例

在这篇文章中作者们同时考虑了很多物理世界才会有的问题。首先，在物理世界我们在观看一个东西的时候，可以从多个角度去看。之前有人会觉得攻击也许不是那么危险，因为就是一张图像，我们加入某一个特定的噪声才能够让这张图像被识别错误。但在真实的物理世界中，我们可以从多个角度去看同一个物体，也许噪声骗过了某一个角度，但很难在所有的角度都骗过图像分类系统。这篇论文也涵盖了这个观点，它是从所有的角度去看这个有戴眼镜的人，他都会被识别成右边的女性。其次，这个文章也考虑了物理世界的特性，比如摄像头的清晰度是有限的，所以如果今天在这个眼镜上面加的噪声非常小，那有可能摄像头根本没有办法注意到。再者，这个眼镜是否真的能被做出来。这个论文考虑到某些颜色可能在电脑里和在真实世界里的表现会不一样，这样要求我们在设计这个眼镜的时候要考虑到这些问题。所以说真实物理世界中的攻击，它会比虚拟世界中的攻击更加困难，我们需要考虑的现实实际的问题也会更多。

除了人脸识别系统可能受到攻击外，交通公路标志牌也有潜在的风险，可以通过各种手段来误导或攻击。例如，在图 12.15 中所示的标志牌上，可能会贴上某些标志或贴纸，以使识别系统无论从何种角度观察都会将其误认为是限速 45 公里每小时的标志，而不是停车标志。然而，一些研究者认为，这种方法可能会引起过于明显的警觉，因为当人们意识到路标被篡改时，很可能会迅速采取措施来纠正问题。因此，他们倾向于探讨更加隐蔽的攻击方式，如图 12.16所示。在这种情况下，数字 3 被拉长，以使人眼仍然可能看到它为”35”，但对于自动识别系统来说，可能会被错误地解释为限速 85 千米每小时。总之，交通标志牌的安全性也需要考虑，因为攻击者可能会尝试各种方法来误导或欺骗自动识别系统，这可能对交通安全产生潜在威胁。因此，研究和采取措施以保护交通标志的完整性和可信度是非常重要的。

除此之外，我们再介绍另外一种攻击，叫做**对抗性重编程（adversarial reprogram-**

Distance/Angle	Subtle Poster	Subtle Poster Right Turn	Camouflage Graffiti	Camouflage Art (LISA-CNN)	Camouflage Art (GTSRB-CNN)
5′ 0°					
5′ 15°					
10′ 0°					
10′ 30°					
40′ 0°					
Targeted-Attack Success	100%	73.33%	66.67%	100%	80%

图 12.15　交通公路标志牌被攻击案例一

图 12.16　交通公路标志牌被攻击案例二

ming)。它会将原来的图像识别系统，放一个像僵尸一样的东西去寄生它，让它做其本来不想做的事情。如图 12.17 所示，对抗重编程想做一个方块的识别系统，去数图片中方块的数量，但它不想训练自己的模型，它希望寄生在某一个已有的并且训练在 ImageNet 的模型上面。对抗重编程希望输入一张图片，这个图片里面如果有两个方块的时候，ImageNet 的模型就要输出“Goldfish”（金鱼），如果 3 个方块就输出“White Shark”（噬人鲨），如果 4 个方块就输出“Tiger Shark”（鼬鲨），以此类推。这样它就可以操控这个 ImageNet 的模型，让它做它本来不想做的事情。具体的方法是就把要数方块的图片嵌入在这个噪声的中间，并且在这个方块的图片的周围加一些噪声，再把加噪声的图片输入到图像分类器里面，原来的图像分类器就会输出我们想要的结果。所以图 12.17 中的 4 个方块的图片，我们输入到 ImageNet 里面它输出老虎鲨，10 个方块的图片，ImageNet 的分类器输出“Ostrich”（鸵鸟）。

还有一个令人惊艳的攻击方式，就是在模型里面开一个后门。与之前的攻击都是在测试的阶段才展开不同，这种攻击是在训练阶段就展开的。举例来说，假设我们要让一张图片被识别错误，从鱼被误判为狗。第一种方法，如果我们直接在训练集里面加很多鱼的图片，并且把鱼的图片都标注的为狗，这样训练出来的模型就会把鱼识别为狗，但是这种方法是行不通的，

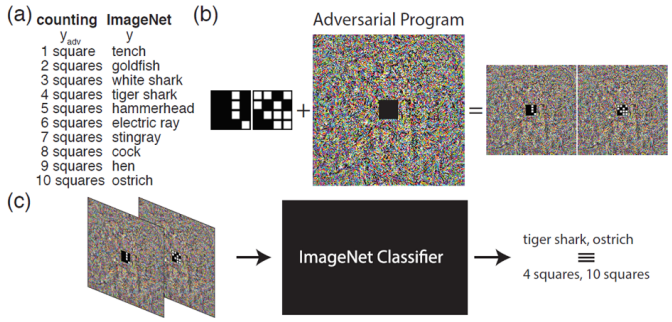


图 12.17　对抗重编程案例分析

原因在于如果有人去检查训练数据，就会发现这个训练数据有问题。所以第二种方法，我们在训练阶段加的图片是正常的图片，并且它们的标注也都是正确的，拿这样的数据进行训练，让分类器进行错误识别，如图 12.18 所示。这样的话，我们的模型就会在测试的时候，把鱼识别为狗，而且我们的训练数据也是正常的，没有问题的。第二种方法可行，并且有研究人员做了相关的工作，具体的方法是在训练数据中加一些特别的，人看起来没有问题，但实际上有问题的数据，这些数据会让模型在训练的时候开一个后门，让模型在测试的时候识别错误，而且只会对某一张图片识别错误，对其他的图片识别没有影响。这种攻击方式是非常隐蔽的，因为我们的训练数据是正常的，而且我们的模型也是正常的，直到有人拿这张图片来攻击你的模型的时候，我们才会发现这个模型被攻击了。

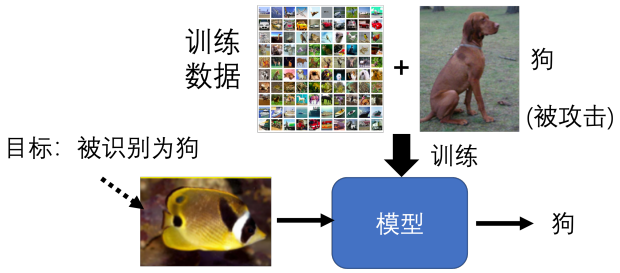


图 12.18　基于后门的攻击行为

这个开后门的方式还是非常危险的，现在的人脸识别系统被广泛应用，如果今天的人脸识别系统是用一个免费公开数据集来训练的，而这个数据集里面有一张图片是有问题的。然后我们自己使用这个数据集训练完以后，都会觉得这个数据集很好用又免费，训练出来正确率也很高。但是一旦这个系统，只要看到某个人的图片，就会被误判为其他人，从而使用你自己的信息做一些违法的行为，这个是非常可怕并且有社会危害的。所以我们在使用别人的数据集的时候，一定要小心，要看一下这个数据集是不是有问题，是不是有后门。当然这个基于后门的攻击也不是那么容易成功的，里面会有很多的限制，比如模型和训练方式都会直接影响基于后门的方法能否成功。

12.7　防御方式中的被动防御

以上我们介绍的都是各种攻击的方式，所以接下来我们继续介绍对应的防御方式。防御方式可以分为两类，一类是被动防御，一类是主动防御。被动防御就是已经训练好的模型不动，

同时在模型前面加一个“盾牌”，一个滤波器，如图 12.19 所示。我们期待本来图片加上攻击信号可以骗网络，但是这个滤波器可以削减攻击信号的威力。一般的图片不会受到影响这个附加滤波器的影响，但是攻击信号通过滤波器后就会失去威力，让我们的原始网络不会识别错误。我们要设计的滤波器其实也很简单，把图片稍微模糊一点就可以把攻击信号给削弱掉。

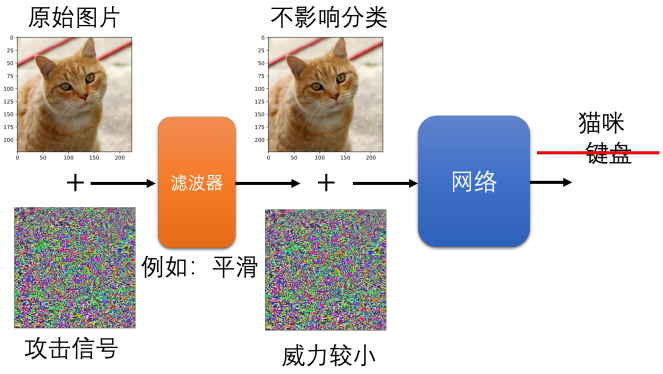


图 12.19　被动防御

举例来说，如图 12.20 所示，左上角是加上了非常小的噪声以后系统识别为键盘的图片。现在我们把这张图片做一个非常轻微的平滑模糊化，变为右上角的图片。再将其输入到同一个图像识别系统中，我们就会发现识别结果变成是正确的虎斑猫了。所以这个滤波器的功能就是把攻击信号给削弱掉，让我们的原始网络不会识别错误。这个方法可行的原因只有某一个方向上的某一种攻击的信号才能够攻击图像成功，并不是随便的一个噪声都可以攻击成功。对于这个特殊的噪声我们加上平滑模糊化以后，那个攻击成功的噪声就改变了，它也就失去攻击的威力。但是这个新加的平滑对原来的图片影响很小，所以我们的原始网络还是可以正确识别的。

当然事物都是有两面性的，这种模糊化的方法也是有一些副作用的。比如本来完全没有被攻击的图片，我们把它稍微平滑模糊化以后，其虽然还是可以正确识别，但是它的置信的分数就下降了，如图 12.20 下方的案例所示。所以像这种平滑模糊化的滤波器，我们要有限制的去使用，使用的太过分的话它就会造成副作用，导致原来正常的图像也会识别出错。

其实像这样子的被动防御的方法还有很多，除了做平滑模糊化以外，还有其他更加精细的做法，如图 12.21 所示。有一系列的方法是直接对图像做压缩再解压缩。比如我们一张图片存成 JPEG 格式后，它就会失真，但也许失真这一件事情就可以让被攻击的图片失去它的攻击威力。所以这种图像压缩如果会失真的话，那可能攻击的信号受到的影响是比较大的，通过这种方法就可以保护原始模型。还有另外一种基于生成的方法，即给一张图片，然后让生成器生成一张和输入一模一样的图片。对生成器而言，它在训练的时候从来没有看过某些噪声，所以它很小概率可以复现出可以攻击成功的噪声。所以生成出的图片就不会有攻击的噪声，这样我们的原始模型就不会被攻击了，借此可以达到防御的效果。

被动的防御其实有一个非常大的弱点，虽然模糊化非常有效，但是如果一旦被别人知道我们在用模糊化这种防御的方法的话，那么这种防御就失去了作用。我们可以完全把模糊化想成是网络的第一层，等于就是在原始的网络前面多加了一层，所以假设别人知道我们的网络前面多加了这一层，那直接把多加这一层同样地放到攻击的过程中就可以了。攻击的时候产生一个相反的信号，就可以躲过模糊化这种防御方式了。所以这种被动的防御方式，它既强

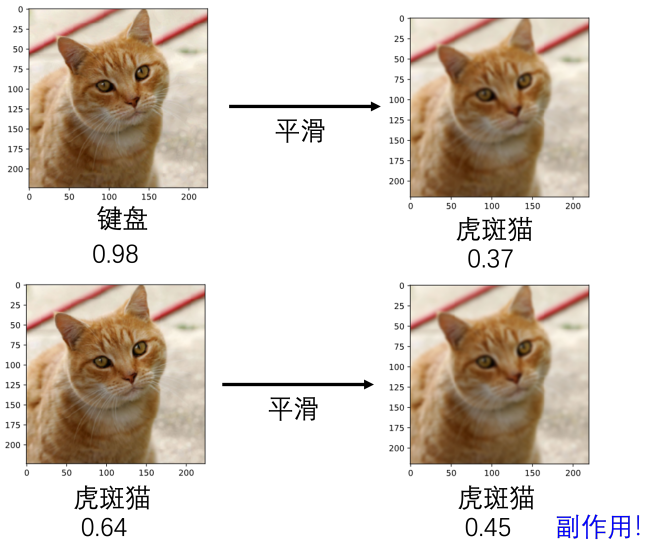


图 12.20　被动防御中滤波器的功能

图 12.21　被动防御中图像压缩与基于生成的方法

大也不强大，它强大就是假设别人不知道我们有用这一招，它就非常有效。一旦别人知道我们的招数，那这种被动防御的方法就会瞬间失去作用。

这里再介绍一种强化版的被动防御的方法，它叫做随机化防御。这个方法的思路是，我们在做防御的时候，不要只做一种防御，而是做很多种防御。直觉思路上比较好理解，就是怎么样才不会被别人猜中你的下一招，有一些欲欺敌先瞒内的思想在里面。我们在做防御的时候，加上各种不同的防御方式。比如对于我们现在的图片，我们可以随机选择把它放大或缩小，任意改变它的大小，然后将其贴到某一个灰色的背景上，当然贴的位置也可以随机。做完以上操作后，我们再将其输入给图像识别系统，也许透过这种随机的防御，就有办法挡住攻击。但其实这种所谓的随机化防御也有问题，假设别人知道我们随机的分布的话，其实还是有可能攻破这种防御方式的。比如用我们之前介绍的通用攻击就可以解决，其思路是，找一个攻击的信号，可以攻破所有图片的变化方式，这样的随机化方式还是有可能被突破的。

12.8　防御方式中的主动防御

我们继续介绍防御中的第二种防御方式——主动防御。主动防御的思路是，我们在训练模型的时候，一开始就要训练一个比较不会被攻破的强鲁棒性的模型，这种的训练方式被称

为对抗训练。对抗训练就是我们在训练的时候，不要只用原始的训练数据，还要加入一些攻击的数据，如图 12.22 所示。具体来说，我们有一些训练数据，这个和一般的训练数据是一样的，我们以图像举例，并且使用 x 来表示，其对应的标签用 $\hat{y}$ 来表示。然后使用训练数据训练一个模型，并且在训练阶段就对这个模型进行攻击。我们将训练的数据都增加一些噪声从而产生一些攻击的数据，这些被攻击后的图像用 $\tilde{x}$ 来表示。我们将训练数据里面的每一张图片都拿出来进行攻击，攻击完以后再把这些被攻击过后的图片标上正确的标签 $\hat{y}$，这样就可以将我们就产生了新的数据集，我们用 X' 来表示，这个数据集就会让机器产生错误。接着，同时使用 X' 和原始的数据集 X 进行训练，这样就可以训练出一个比较强鲁棒性的模型了。

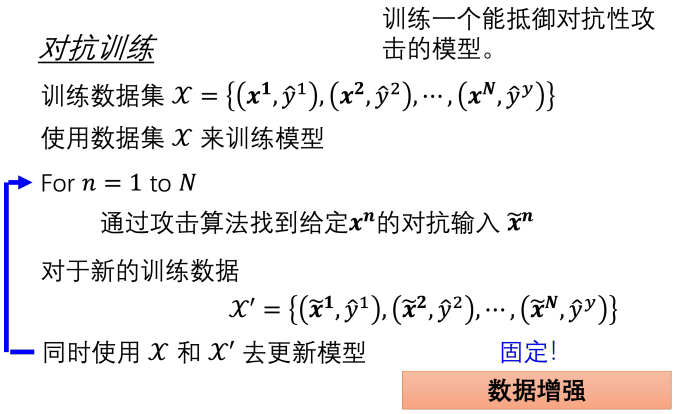


图 12.22　防御方式中的主动防御方法

所以这整个对抗训练的流程就是，我们先训练好一个模型，然后看看这个模型有没有什么漏洞，把漏洞找出来，并且填好漏洞，这样循环往复最后就可以训练出一个比较强鲁棒性的模型了。这种方法可以被视为一种数据增强的方法，因为我们产生了更多的图片 $\tilde{x}$，把这些新图片加到原始的训练数据里面就等于做了数据增强。所以有的研究者也会把对抗训练当做一个单纯数据增强的方法，就算没有人要攻击我们的模型，我们还可以用这样的方法产生更多的数据，并用于训练。这样可以让我们原始的模型，泛化性能更优。

当时对抗训练也有一个非常大的缺点，就是它不见得挡得住新的攻击方式。具体来说，假设我们今天在找 $\tilde{x}$ 的时候用的是算法一二三四，然后接下来有人在实际攻击的时候使用算法五去攻击我们的模型，这样就可能会成功地规避目前的对抗训练技术。所以如果今天实际上攻击我们模型的方法，并没有在对抗训练的时候被考虑过，那对抗训练也不见得能够挡住新的攻击算法。另外，对抗训练它需要非常大的运算资源。因为本来在训练普通模型的时候，训练完模型有输出的结果就结束了。但是对于对抗训练，首先我们要花时间找出这些 $\tilde{x}$，我们原始数据库中有几张图片，我们就要找出多少张新的 $\tilde{x}$。比如说我们有 100 万张图片，我们就要找 100 万个 $\tilde{x}$，光做这件事情就已经很花时间了。

所以总结起来，相对来说攻击是比较容易成功，黑盒攻击也是有可能成功的，防御的难度就会大一些。所以在这一章中，我们先介绍了一些攻击的方式，比如白盒攻击和黑盒攻击，本章介绍了几种经典的防御方法。目前攻击和防御的方法仍然不断地在演化，所以在国际会议中会不断有新的攻击和防御的方法被提出，它们仍然在以对手的方式独自进化中。

第 13 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）**。

13.1 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 13.1 所示，数字识别这种简单的问题，在基准数据集 MNIST [1] 上能做到 99.91% 的正确率[2]。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[3] 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%[4]。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）。

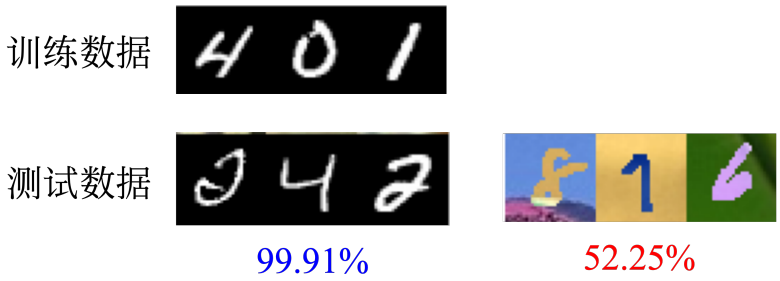


图 13.1 数据分布不同导致的问题

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 13.2 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。

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

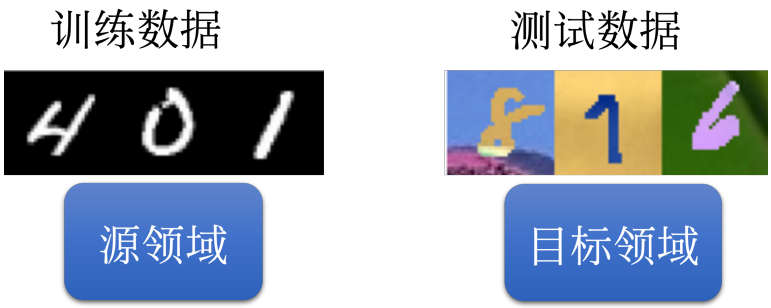


图 13.2 源领域与目标领域

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。

13.2 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 13.3 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。

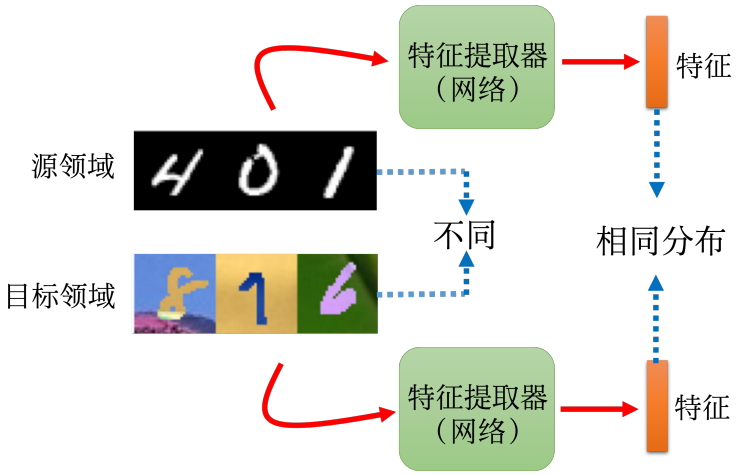


图 13.3　通过特征提取器过滤颜色信息

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，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 13.4 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 13.4 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。

如图 13.5 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 $\boldsymbol{\theta}_p$，领域分类器的参数为 $\boldsymbol{\theta}_d$，特征提取器的参数为 $\boldsymbol{\theta}_f$。源领

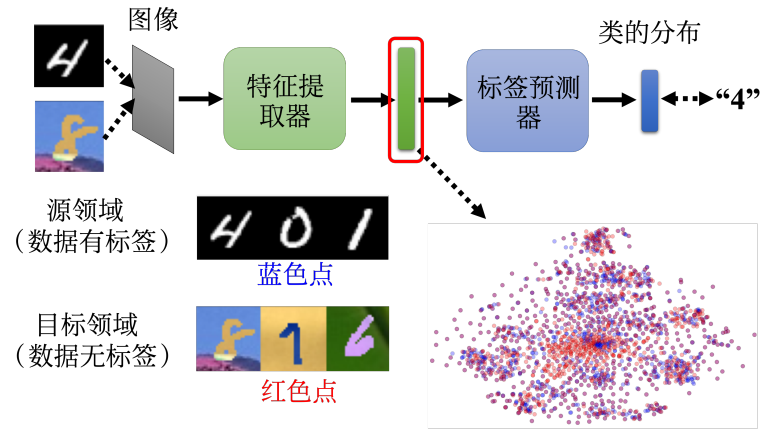


图 13.4　训练特征提取器，让源领域和目标领域的特征无差异[4]

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 $\boldsymbol{\theta}_p$，它可以让 L 越小越好，即

$$\boldsymbol{\theta}_p^* = \min_{\boldsymbol{\theta}_p} L \tag{13.1}$$

我们要去找一个 $\boldsymbol{\theta}_d$，它可以让这个 L_d 越小越好，即

$$\boldsymbol{\theta}_d^* = \min_{\boldsymbol{\theta}_d} L_d \tag{13.2}$$

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，它要去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 $\boldsymbol{\theta}_f$ 让 $L - L_d$ 的值越小越好，即

$$\boldsymbol{\theta}_f^* = \min_{\boldsymbol{\theta}_f} L - L_d \tag{13.3}$$

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，这样做也就可以分开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，其目的都是分开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做了如图 13.6 所示的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 13.1 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。

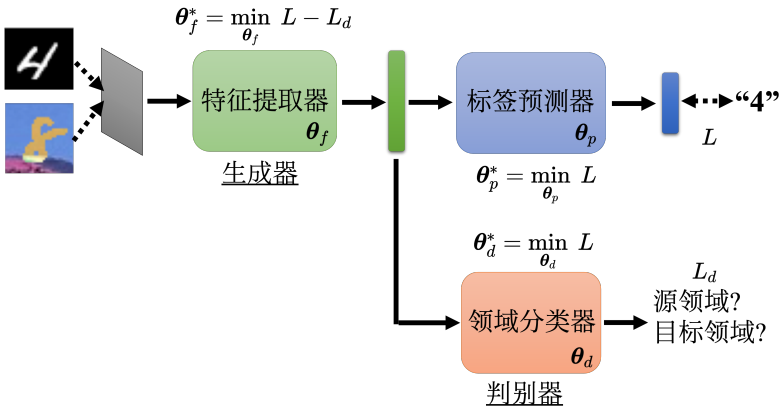


图 13.5 领域对抗训练

图 13.6 领域对抗训练最原始论文使用的任务

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 13.7（a）所示的情况中，红色的点跟蓝色的点是挺对齐在一起的。在图 13.7 （b）所示的情况中，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 13.7 （b）的情况发生，而避免让在图 13.7 （a）的状况发生。

让正方形远离边界（boundary）最简单的做法如图 13.8 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[5]、最大分类器差异（maximum classifier discrepancy）[6] 等方法。这些方法在领域自适应中是不可或缺的。

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

表 13.1 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 MNIST-M	合成数字 SVHN	SVHN MNIST	合成标志 GTSRB
只使用源领域数据训练	57.49%	86.65%	59.19%	74.00%
使用领域对抗训练	81.49%	90.48%	71.07%	88.66%
只使用目标领域数据训练	98.91%	92.44%	99.51%	99.87%

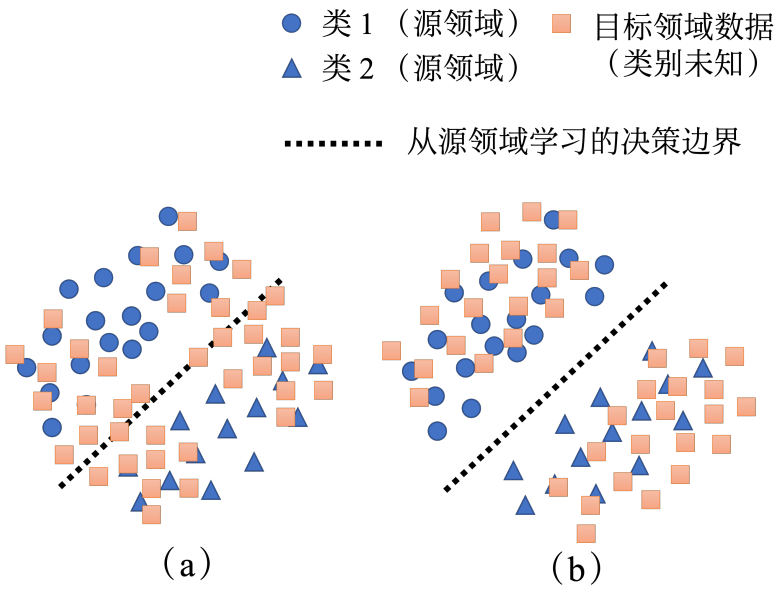


图 13.7　决策边界

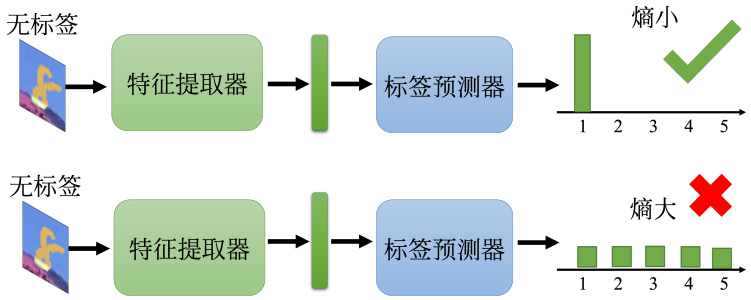


图 13.8　离边界越远越好

面的类别是未知的。如图 13.9 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 13.9（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 13.9（b）中源领域里面的东西比较少，目标领域的东西比较多。图 13.9（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 13.9（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”[7]。

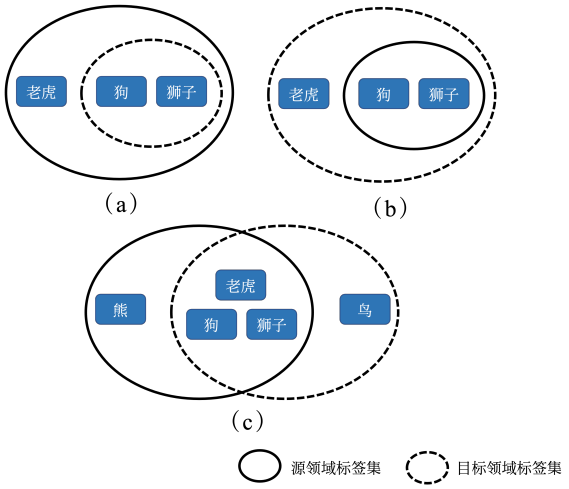


图 13.9 强制完全对齐源领域跟目标领域的问题

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。比如我们觉得 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]。

13.3 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 13.10（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 13.10（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]。

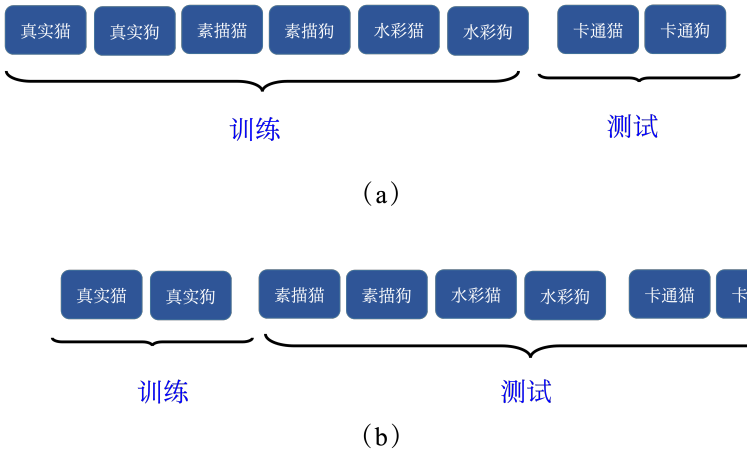


图 13.10 领域泛化示例

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 14 章 强化学习

强化学习（Reinforcement Learning，RL）是一种实现通用人工智能的可能方法。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。图 14.1 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。

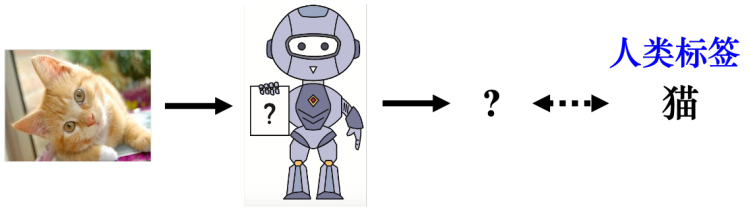


图 14.1 监督学习

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 14.2 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（environment）**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。

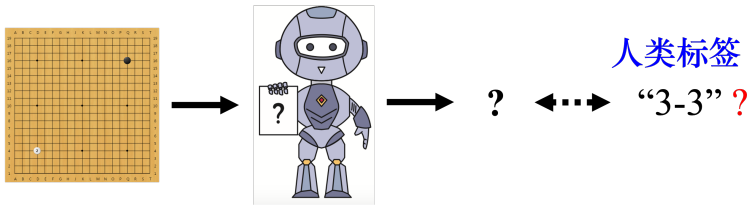


图 14.2 强化学习

如图 14.3 所示，强化学习里面有一个**智能体（agent）**和一个环境，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。

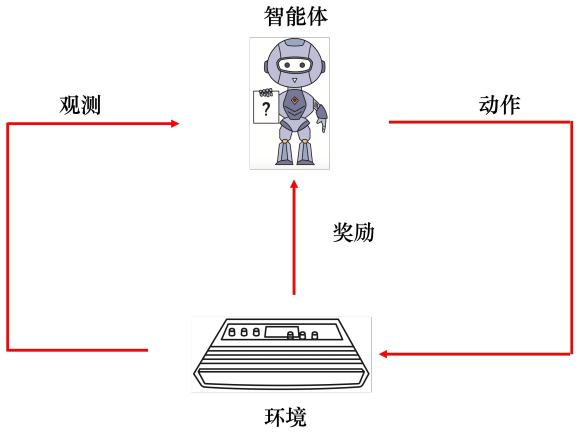


图 14.3 强化学习示意

14.1 强化学习应用

强化学习有很多的应用，比如玩视频游戏、下围棋等等。

14.1.1 玩电子游戏

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 14.4 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。

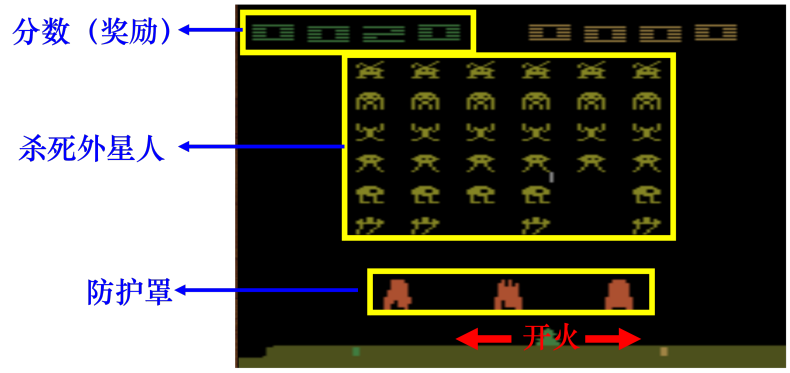


图 14.4 《太空侵略者》游戏

如果用强化学习玩《太空侵略者》，如图 14.5 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，不可能杀掉外星人，所以奖励为 0。智能体体采取一个动作后，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 14.6 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。

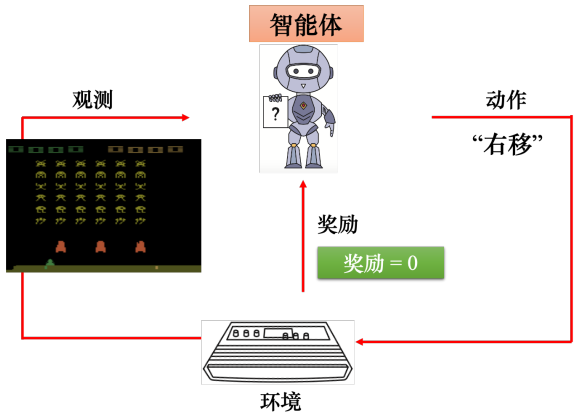


图 14.5　智能体玩《太空侵略者》采取右移的动作

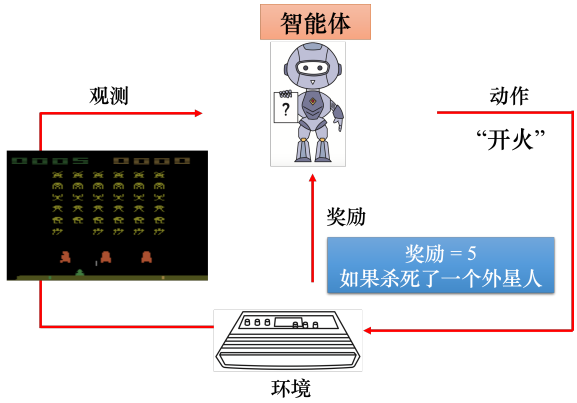
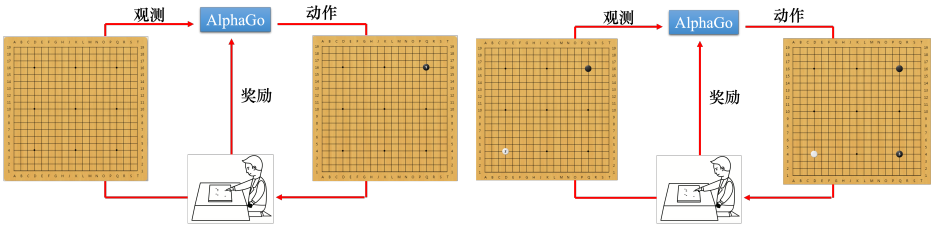


图 14.6　智能体玩《太空侵略者》采取开火的动作

14.1.2　下围棋

如果用强化学习下围棋，如图 14.7(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 19×19 个可能性，每个可能性对应棋盘上的一个位置。

如图 14.7(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 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。



(a) AlphaGo 第一次决策　　(b) AlphaGo 第二次决策

图 14.7　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。

14.2　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。

14.2.1　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 14.8 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 14.8 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。

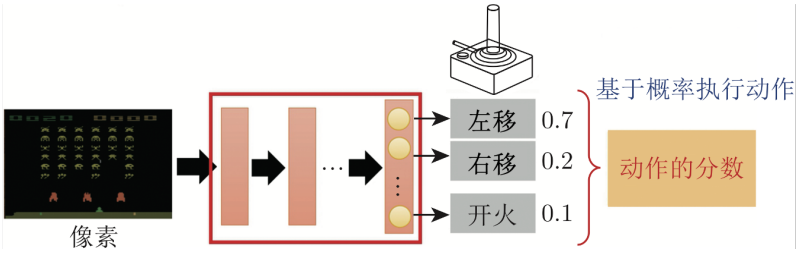


图 14.8　策略网络

14.2.2　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 14.9 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。

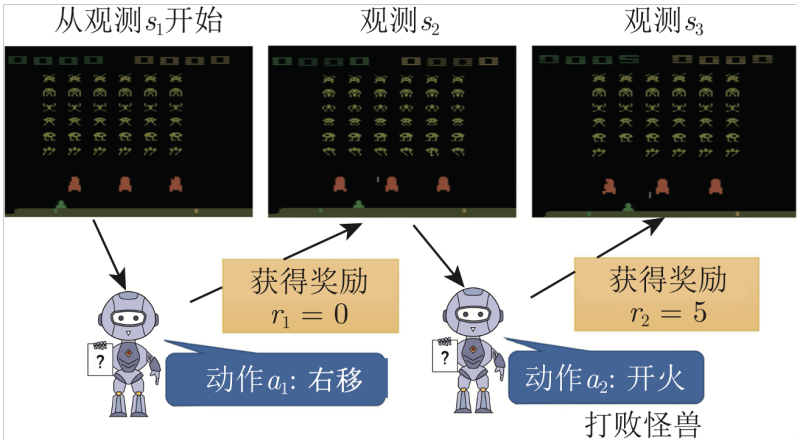


图 14.9　玩视频游戏的例子

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作，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。

14.2.3 第 3 步：优化

图 14.10 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。在一场游戏中，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）** τ，即

$$\tau = \{s_1, a_1, s_2, a_2, \cdots, s_t, a_t\} \tag{14.1}$$

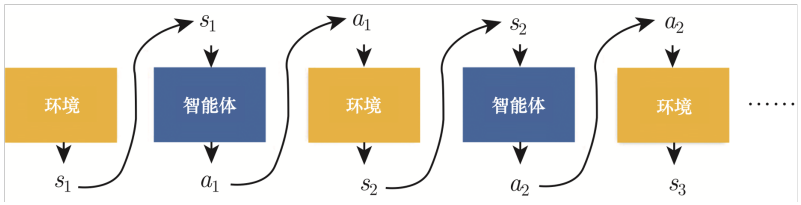


图 14.10 智能体与环境互动

如图 14.11 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 14.11 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

$$R(\tau) = \sum_{t=1}^{T} r_t \tag{14.2}$$

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 14.11 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。

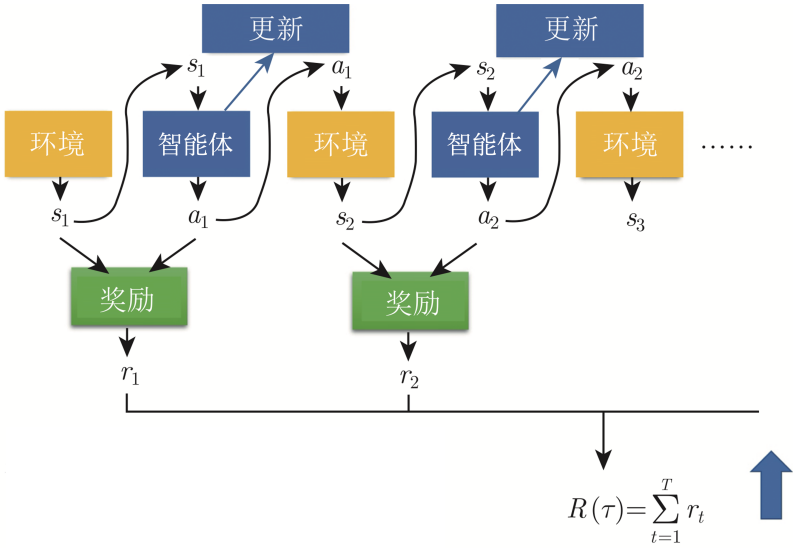


图 14.11　期望的奖励

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 14.12 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。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 $\boldsymbol{\theta}$ 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。

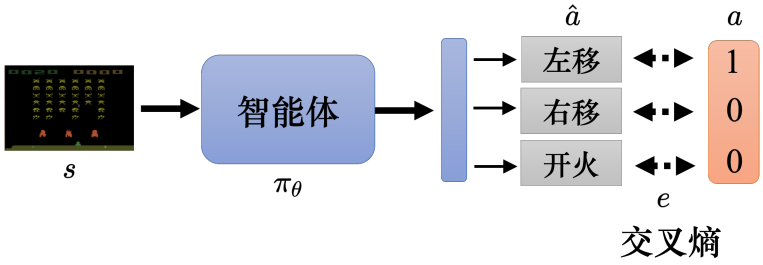


图 14.12　使用交叉熵作为损失

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 14.13 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 $\boldsymbol{\theta}$ 最小化损失，得到 $\boldsymbol{\theta}^*$，如式 (14.3) 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。

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L \tag{14.3}$$

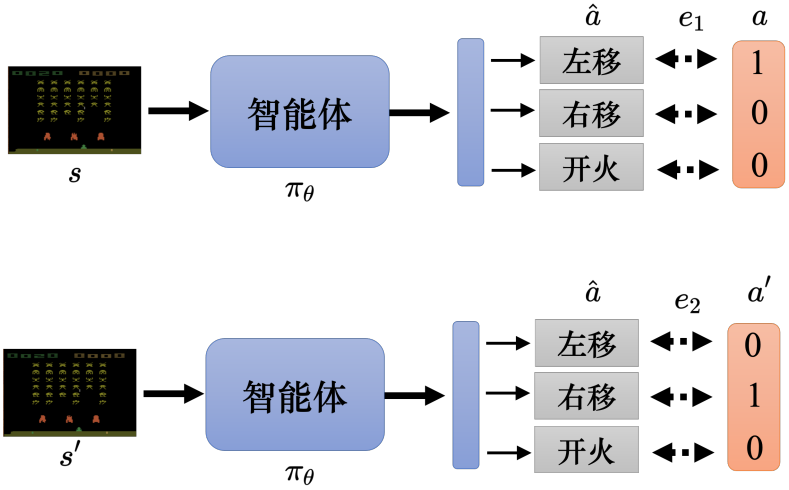


图 14.13　定义合适的损失

如图 14.14 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：

$$L = +e_1 - e_2 + e_3 \cdots - e_{\mathrm{N}} \tag{14.4}$$

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。

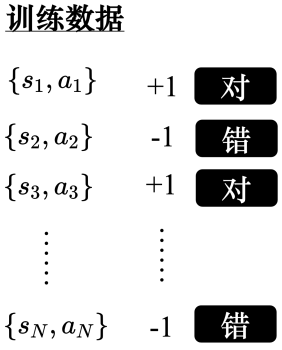


图 14.14　收集训练数据

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 14.15 中第一笔数据

的分数定为 +1.5，第三笔数据的分数为 +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 (14.5) 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 $A_i(i=1,\cdots,n)$，通过 A_i 来控制每一个动作执行的程度。

$$L = \sum \mathrm{A}_n e_n \tag{14.5}$$

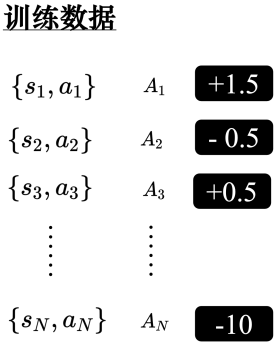


图 14.15 对每个状态-动作对分配不同的分数

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。

14.3 评价动作的标准

本节介绍下评价动作的多种标准。

14.3.1 使用即时奖励作为评价标准

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 14.16 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。

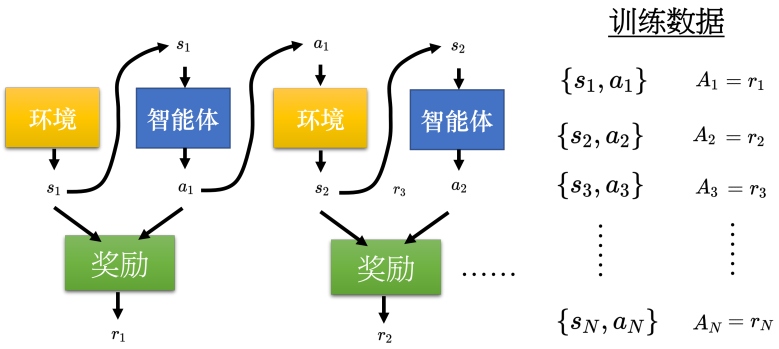


图 14.16 短视的版本

14.3.2 使用累积奖励作为评价标准

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 14.17 所示。G_t 是从时间点 t 开始，r_t 一直加到 r_N，即

$$G_t = \sum_{i=t}^{N} r_i \tag{14.6}$$

比如 G_1、G_2、G_3 的定义为

$$\begin{aligned} G_1 &= r_1 + r_2 + r_3 + \ldots\ldots + r_N \\ G_2 &= r_2 + r_3 + \ldots\ldots + r_N \\ G_3 &= r_3 + \ldots\ldots + r_N \end{aligned} \tag{14.7}$$

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。

14.3.3 使用折扣累积奖励作为评价标准

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 (14.8) 所示，r 前面乘一个折扣因子（discount factor）γ。折扣因子也会设一个小于 1 的值，比如 0.9 或 0.99 之类的。

$$G'_t = \sum_{i=t}^{N} \gamma^{i-t} r_i \tag{14.8}$$

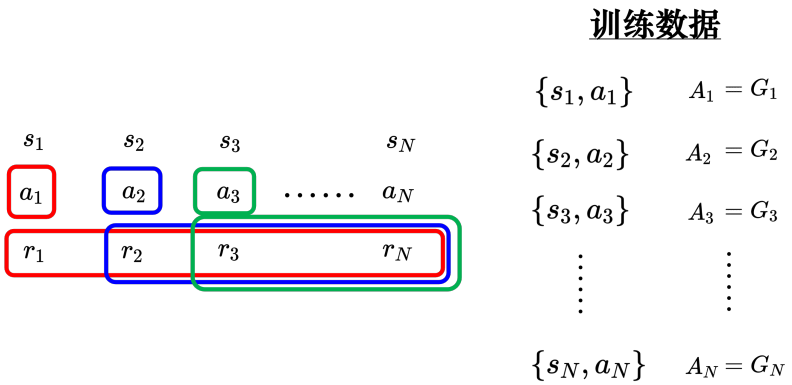


图 14.17 使用累积奖励作为评价标准

图 14.18 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

$$G'_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots\ldots \tag{14.9}$$

距离采取动作 a_1 越远，γ 的次方数越多。r_2 距离 a_1 一步，就乘个 γ，$r3$ 距离 a_1 两步，就乘 γ 平方，一直加到 r_N 的时候，r_N 对 G'_1 几乎没有影响，因为 γ 乘了非常多次，γ 是一个小于 1 的值，即使 γ 设 0.9，0.9^{10} 也很小了。

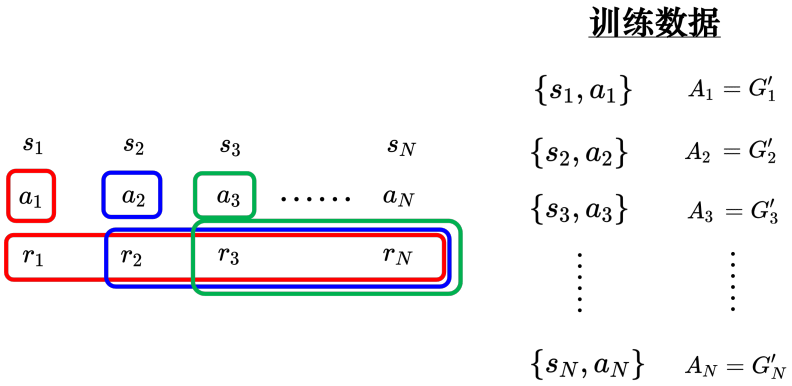


图 14.18 使用折扣奖励作为评价标准

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，γ 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：在游戏等情况里，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。

14.3.4 使用折扣累积奖励减去基线作为评价标准

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 14.19 所示。

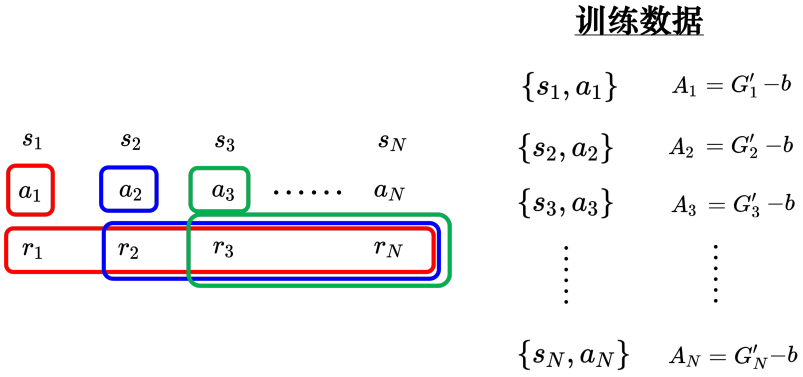


图 14.19 减去基线

策略梯度算法中的评价标准就是 $G' - b$，其详细过程如算法 14.1 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 $\boldsymbol{\theta}_0$。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 η，接着用该梯度更新模型，把 $\boldsymbol{\theta}_{i-1}$ 更新成 $\boldsymbol{\theta}_i$。

算法 14.1 策略梯度

初始化智能体网络参数 $\boldsymbol{\theta}$;
for $i = 1$ **to** T **do**
  使用智能体 $\pi_{\boldsymbol{\theta}_{i-1}}$ 进行交互;
  获取数据 $\{s_1, a_1\}, \{s_2, a_2\}, \cdots, \{s_N, a_N\}$;
  计算 $A_1, A_2, \cdots, A_N$;
  计算损失 L;
  $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_{i-1} - \eta \nabla L$
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 14.20 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花时间的原因。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。

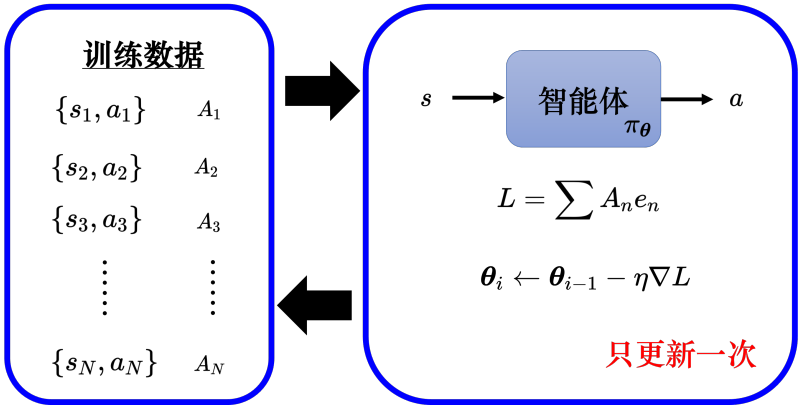


图 14.20 强化学习训练过程

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 14.1 所示，这些数据是由 $\pi_{\boldsymbol{\theta}_{i-1}}$ 所收集出来的，这是 $\pi_{\boldsymbol{\theta}_{i-1}}$ 跟环境互动的结果，这个是 $\pi_{\boldsymbol{\theta}_{i-1}}$ 的经验，这些经验可以拿来更新 $\pi_{\boldsymbol{\theta}_{i-1}}$，可以拿来更新 $\pi_{\boldsymbol{\theta}_{i-1}}$ 的参数，但它不一定适合拿来更新 $\pi_{\boldsymbol{\theta}_i}$ 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，它的好是不一样的。

如图 14.21 所示，假设用 $\pi_{\boldsymbol{\theta}_{i-1}}$ 收集了一堆的数据，这些数据只能用来训练 $\pi_{\boldsymbol{\theta}_{i-1}}$，不能用来训练 $\pi_{\boldsymbol{\theta}_i}$。假设 $\pi_{\boldsymbol{\theta}_{i-1}}$ 和 $\pi_{\boldsymbol{\theta}_i}$ 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 $\pi_{\boldsymbol{\theta}_i}$ 跟 $\pi_{\boldsymbol{\theta}_{i-1}}$ 收集的数据根本就不一样。使用 $\pi_{\boldsymbol{\theta}_{i-1}}$ 的收集数据来评估 $\pi_{\boldsymbol{\theta}_i}$ 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。

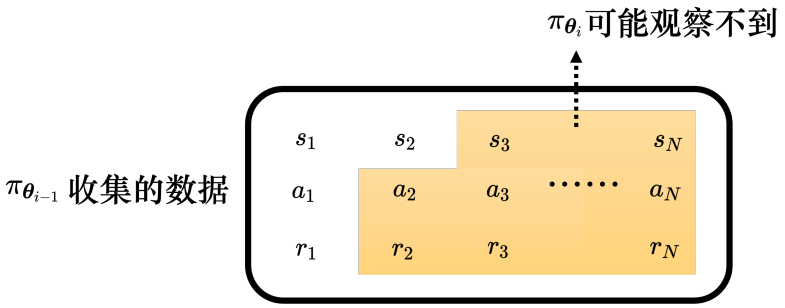


图 14.21　不同智能体收集的数据不能共用

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。

14.3.5　Actor-Critic

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。版本 3.5 跟 Critic 及其训练方法相关。假设有一个智能体的参数为 $\pi_{\boldsymbol{\theta}}$，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 funciton）**，可以用 $V_{\pi_{\boldsymbol{\theta}}}(s)$ 来表示。上标 $\pi_{\boldsymbol{\theta}}$ 代表这个 V 观测的 Actor 的策略为 $\pi_{\boldsymbol{\theta}}$。如图 14.22(a) 所示，其输入是 $\boldsymbol{s}$，$V_{\pi_{\boldsymbol{\theta}}}$ 就是一个函数，输出是一个标量 $V_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s})$。价值函数 $V_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s})$ 表示智能体 $\pi_{\boldsymbol{\theta}}$ 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 14.22(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 14.22(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。

(a) 状态价值函数 $V_{\pi_{\boldsymbol{\theta}}}$　(b) $V_{\pi_{\boldsymbol{\theta}}}(s)$ 较大　(c) $V_{\pi_{\boldsymbol{\theta}}}(s)$ 较小

图 14.22　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 $\boldsymbol{s}_a$，累积奖励为 G'_a；看到游戏画面 $\boldsymbol{s}_b$，累积奖励为 G'_b。如果使用蒙特卡洛（Monte Carlo，MC）的方法，如图 14.23 所示，输入 $\boldsymbol{s}_a$ 给价值函数 $V_{\pi_{\boldsymbol{\theta}}}$，其输出 $V_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s}_a)$ 跟 G'_a 越接近越好。输入 $\boldsymbol{s}_b$ 给价值函数 $V_{\pi_{\boldsymbol{\theta}}}$，其输出 $V_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{s}_b)$ 跟 G'_b 越接近越好。

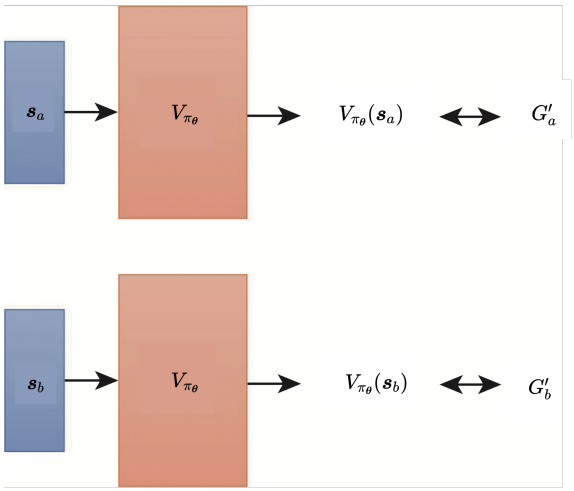


图 14.23　基于蒙特卡洛的方法

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 $V_{\pi_{\boldsymbol{\theta}}}(s)$，就可以更新 $V_{\pi_{\boldsymbol{\theta}}}(s)$ 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，$V_{\pi_{\boldsymbol{\theta}}}(s_t)$ 跟 $V_{\pi_{\boldsymbol{\theta}}}(s_{t+1})$ 之间的关系如式 (14.10) 所示（为了简化，没有取期望值）。

$$\begin{aligned} V_{\pi_{\boldsymbol{\theta}}}\left(s_t\right) =& r_t+\gamma r_{t+1}+\gamma^2 r_{t+2}+\cdots \\ V_{\pi_{\boldsymbol{\theta}}}\left(s_{t+1}\right) =& r_{t+1}+\gamma r_{t+2}+\cdots \\ V_{\pi_{\boldsymbol{\theta}}}\left(s_t\right) =& \gamma V_{\pi_{\boldsymbol{\theta}}}\left(s_{t+1}\right)+r_t \end{aligned} \tag{14.10}$$

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 $V_{\pi_{\boldsymbol{\theta}}}(s_t)$，$s_{t+1}$ 代到价值函数得到 $V_{\pi_{\boldsymbol{\theta}}}(s_{t+1})$。虽然 $V_{\pi_{\boldsymbol{\theta}}}(s_t)$ 和 $V_{\pi_{\boldsymbol{\theta}}}(s_{t+1})$ 具体的值是未知的，但把两者应该满足如下关系

$$V_{\pi_{\boldsymbol{\theta}}}\left(s_t\right)-\gamma V_{\pi_{\boldsymbol{\theta}}}\left(s_{t+1}\right) \leftrightarrow r_t \tag{14.11}$$

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

同样的 $\pi_{\boldsymbol{\theta}}$ 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 14.24 是某个智能体跟环境互动，玩了某个游戏八次的记录。

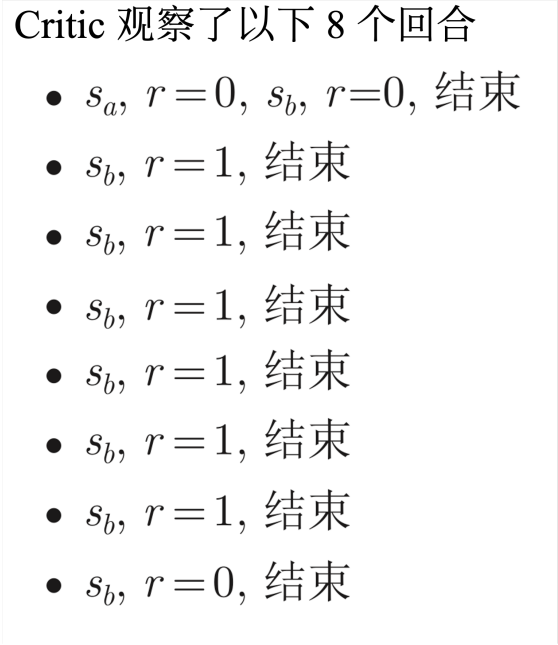


图 14.24　时序差分方法与蒙特卡洛方法的差别[1]

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 14.24 中的例子里面是无法处理的。因为在图 14.24 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。$V_{\pi_{\boldsymbol{\theta}}}(s_b)$ 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

$$\frac{6\times 1+2\times 0}{8}=\frac{6}{8}=\frac{3}{4} \tag{14.12}$$

$V_{\pi_{\boldsymbol{\theta}}}(s_a)$ 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 $V_{\pi_{\boldsymbol{\theta}}}(s_a)=0$。但如果用时序差分算出来的，因为 $V_{\pi_{\boldsymbol{\theta}}}(s_a)$ 跟 $V_{\pi_{\boldsymbol{\theta}}}(s_b)$ 中间有关系

$$V_{\pi_{\boldsymbol{\theta}}}\left(s_a\right)=V_{\pi_{\boldsymbol{\theta}}}\left(s_b\right)+r \tag{14.13}$$

因此

$$\begin{aligned} V_{\pi_{\boldsymbol{\theta}}}\left(s_a\right) &= V_{\pi_{\boldsymbol{\theta}}}\left(s_b\right)+r \\ &= \frac{3}{4}+0=\frac{3}{4} \end{aligned} \tag{14.14}$$

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 $V_{\pi_{\boldsymbol{\theta}}}(s_a)$ 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 14.25 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 $A_1 = G'_1 - b$ 。

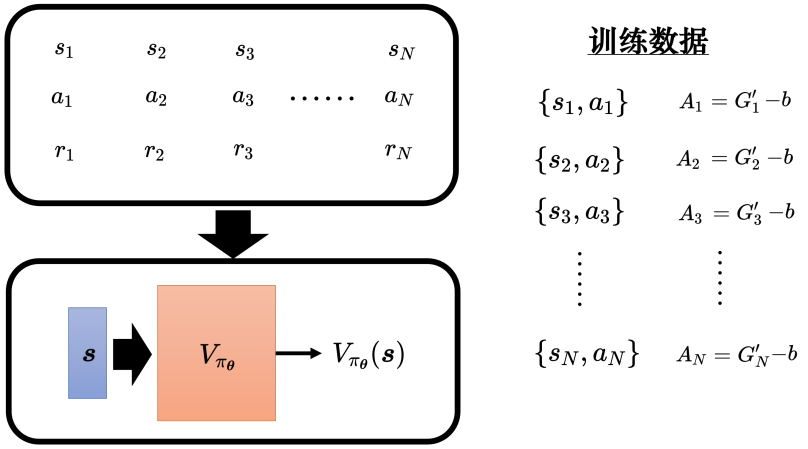


图 14.25　使用折扣累积奖励减去基线作为评价标准

学习出 Critic V_{π_θ} 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 14.26 所示。

训练数据

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

图 14.26　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 14.27 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）。

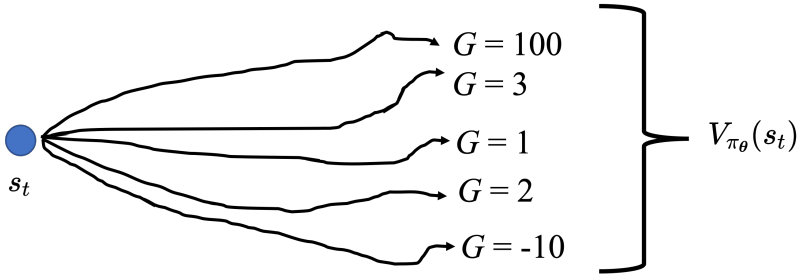


图 14.27　看到 s_t 后不同的累积奖励

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 $A_t < 0$。

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，即优势 Actor-Critic。

14.3.6 优势 Actor-Critic

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 14.28 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+$V_{\pi_\theta}(s_{t+1})$ -$V_{\pi_\theta}(s_t)$。

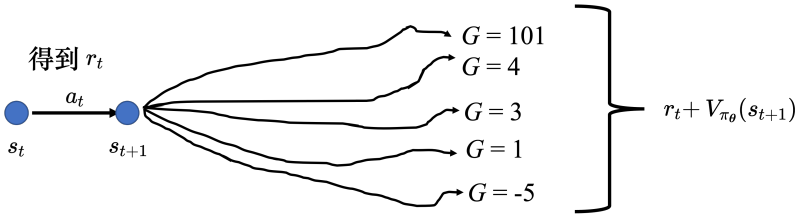


图 14.28 优势 Actor-Critic

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 14.29 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。

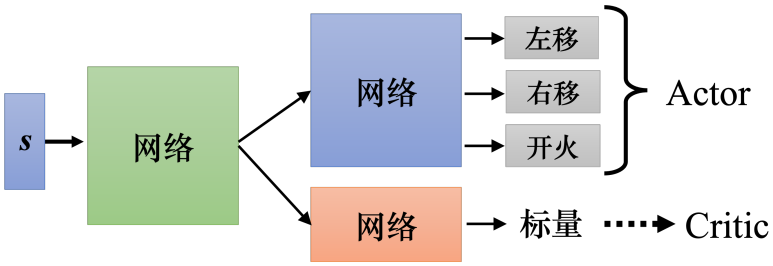


图 14.29 Actor-Critic 训练技巧

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”[2]，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）。

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》[3]，此处不再赘述。此外，视觉强化学习（Visual Reinforcement Learning，VRL）是强化学习中非常有潜力的强化学习方向，与之前传统的基于状态的强化

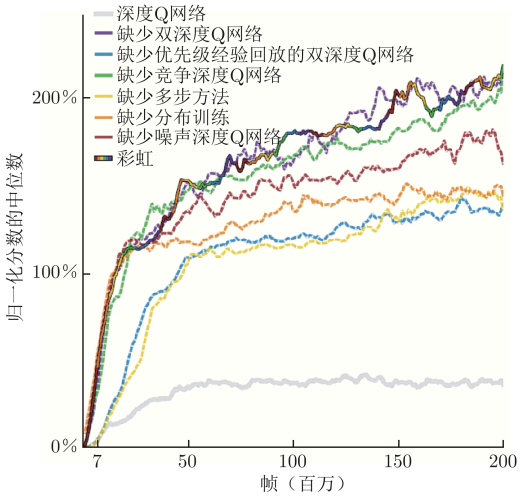


图 14.30 彩虹方法

学习方法不同，其根据图片来直接学习控制策略，感兴趣的同学可阅读相关的论文，可参考 Awesome Visual RL 论文清单：https://github.com/qiwang067/awesome-visual-rl。

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.

第 15 章 元学习

15.1 元学习的概念

我们这一章介绍**元学习（meta learning）**。元学习从字面的意思就是“学习”的“学习”，也就是学习如何学习。大部分的深度学习就是在不断的调整超参数，或者在决定网络架构，改变学习率等等。实际上没有什么好方法来调这些超参，今天工业界最常拿来解决调整超参数的方法是买很多张 GPU，然后一次训练多个模型，有的训练不起来、训练效果比较差的话就输入掉，最后只看那些可以训练的比较好的模型会得到什么样的性能。所以在业界做实验的时候往往就是一次开几张 GPU，这些 GPU 跑多组不同的超参数，看看哪一组超参数可以得到最好的结果。但是在学术界我们通常没有那么多张 GPU，通常需要凭着经验和直觉定义可能效果比较好的超参数，然后看看这些超参数会不会得到好的结果。但是这样的方法往往会花费很多时间，因为需要不断的去调整这些超参数。所以我们就会想办法让机器自己去调整这些超参数，机器自己学习一个最优的模型和网络架构，然后得到好的结果。元学习就这样诞生了。

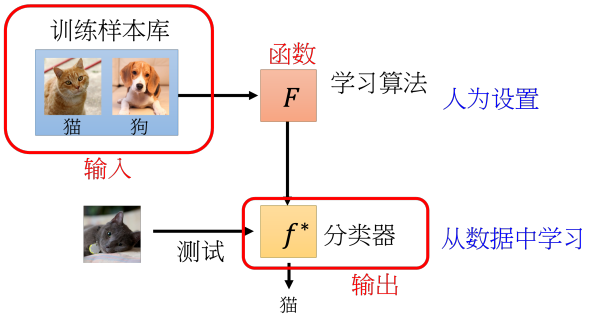


图 15.1 元学习的概念

我们接下来分析元学习的本质以及其主要的三个步骤。首先元学习算法，把它简化来看其实就是一个函数。这个函数我们用 F 来表示它，不同于普通的机器学习算法的输入一张图片，元学习的函数 F 是一个数据集，这个数据集里面有很多的训练数据。我们把训练数据集输入到这个函数 F 里面，它会输出训练完的结果。假设我们要训练的是一个分类器，那这个函数就是输入训练数据，输出就是分类器。有了这个分类以后，我们就可以把测试数据输入，它输出来的结果是我们要的分类结果。所以一个学习算法它是一个函数，我们用 F 来表示它，而它的输入就是训练数据，输出是另外一个函数，我们用 f 来表示它。这个 f 的输入是一张图片，输出是分类的结果，整个元学习的框架如图 15.1 所示。所以元学习的目标就是要找到一个 F，这个 F 可以让 f 的损失越小越好。那这个 F 函数是人为设定的，或者说是我们提前设置的。我们其实也可以直接学习这个 F 函数，对应于我们在机器学习里面介绍的三个步骤，在元学习里，其实我们要找的也是一个函数，只是这个函数跟机器学习一般要找函数不一样。在元学习里我们要找的函数是一个学习算法。我们下面分别类比机器学习的三个步骤来介绍元学习中的三个步骤，来寻找学习函数。

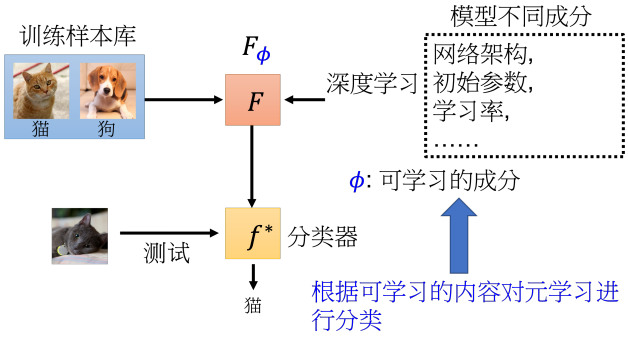


图 15.2　元学习的步骤一：学习算法

15.2　元学习的三个步骤

首先第一个步骤（如图 15.2 所示）是我们的学习算法里要有一些要被学的东西，就像在机器学习里面神经元的权重和偏置是要被学出来的一样。在元学习里面，我们通常会考虑要让机器自己学习网络的架构，让机器自己学习初始化的参数，让机器自己决定学习率等等，我们期待它们是可以通过学习算法被学出来的，而不是像机器学习一样我们人为设定的。我们把这些在学习算法里面想要它自学的东西统称为 ϕ，在机器学习里面我们是用来 θ 代表一个函数里面我们要学的东西。接下来，我们都将学习算法写为 F_ϕ，代表这个学习算法里面有些是未知的参数。对于不同的元学习的方法，它会想办法去学模型不同的成分，当去学不同模型成分的时候，我们就有了不同的元学习的方法。

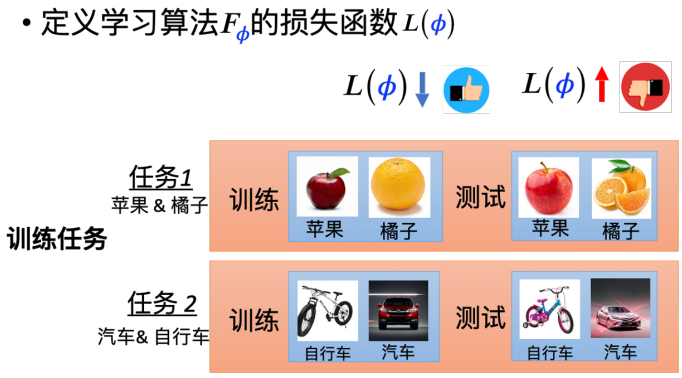


图 15.3　元学习的步骤二：定义损失函数

接下来，第二步（如图 15.3 所示）是设定一个损失函数，损失函数在元学习里是要决定学习算法的好坏。$L(\phi)$ 代表现在用 ϕ 作为参数的算法的性能。如果 $L(\phi)$ 的值很小就代表它是一个好的学习算法，反之它就是一个不好的学习算法。我们需要如何决定这个 L 损失函数呢？在机器学习里面，损失函数来自于训练数据，那在元学习里面，我们收集的是训练任务。举例来说，假设今天想要训练一个二分类的分类器，来分辨苹果和橘子（任务一），以及分别脚踏车和车（任务二），以上每一个任务里面我们都会有分训练数据和测试数据。

接下来我们就要分析在元学习中的损失函数 L 应该如何定义。我们评价一个学习算法的好坏，是看其在某一个任务里面使用训练数据学习得到的算法的好坏。比如，任务一是分辨苹果和橘子，我们就把任务一里的训练数据拿出来输入给这个学习算法，从而学出一个分类。我

们使用 $f_{\theta^{1*}}$ 来代表这个是任务一的分类，分辨苹果和橘子。如果这个分类是好的，那就代表我们的 L 的规则是好的，反之如果这个分类是不好的，就代表说这个学习算法是不好的。对于不好的学习算法（测试数据中表现不好的），我们就会给它比较大的损失 $L(\phi)$。

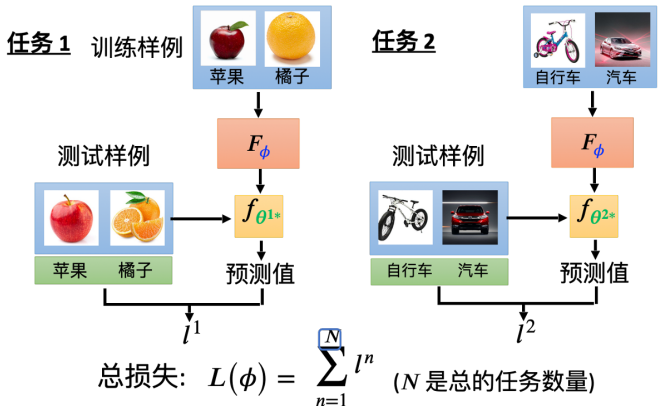


图 15.4　元学习的步骤二中的多任务分类

到目前为止，我们都只考虑了一个任务，那在元学习中我们通常不会只考虑一个任务，也就是我们不会只用苹果和橘子的分类来看一个二分类学习算法的好坏。我们还希望拿别的二元分类的任务来测试它，比如说区分自行车和汽车的训练数据（如图 15.4 所示），输入给这个学习算法，让它进行分类。像这两个学习算法是一样的，但是因为输入的训练数据不一样，所以产生的分类也不一样。θ^{1*} 代表的是这个学习算法在任务一（分类苹果和橘子）上面学习得到的参数，θ^{2*} 代表的是这个学习算法在任务二（分类自行车和汽车）上面学习得到的参数。与任务一相同，任务二它自身也有一些测试的数据，我们一样把它的测试数据输入给 $f_{\theta^{2*}}$，然后看看得到的正确率如何，就可以计算出其在任务二中的表现。所以我们现在知道这个学习算法 $L(\phi)$ 在任务一和任务二上的表现以后，就可以把任务一和任务二上的表现把它加起来，得到这个整个学习算法的损失。当然如果扩展到 N 个任务的话，整个损失就是一个 N 个任务的损失，我们把这个损失写成 $L(\phi)$，这个损失就是我们的元学习的损失。其代表了现在这个学习算法学习所有问题的表现有多好。

大家应该已经关注到了一件事情，元学习中在每一个任务计算损失的时候，我们是用测试数据来进行计算。而在一般的机器学习里面，所谓的损失其实是用训练数据来进行计算的。这个问题是因为我们的训练单位是任务，所以可以用训练的任务里面的测试数据，训练的任务里面的测试数据是可以在元学习的训练的过程中被使用到的。我们将元学习的演算法介绍完以后，再会把元学习和机器学习再做一次比较，届时会更加清楚。

最后，元学习的第三个任务就是要找一个学习算法，即找一个 ϕ 让损失越小越好。这件事怎么做呢？我们已经写出了损失函数 $L(\phi)$，是 N 个任务的损失的总和。我们现在要找一个 ϕ 使得 $L(\phi)$ 最小，我们将这个 ϕ 定义为 ϕ^*。解这个优化问题的过程有很多，比如之前我们介绍过的梯度下降；如果没有办法计算梯度，也可以用强化学习的方法来解这个优化问题，或者使用进化算法来解这个优化问题。总之，我们就可以让机器自己找出来的一个学习算法 F_{ϕ^*}，这个学习算法 F_{ϕ^*} 就是我们的元学习算法。

我们再来回顾一下元学习的三个过程，如图 15.5 所示。首先收集一批训练数据，这些训练数据是由很多个任务组成的，并且每一个任务都有训练数据和测试数据。根据这些训练数据通过我们刚才讲的三个步骤，可以得到一个学习算法 F_{ϕ^*}。接下来我们就可以用这个学习

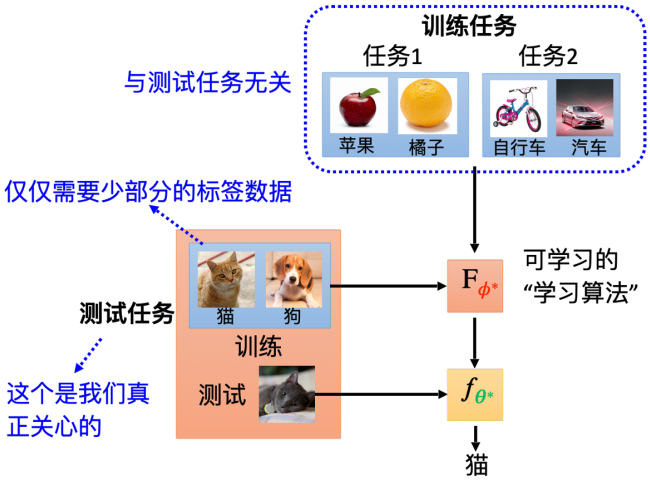


图 15.5　元学习的完整框架

算法 F_{ϕ^*} 来进行测试。假设训练的时候，训练任务是教机器分苹果和橘子以及自行车和汽车，测试的时候是要分猫和狗，那每一个任务里面都有训练数据，也都有测试数据。我们就需要把测试任务里面的训练数据学出一个分类，然后再把这个分类用在测试任务里面的测试数据上。其中我们真正关心的是测试任务里面的测试数据，因为这个测试数据是我们真正要分类的东西。

很多人会觉得小样本学习和元学习非常的像，所以我们也简单区分一下元学习和小样本学习。简单来说，小样本学习指的是期待机器只看几个样例，比如每个类别都只给他三张图片，它就可以学会做分类。而我们想要达到小样本学习中的算法通常就是用元学习得到的学习算法，这就是两者的关系。

15.3　元学习与机器学习

这一小节比较一下机器学习和元学习的差异。首先来看一下机器学习和元学习的目标，如图 15.6 所示。机器学习的目标是要找一个函数 f，这个函数可以是一个分类器，把几百张图片输入进去，分类器就告诉我们分类的结果。元学习一样是找一个函数，但它要找的是一个学习算法 F_{ϕ^*}，这个学习算法可以接受训练数据，然后输出一个分类器 f。这个 F_{ϕ^*} 将训练数据作为输入，它直接输出训练的分类结果 f，这个 f 就是我们想要的分类器。

从训练数据角度分析，在机器学习里面，我们是拿某一个任务里面的训练数据进行训练，而在元学习中我们是拿训练的任务来进行训练。这个很容易搞混，所以在文献中，我们会把任务里面的训练数据叫做支持（support），把测试数据叫做查询（query）。在元学习里面，我们是拿查询来进行训练，而在机器学习里面，我们是拿支持来进行训练。

那在机器学习里面，我们需要手动设置一个学习算法，而在元学习里面，我们是有一系列的训练任务，所以我们也将元学习中的学习算法部分的学习称为跨任务学习。而对应的机器学习中的学习称为单一任务学习，因为我们是在一个任务里面进行学习。

我们再看一下两者的完整框架，如图 15.7 所示。在机器学习中，完整的框架就是把训练数据拿去产生一个分类器，接着再把测试数据输入到这个分类器里面得出分类的结果。而在元学习中，我们是有一些训练的任务，把这些训练的任务拿来产生一个学出来的学习算法叫做 F_{ϕ^*}。对于接下来的测试任务，测试任务里面有支持数据和查询数据，我们再把这个测试任

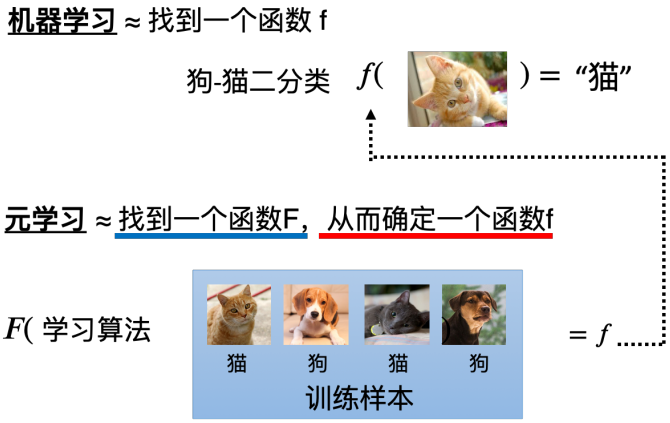


图 15.6　元学习和机器学习的目标

务里面的训练数据输入到学习出来的学习算法里面，得到一个分类器后，再把测试数据输入进去，得到分类的结果。我们把元学习里面的这个测试叫做跨任务测试，因为它不是一般的测试。一般的机器学习，我们的这个测试叫做单一任务测试，因为我们是在一个任务里面进行测试。在元学习里面，我们要测试的不是一个分类表现的好坏，而是一个学习算法的表现的好坏，所以在元学习里面为跨任务的测试。那有时候我们也在一些论文中会看到整个流程中一次单一任务的训练和一次跨任务的测试，我们把这个流程叫做一个回合。所以在元学习里面，我们是在一个回合里面进行训练和测试，而在机器学习里面，我们是在一个任务里面进行训练和测试。

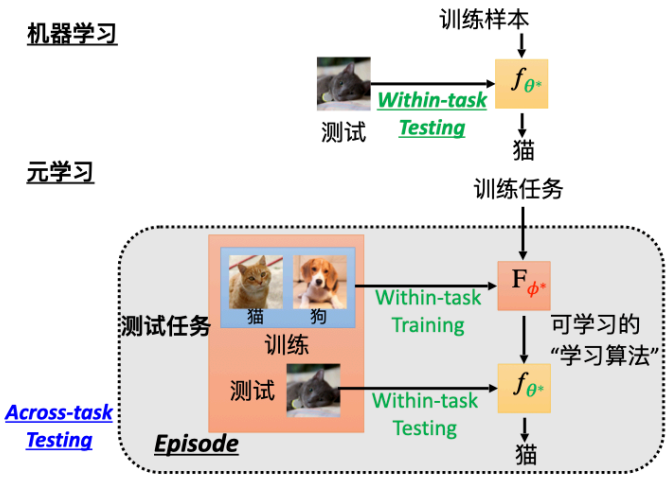


图 15.7　元学习和机器学习的框架对比

对于损失，在机器学习中我们使用 $L(\theta) = \sum_{k=1}^{K} e_k$ 表示损失函数，其中 e_k 表示第 k 个训练样本的损失，其中的加和为所有训练数据在一个任务中的损失总和。在元学习里面，我们使用 $L(\phi) = \sum_{n=1}^{N} l^n$ 表示损失函数，其中的 l^n 表示第 n 个测试样本中的损失，其中的加和为在所有任务中的损失总和。

对于训练的过程两者也有一些差异，元学习的训练需要算 l^n，也就是计算每一个小任务的损失函数，在这个过程中，我们需要做一次单一任务的训练 + 测试，也就是一个回合。现

在假设我们的优化算法中，要找一个 ϕ 让 $L(\phi)$ 最小，那做这件事情的时候，我们需要算这个 L 很多次，也就是跨任务的训练包含了很多次的单一任务的训练 + 测试。这个是非常复杂的且耗时的。所以在文献中，学习如何去初始化整个过程的工作中，往往把跨任务训练叫做外循环，把单一任务训练叫做内循环。这是因为在跨任务训练里要跑好几次单一任务训练，所以跨任务训练是外循环，单一任务训练是内循环。不过，外循环和内循环这些称呼通常只有在学习初始化的工作中才会有，其他时候通常也不会这样叫。

刚才介绍的都是元学习和机器学习的差别，他们其实也有很多的共同之处，事实上很多在机器学习那边学到的知识和基本概念都可以直接搬到元学习来。举例来说，在机器学习上面你会害怕训练数据上可能会有过拟合的问题，那在元学习里面也有可能会有过拟合的问题，比如机器学习到了一个学习算法，这个学习算法在训练任务上做得很好，面对一个新的测试的任务反而会做得不好，所以元学习也有可能有过拟合的问题。如果遇到过拟合问题应该怎么办呢？我们类比一下机器学习，在机器学习里面，最直观的方法就是收集更多的训练数据，所以在元学习里面也可以做一样的事——收集更多的训练任务。也就是如果训练的任务越多，就代表训练的数据样本越多，那学习算法就越有机会可以泛化并用到新的任务上面。

另外，我们在机器学习上会做数据增强，也就是在训练的时候，我们会把训练数据做一些变化，比如说把图片做一些旋转、平移、缩放等等，这样可以让训练数据更多。在元学习里面我们同样也可以做数据增强，也可以想一些方法来增加训练任务，比如说我们可以把训练任务做一些变化，比如说把训练任务的类别做一些变化，或者把训练任务的数据做一些变化等等。此外，我们在做元学习的时候还是要做优化，我们还是要想办法去找一个 ϕ 让 $l(\phi)$ 越小越好。假设我们最小化 $l(\phi)$ 的方法是梯度下降法，那我们在做梯度下降法的时候，我们还是要去调学习率，只不过与机器学习不同，我们需要调整的参数是可学习的学习算法的参数。有人可能会问，既然都要调整参数，那我何必还要用元学习，直接讲每一个机器学习的问题调整参数不就可以了吗？其实并不是，因为在元学习里面，我们只需要把学习算法的参数调整好，其就可以一劳永逸的使用到其他的任务当中，而不需要每一个任务都去调整参数。这样的话，我们就可以节省很多的时间，也可以让我们的学习算法更加的有效率。

那既然说到要调参数就遇到另一个问题了，在机器学习中我们不仅仅有训练样本和测试样本，同时还有验证集的样本，用验证集样本中的表现来选择你的模型。所以元学习中也应该要有用于验证的任务，也就是说在元学习中，我们应该要有训练任务、验证任务和测试任务。其中验证任务来决定训练学习算法的时候的一些超参数，然后才跑在测试的任务中。

15.4 元学习的实例算法

前面我们已经讲完了元学习的基本概念，接下来我们就要讲一些元学习的实例算法。在这里我们会介绍两个算法，一个是**模型诊断元学习(model-agnostic meta-learning，MAML)**，另一个是 Reptile。这两个算法都是在 2017 年提出来的，而且都是基于梯度下降法进行优化的。那我们最常用的学习算法是梯度下降，在梯度下降中，我们要有一个网络架构，同时初始化一下这个网络的参数 θ^0。我们也有一些训练数据，他们是从训练数据里面采样一个批次出来，然后用这个批次来计算梯度，并用这个梯度来更新参数，从 θ^0 到 $\theta^{'}$，接着再重新计算一次梯度，再更新参数，就反复这样下去，直到次数够多，输出一个满意的 θ^* 出来。

初始化的参数 θ^0 可以训练的，一般 θ^0 是随机初始化的，也就是从某一个固定的分布里面采样出来的。同时 θ^0 对结果往往有一定程度的影响，好的初始化参数不好的初始化参数可以天差地远，那我们能不能够透过一些训练的任务，来找出一个对训练特别有帮助的初始化

参数呢？这个方法是模型诊断元学习。

MAML 的基本思路是，算法要最大化模型对超参数的敏感性。也就是说，要让学习到的超参数让模型的损失函数因为样本的微小变化而有较大的优化。因此，模型的超参数设置应该能够让损失函数变化的速度最快，即损失函数此时有最大的梯度。因此，损失函数被定义为每一个任务下该模型的损失函数的梯度的和。剩下要做的，就是根据这个定义的损失函数，用梯度下降法进行求解。在训练的过程中算法会求取以下两次梯度。第一次求梯度：针对每个任务，计算损失函数的梯度，进行梯度下降；第二次求梯度：对梯度下降后的参数求和，再求梯度，进行梯度下降。当然，MAML 有另外的变形就叫做 Reptile，翻译过来叫爬虫，大家可以自行了解。需要补充的是，虽然在 MAML 中，我们要去学习初始化参数的过程，但是在其中我们也是有很多超参数需要自己决定的。

这里做一个联想，我们在介绍自监督学习的时候，我们也有提到好的初始化参数的重要性。在自监督学习中，我们就是有很多的没有标记的数据，可以用一些代理的任务去训练它，比如说在 Bert 里面就是用填空题来训练模型，在图像上也可以做自监督学习。比如把图片的其中一块盖起来，让机器预测被盖起来的一块是什么东西，机器就可以从中学到一些特征，然后再把这些特征用在其他的任务上。当然，现在在做图像的自监督学习的时候，可能这个掩码的方法以及所谓的填空的方法不是最常用的，目前比较流行用对比学习的方法。总之，在自监督学习中我们会先拿一大些的数据去做预训练，那预训练的结果我们也说它是好的初始化参数，然后再把这些好的初始化参数用在测试的任务上。

那这 MAML 和自监督学习有什么不同呢？其实它们的目的都是一样的，都是要找到好的初始化参数，但是它们的方法不一样。自监督学习是用一大些的数据去做预训练，而 MAML 是用一大些的任务去做预训练。另外过去在自监督学习还没有兴起的时候，也有一些方法是用一大些的任务去做预训练，这个方法叫做多任务学习。具体来讲，我们一样有好几个任务的数据，并且把这些好几个任务的数据通通放在一起，然后接下来我们同样可以找到一个好的初始化参数，然后再把这个好的初始化参数用在测试的任务上，这就是多任务学习。现在我们在做有关 MAML 研究的时候，通常会把这种多任务学习的训练方法来当做元学习的基线。因为这两个方法他们用的数据都是一样的，一边只是我们会把不同的任务分开，另外一边把所有的任务的数据倒在一起。

其实 MAML 很像是域适应或者迁移学习，也就是我们在某些任务上面学到的东西可以被被迁移到另外一个域，我们可以说他们是基于分类问题的域适应或者迁移学习。所以我们在研读文献的时候其实也不用太拘泥于这些词汇，我们要真正要在意的是这些词汇背后所代表的含义是什么。

我们下面解释一下 MAML 的优势。首先有两个假设，第一个假设是 MAML 找到的初始参数是一个很厉害的初始参数。它可以让我们的例如梯度下降这种学习算法快速找到每一个任务的参数。另外一个假设是这个初始化的参数它本来就和每一个任务上最终好的结果已经非常接近了，所以我们直接应用很少几次的梯度下降就可以轻易的找到好的结果，这个也是使得 MAML 有效的关键。当然 MAML 也有一些变形，比如 ANIL（almost no inner loop）、First order MAML（FOMAML）以及 Reptile 等等，这里我们不做扩展。

除了可以学习初始化的参数外，MAML 还可以学习优化器，如图 15.8 所示。我们在更新参数的时候，需要决定比如说学习率、动量等等的参数。像学习率这种超参数进行自动更新的方法在很早以前就有了，NIPS2016 年就有一篇文章，叫做“Learning to learn by gradient descent by gradient descent”。在这篇文章里面呢，作者直接学习了优化器，一般我们的优化

器都是人为规定的（比如 ADAM 等等），而这个文章中的参数是根据训练的任务自动学出来的。

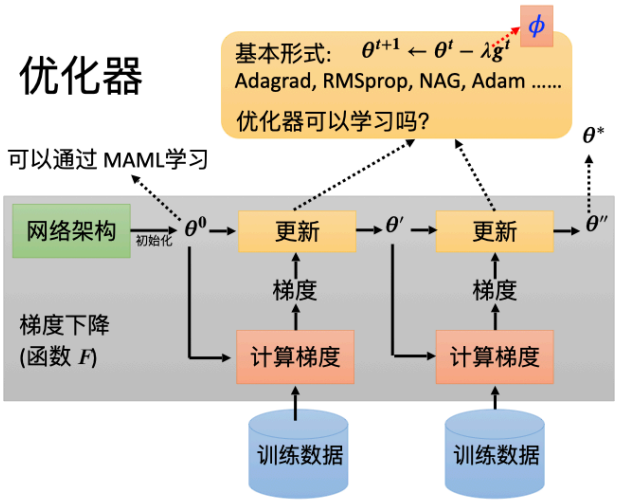


图 15.8　MAML 中可学习的优化器

当然我们还可以训练网络架构，这部分的研究被称为**神经网络架构搜索 (Neural Architecture Search，NAS)**。如果在元学习中我们学习的是网络的架构，讲网络架构当作 ϕ 的话，那我们就是在做 NAS。在 NAS 里面，ϕ 是网络架构，我们要找一个 ϕ 去最小化 $L(\phi)$。这其中我们做 $L(\phi)$ 的微分就有可能出问题了。当我们遇到优化问题时并且没办法上微分的时候，强化学习也许是一个解决方法。具体做法是，我们可以把 ϕ 想成是一个智能体的参数，这个智能体的输出是 NAS 中相关的超参数。举例来说，现在第一层的滤波器它的长是多少，宽是多少，步长是多少，数目是多少等等。我们智能体的输出就是 NAS 中相关的参数，然后接下来我们要训练智能体，让它去最大化一个回报，即 $-L(\phi)$，让他去最大化这个回报，就等于最小化 $L(\phi)$，也就是我们要找的最优的 ϕ。

下面是从一篇文章中截取的一个 NAS 的例子，以此为例介绍 NAS 的过程，如图 15.9 所示。具体来讲，我们有一个智能体是 RNN 架构。这个 RNN 架构每次会输出一个网络架构有关的参数，比如它会输出滤波器的高是多少，再输出过滤器的宽是多少，接着再输出步长是多少等等。第一层第二层输出完了以后，接下来输出 n+1 层，再输出 n+2 层，以此类推。有了这些参数以后，就根据它们设计一个网络以后就去训练这个网络，这个过程其实就是之前我们介绍的单一任务训练。然后接下来就去做强化学习，我们可以把这一个网络他在测试数据上面的准确率当做回报来训练智能体，目标是最大化回报，这个过程其实就是跨任务训练。那除了强化学习以外，其实用演变算法也是可以的。本质上其实就是硬要把网络架构改一下，让它变得可以微分。其中有一个经典的做法叫做可微分架构搜索，这个方法的本质就是想办法让问题变的是可以微分，从而可以直接用梯度下降来最小化这个损失函数。

除了网络架构可以学习外，其实数据处理部分也有可能可以学习。我们在训练网络的时候，通常要做数据增强。那在元学习中我们可以让机器自动进行数据增强。另一个角度，我们在训练的时候，有时候会需要给不同样本不同的权重。具体操作的话就会有不同的策略，比如有的策略就是如果有一些样例距离分类边界线特别近，那说明其很难被分类，这样类似的样例也许就要给它们比较大的权重，这样网络就会聚焦在这些比较难分别的样例中，希望它们

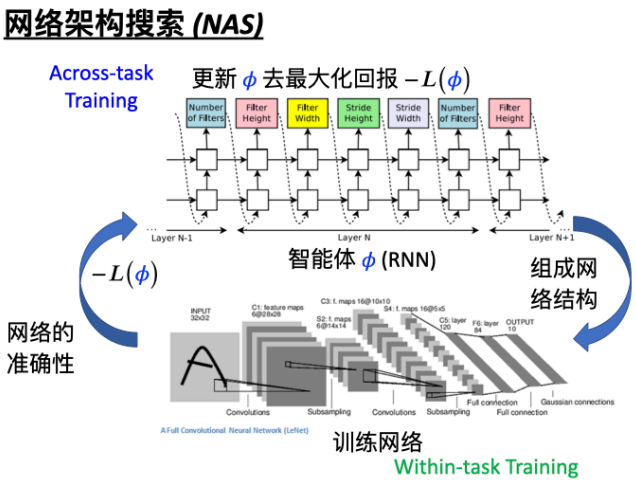


图 15.9 NAS 的实例

可以被学得比较好。但是也有文献有相反的结论，比如比较有干扰噪声的这些样本应该给它比较小的权重，这些样例如果比较接近分类边界线，可能代表它比较有噪声干扰，代表它可能标签本身就标错了，或者分类是不合理的等等，也许就应该给这些样本比较小的权重。那元学习中如何决定这个采样权重的策略呢？我们可以用学习的方式把采样策略直接学出来，然后根据采样数据的特性自动决定采样数据的权重如何设计。

到目前为止，我们看到的这些方法都是基于梯度下降再去做改进的，我们有没有可能完全舍弃掉梯度下降呢？比如，我们有没有可能直接学习一个网络，这个网络的参数 ϕ，网络直接训练数据作为输入直接输出训练好的结果？如果真的有这样一个网络，它可以将训练数据作为输入输出训练好的网络的参数，那我们就可以让机器发明了新的学习算法。这个是有可能的，并且已经有了一些论文。不过到目前为止，我们还是把训练跟测试分成两个阶段，我们有一个学习算法使用训练数据进行训练，然后输出训练好的结果，并把训练好的结果用在测试数据上。我们想看看有没有可能更进一步，直接把整个回合，也就是一次训练加一次测试合并在一起。有一些工作，它就是直接把训练数据和测试数据当做网络的输入，网络输入完训练数据以后，机器也许学出了一个学习算法，也许找出一组参数等等，再它输入训练数据以后再给他测试数据，它就可以直接输出这些测试数据的答案。这个时候我们不再有训练和测试的分界，在一个回合里面不再分训练和测试，而是直接用个网络把训练和测试这件事情一次搞定。目前这种系列的方法叫做“learning to compare”，它又叫做基于度量的方法，这一系列的做法就可以看作是训练和测试没有分界，一个网络直接把训练数据和测试数据都读进去，而直接输出测试数据的结果。

15.5 元学习的应用

本章最后，我们再简单分享一些元学习的应用。现在在做元学习的时候，我们最常拿来测试元学习技术的任务叫做少样本图像分类，简单来讲就是每一个任务都只有几张图片，每一个类别只有几张图片。比如我们使用图 15.10 的案例为例说明。现在分类的任务是分为三个类别，每个类别都只有两张图片作为输入，我们希望通过这样一点点的数据就可以训练出一个模型。也就是给这个模型一张新的图片，它可以知道这张图片属于哪一个类别。在做这种

少样本图像分类的时候，我们会常看到一个名词叫做 N 类别 K 样例下的分类任务，这个名词是什么意思呢？N 类别 K 样例的分类任务，它的意思就是在每一个任务里面，我们有 N 个类别，而每一个类别我们只有 K 个样例。举例来说，在图 15.10 这个例子里面我们有三个类别，每一个类别只有两个样例，那它就是 3 类别 2 样例分类。在元学习里，如果我们要教机器能够做 N 类别 K 样例分类，那意味着说我们需要准备很多的 N 类别 K 样例下的分类任务当做训练的任务，这样机器才能够学到 N 类别 K 样例的算法。

图 15.10　少样本的案例分析

那要怎么去找一系列的 N 类别 K 样例下的任务呢？在文章中最常见的一种做法是使用 Omniglot 当做基准，Omniglot 是一个手写的数据集，它有 1623 个不同的字符，每一个字符有 20 个样例。那有这些字符以后呢，我们就可以去制造 N 类别 K 样例下的分类。比如我们从 Omniglot 里面选出 20 个字符，然后每一个字符就只取一个样例，这样就得到一个 20 类别 1 样例的分类任务。如果我们把这个任务当做训练数据，那我们就可以让机器学习到 20 类别 1 样例的分类算法。如果我们把这个任务当做测试数据，那我们就可以测试机器在 20 类别 1 样例的分类任务上的表现。同理，我们可以制造出 20 类别 5 样例的分类任务，这个任务里面每一个类别都有 5 个样例，然后我们可以把这个任务当做训练数据，让机器学习到 20 类别 5 样例的分类算法。

在使用 Omniglot 的时候，我们会把字符分成两半，一半是拿来制造训练任务的字符，另外一半是拿来制造测试任务的字符。如果我们要去制造一个 N 类别 K 样例的任务，那么就是从这些训练任务的字符里面先随机采样 N 个字符，然后这 N 个字符每个字符再去采样 K 个样例，集合起来就得到一个训练的任务。对于测试的任务，就从这些测试的字符里面拿出 N 个字符，然后每个字符采样 K 个样例，你就得到一个 N 类别 K 样例下的测试任务。这样我们就可以把 Omniglot 当做一个基准，然后在这个基准上面测试不同的元学习算法。

总之，元学习不是只能用非常简单的任务，今天在学界已经开始把元学习推向更复杂的任务，我们也一直希望未来元学习这个技术能够真的用在现实的应用上，可以走得多远好。

第 16 章 终身学习

终身学习本质上基于人类对于人工智能的想象，期待人工智能可以像人类一样能够持续不断地学习。

16.1 灾难性遗忘

如图 16.1 所示，我们先训练机器做任务一语音识别，再教它做任务二图像识别，接着再教它做第三个任务翻译，这样一来它就一起学会了这三个任务。我们不断去教会机器学习新的技能，等它学会成百上千个技能之后，它就会变得越来越厉害，以至于人类无法企及，这就是我们所说的**终身学习（LifeLong Learning，LLL）**。

> 终身学习也称为**持续学习（continous learning）、无止尽学习（never-ending learning）、增量学习（incremental learning）**。

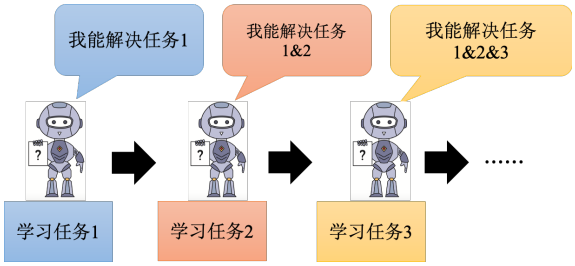


图 16.1 自监督学习框架

读者可能会疑惑，这个终身学习的目标过于远大，并且难以实现，其意义又在哪里呢？其实在真实的应用场景中，终身学习也是派得上用场的。举例来说，如图 16.2 所示，假设我们首先通过收集一些数据然后训练得到模型，模型上线之后它会收到来自使用者的反馈并且得到新的训练数据。这个时候我们希望能够形成一个循环，即模型上线之后我们得到新的数据，然后讲新的数据用于更新我们的模型，模型更新之后又可以收到新的反馈和数据，对应地再次更新我们的模型，如此循环往复下去，最终我们的模型会越来越厉害。我们可以把过去的任务看成是旧的数据，把新的数据即来自于反馈的数据，这种情景也可以看作是终身学习的问题。

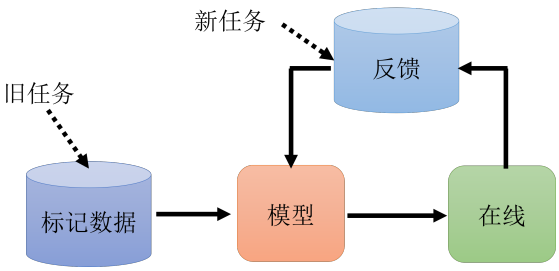


图 16.2 自监督学习框架

终身学习有什么样的难点呢？看上去不断更新它的数据和对应网络的参数就能实现终身

学习，但实际上没有那么容易。我们来看一个例子，如图 16.3 所示，假设我们现在有两个任务，第一个任务是手写数字识别，给一张包含噪声的图片，机器要识别出该图片中的数字“0”，第二个任务也是手写数字识别，只是给的图片噪声比较少，相对来说更容易。当然有读者可能会认为这不算两个任务，最多算同一个任务的不同域，这样说也没错。可能读者想象中的终身学习应该是先学语音识别然后图像识别之类的，但其实现在的终身学习都还没有做到那种程度。目前关于终身学习的论文中所说的不同任务大概指的就是本示例中同一个任务的不同域这种级别，只是我们把它们当作不同的任务来对待。但即使是非常类似的任务，在做终身学习的过程中也会遇到一些问题，具体哪些问题我们接下来一一说明。

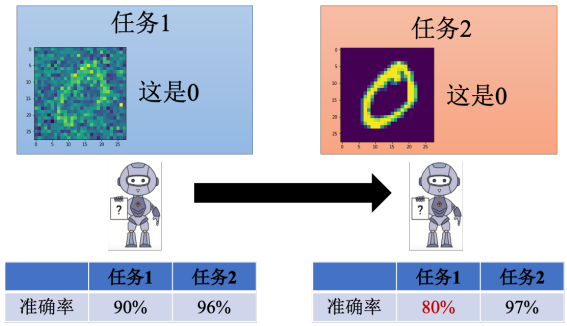


图 16.3　自监督学习框架

我们先训一个比较简单的网络来做第一个手写数字识别任务，然后再做任务二，任务一上的准确率是 90%，此时就算没有训过任务二，在任务二上也已经有了 96% 的准确率，可以说迁移得非常好，说明能够解决任务一，相应地也能解决任务二。那么接下来我们用同一个的模型，即在任务一中训练好的模型再去训练任务二，结果我们发现在任务二上的准确率变得更好了（97%）。但是比较糟糕的事情是此时机器已经忘了怎么去做任务一了，即在任务一上的准确率从 90% 降到了 80%。有读者可能会想是不是网络设置的太过简单然后导致出现这样的现象，但实际上我们把任务一和任务二的数据放在一起让这个网络同时去学的时候，会发现机器是能够同时学好这两个任务的，如图 16.4 所示。

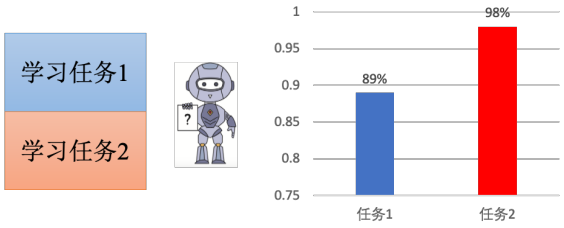


图 16.4　自监督学习框架

接下来举一个自然语言处理方面的例子，即 QA 的任务，QA 指的是给一篇文档，模型训练之后能够基于这个文档回答一些问题。当然为了简化，我们会将一个更简单的实例，即“bAbi”，它是一类非常早期和基础的研究任务，总共有二十个子任务。

如图 16.5 所示，其中第五个任务是给了三个句子，即 Mary 把蛋糕给了 Fred，Fred 把蛋糕给了 Bill，Bill 给了牛奶给 Jeff，最后问“谁给了蛋糕给 Fred？”类似于这种简单问题，其他任务包括第十五个任务也同理。我们的目标是让 AI 依次去学这二十个任务，要么让一个模型

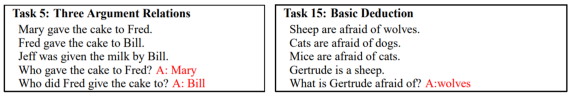


图 16.5　bAbi 任务示例

同时学这二十个任务，要么用二十个模型，每个模型分别学一个任务，这里我们主要讲的是前者。最后我们实验的结果如图 16.6 所示。

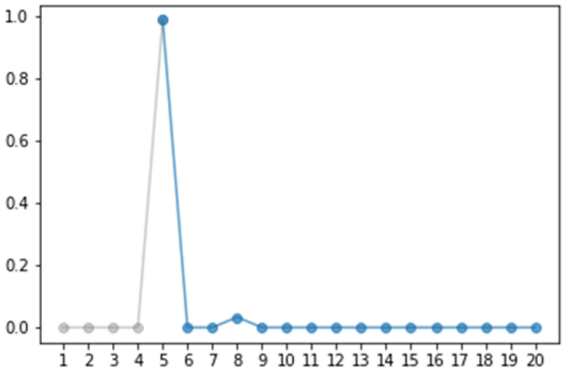


图 16.6　任务 5 的准确率（依次学习 20 个任务）

可以看到，一开始我们没有学过任务五，所以准确率是零。当开始学第五个任务之后，准确率达到百分百，然而开始学下一个任务的时候，准确率就开始暴跌，即学习新的任务之后一下子就完全忘了前面的任务。读者可能以为模型本身就没有学习那么多任务的能力，但其实不是的。如图 16.7 所示，当模型同时学二十个任务的时候，我们会发现机器是有潜力能够学习多个任务的，当然这里第十九个任务可能有点难，模型的准确率非常低。

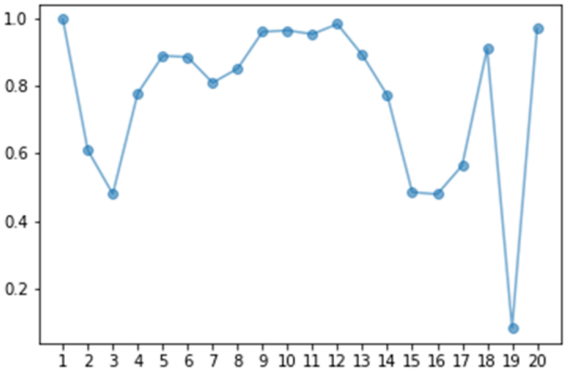


图 16.7　所有任务的准确率（同时学习 20 个任务）

当模型依次学习多个任务的时候，它好像一个上下两端都接有水龙头的池子，新的任务从上面水龙头进来，旧的东西就从下面水龙头那里冲掉了，永远学不会多个技能，这种情况称为**灾难性遗忘（catastrophic forgetting）**。我们知道即使是人类也可能会有遗忘的时候，而这里在遗忘前面我们加上了灾难性这个形容词，意在强调模型的这个遗忘不是一般的遗忘，而是特别严重程度的遗忘。

讲到这里我们接下来就需要知道怎么去解决这个灾难性遗忘的问题了。在我们讨论具体

的技术之前，也许读者会有这样一个问题。刚才我们的例子有提到模型是能够同时学多个人任务的，这种学习方式称为**多任务学习（multitask learning）**，既然有这个多任务学习的例子，那为什么还要去做终身学习的事情呢？但其实这种多任务学习会有这样一个问题，加上我们需要学习的任务不再是简简单单的二十个，而是一千个，那么我们在学第一千个任务的时候，按照这个逻辑还得把前面九百九十九个任务的数据放在一起训练，这样需要的时间是比较久的。

就好比假设一个人要学一门新的课程，那他就必须要把这辈子上过的所有课程都学过一遍才有办法学习新的任务，这样其实是比较低效的。而且随着要学习的任务越来越多，所需要的训练时间也会越来越长。但如果我们能够解决终身学习的技术，那么其实就能够高效地学习多种新任务了。当然这种多任务学习也并非没有意义，我们通常把多任务学习的效果当成终身学习的上界，即我们在做终身学习的时候往往会先跑一个多任务学习的结果，来帮助我们参考终身学习的上限在哪里。

这里可能读者又会有一个问题，即为什么我们不能每个任务都分别用一个模型呢？其实这样做会有一些问题，首先这样做可能会产生很多个模型，这样对机器存储也是一个挑战。另外不同任务之间可能是有共通的，从任务 A 学到的数据也可能在学习任务 B 的时候有所参考。类似的还有一个迁移学习的概念，虽然终身学习和迁移学习都是让模型同时学习多个任务，但它们的关注点是不一样的。在迁移学习中，我们在意的是模型在前一个任务学习到的东西能不能对第二个任务有所帮助，只在乎新的任务做得如何。而终身学习更注重的是在解决完第二个任务之后能不能再回头解决第一个任务。

16.2 终身学习评估方法

现在我们可以先看看怎么评判终身学习技术做不做得好的一些标准。在做终身学习之前，得先有一系列任务让模型去学习，其实通常都是比较简单的任务。如图 16.8 所示，任务一就是常规的手写数字识别，任务二其实还是手写数字识别，但只是把每一个数字用某一种特定的规则打乱，称之为排列，这种算是比较难的，还有更简单的就是把数字右转一次。

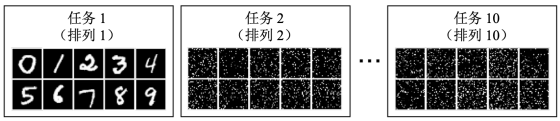


图 16.8 终身学习手写数字识别示例

具体的评估方式如图 16.9 所示，首先有一排任务，并且有一个随机初始化的参数，用在这 T 个任务上，得到对应的准确率。然后让模型先学第一个任务，然后在所有任务上分别测一次准确率，即 $R_{1,1}, ..., R_{1,T}$，依此类推。直到学完所有的任务，得到一个准确率的表格，来评估终身学习的结果。

具体的最终准确率公式表示为

$$\frac{1}{T}\sum_{i=1}^{T} R_{T,i} \tag{16.1}$$

另一种评估方法叫做反向迁移，即

$$\frac{1}{T-1}\sum_{i=1}^{T-1} R_{T,i} - R_{T,i} \tag{16.2}$$

		测试			
		任务1	任务2	……	任务T
随机初始化		$R_{0,1}$	$R_{0,2}$		$R_{0,T}$
训练之后	Task 1	$R_{1,1}$	$R_{1,2}$		$R_{1,T}$
	Task 2	$R_{2,1}$	$R_{2,2}$		$R_{2,T}$
	⋮				
	Task T-1	$R_{T-1,1}$	$R_{T-1,2}$		$R_{T-1,T}$
	Task T	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

图 16.9　评估终身学习的准确率表格

16.3　终身学习的主要解法

解决终身学习问题，即主要解决灾难性遗忘的问题，通常来讲学术界有几种主要的研究思路。我们首先将第一种主要的解法，即选择性的突触可塑性（selective synaptic plasticity）。顾名思义，就是只让神经网络中的某些神经元之间的连接具有可塑性，其余的必须被固化，这类方法又叫做基于正则的方法。我们可以先回顾一下会什么发生灾难性遗忘这种现象，例如现在有任务一和任务二，并且处于简化考虑，假设训练的模型只有两个参数 θ_1 和 θ_2。如图 16.10 所示，分别表示了模型在不同任务上的损失函数，如果颜色越偏蓝色，代表损失越小，反之越大。

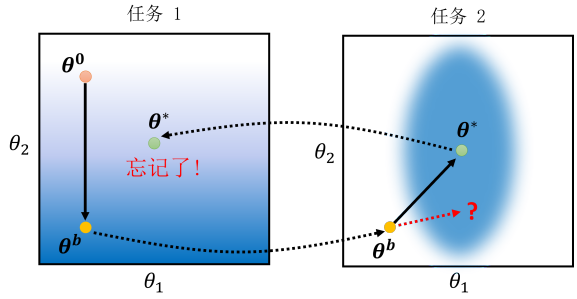


图 16.10　灾难性遗忘示意图

我们先让模型训练任务一，比如给一个随机化的初始参数 θ^0 ，用梯度下降的方法更新足够多次数的参数后得到 θ^b 。接下来继续训练任务二，我们先把 θ^b 复制过来放到任务二上，那么由于任务二的损失截面是不同的，即蓝色的区域不同，通过多次迭代我们可能会把参数更新到 θ^* 的位置。然后我们把在任务二上训练好的参数 θ^* 拿回到任务一上使用时，会发现并没有办法得到好的结果。因为这个 θ^* 只是在任务二上表现较好，而不见得会在任务一上有较低的损失和好的表现，这就是灾难性遗忘产生的原因。

那怎么解决这个问题呢？其实对于一个任务而言，要实现较低的损失是可能有很多种不同的参数组合的，比如在任务二中可能所有椭圆内的参数都能有好的表现，而对于任务一则是偏下方的位置都能实现较低的损失。那么如果在跑完任务一之后在训练任务二时，我们的参数不是像右上角移动形成 θ^*，而是只往右移动，让最终得到的参数同时处于任务一和任务二较低的损失区域，这种情况下是有可能不会产生灾难性遗忘的。对于这种方式，基本的思想就是每一个参数对我们过去学过的任务的重要性程度是不同的，因此在学习新的任务时，我

们尽量不要动那些对过去任务很重要的参数，而是去学一些其他的对新任务比较重要的参数。假设 θ^b 是前一个任务学出来的参数，在选择性的突触可塑性这一解法中会给每一个参数 θ_i^b 赋予一个系数 b_i，这个系数代表着对应的参数对过去的任务到底重不重要，也称作“守卫”。因此在更新参数的时候，会改写损失函数，如下：

$$L'(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \sum_i b_i \left(\theta_i - \theta_i^b\right)^2$$

原来的损失函数写成 $L(\boldsymbol{\theta})$，在学习新任务时不会直接去最小化这个损失函数，否则就会发生灾难性遗忘的问题。我们会在右边一项增加一个当前参数 θ_i 与之前对应的参数 θ_i^b 的平均差并乘上我们讲的重要性系数 b_i 。这里 $b_i = 0$ 时，表示我们并不关心学习新任务时的参数需要跟过去的参数有什么联系，即一般的训练，这种情况下就容易发生遗忘。另一种极端就是当 b_i 趋近于无穷大时，即模型参数不肯在新任务上发生妥协，此时可能不会忘记旧的任务，但是也很有可能学不好新的任务，这种情况我们称作不妥协。

接下来比较关键的是 b_i 要怎么设定的问题，也就是某个参数 b_i 到底对任务的重要性有多少。其实有一种简单的控制变量的方法，就是移动或改变某个参数，如图 16.11 所示，我们移动 θ_1^b 时会发现在一定范围内损失值都是很小的，即接近最优的参数，那么我们就可以认为这个参数在一定范围可变，相应的重要性参数 b_1 就可以很小，即这个参数对旧任务来说不是很重要，反之像 θ_2^b 这种不能随意移动的参数对应的重要性参数就必须很大。

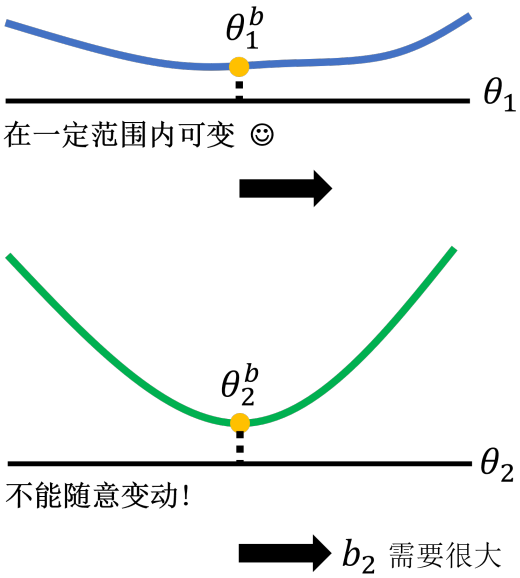


图 16.11　重要性参数设定示例

当然随着后续研究的深入，b_i 的设定也会有各种各样的方法，感兴趣的读者可以了解相关论文，这里就不一一讲解了。

其实在基于正则的方法出现之前，还有一类方法，叫做**梯度回合记忆（gradient episodic memory，MMD）**，它不是在参数上做限制，而是在梯度更新的方向上做限制，因此又被称为基于梯度的方法。原理如图 16.12 所示，它在计算当前任务梯度方向的同时，也会回去算历史任务对应的梯度方向，然后把两个梯度进行向量求和，得出实际的梯度方向，这样持续更新就能尽可能接近一个不会陷入灾难性遗忘的最优解了。当然新的梯度方向需要满足大于等于

0 的条件，否则很难朝最优解方向优化。

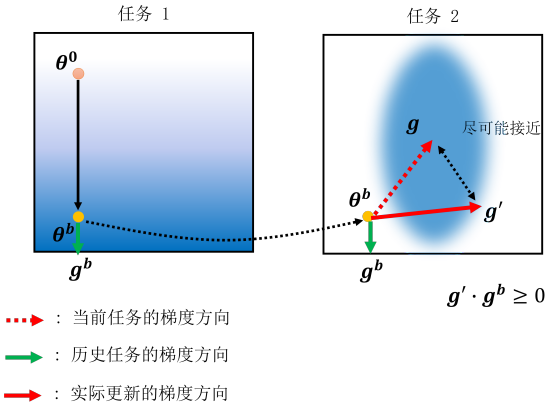


图 16.12　GEM 方法示例

但是这类方法需要把过去的数据同时存储下来，其实这跟终身学习的初衷有些违背了，因为终身学习本身就是希望不需要依赖过去的数据。但由于这里只是存储梯度这些少量的信息，像前面讲的选择性的突触可塑性方法也需要存储一些历史模型信息，所以在实际操作过程中也还能接受。

第 17 章 网络压缩

网络压缩（network compression）是一个很重要的方向，Bert 或 GPT 之类的模型很大，能不能够把这些硕大无朋的模型它缩小，能不能够简化这些模型，让它有比较少量的参数，但是跟原来的性能其实是差不多的呢，这就是网络压缩想要做的事情。很多时候，需要把这些模型用在资源受限的环境下。有时候我们会需要把这些机器学习的模型。举个例子，跑在智能手表，这些边缘设备（edge device）只有比较少的内存，只有比较少的计算力。模型如果太过巨大，手表可能会是跑不动的，所以会需要比较小的模型。

> Q：为什么会需要在这些边缘设备上面跑模型呢？为什么不把数据传到云端，直接在云端上做计算，再把结果传回到边缘设备？
> A：以手表为例，为什么一定要在手表上面做计算呢？一个常见的理由是延迟的问题。假设需要把数据传到云端，云端计算完再传回来，那中间就会有一个时间差。假设边缘设备是自驾车的一个传感器，也许自驾车的传感器需要做非常即时的回应，需要把数据传到云端再传回来，中间的延迟太长了，也许会长到是不能接受的。虽然在未来 5G 的时代延迟可能可以忽略不计，但需要在边缘设备上面做计算的理由，这个理由就是隐私。如果需要把数据传到云端，云端的系统持有者不就看到我们的数据了。因此为了保障隐私，也许在智能手表上直接进行计算并进行决策是一个可以保障隐私的做法。

接下来介绍五个以软件为导向的网络压缩技术，这些技术只是在软件上面对网络进行压缩，都不考虑硬件加速的部分。

17.1 网络剪枝

第一个技术是**网络剪枝（network pruning）**。网络剪枝就是要把网络里面的一些参数剪掉。剪枝就是修剪的意思，把网络里面的一些参数剪掉。为什么可以把网络里面的一些参数剪掉呢？这么大的网络里面有很多很多的参数，每一个参数不一定都有在做事。参数这么多的时候，也许很多参数只是在划水，什么事也没有做。这些没有做的参数放在那边，就只是占空间而已，浪费计算资源而已。为何就把它们剪掉呢？所以网络剪枝就是把一个大的网络中没有用的那些参数把它找出来，把它扔掉。人刚出生的时候，脑袋是空空的，神经元跟神经元间没什么连接。在六岁的时候会长出非常多的连接。但是随着年龄渐长，有一些连接就慢慢消失了。这个跟网络剪枝有异曲同工之妙。网络剪枝不是太新的概念，早在这个 90 年代，Yann Le Cun 的论文“Optimal Brain Damage”是讲网络剪枝，他把大脑（网络）剪枝，把它剪掉一些权重看成是一种脑损伤，最优的意思是要找出最好的剪枝的方法，让一些权重被剪掉之后，但是对这个脑的损伤是最小的。

网络剪枝其框架如图 17.1 所示，首先，先训练一个大的网络。接下来去衡量这个大的网络里面每一个参数或者是每一个神经元的重要性，去评估一下有没有哪些参数是没在做事的，或有没有哪些神经元是没在做事的。怎么评估某一个参数有没有在做事呢，怎么评估某一个参数重不重要呢。最简单的方法也许就是看它的绝对值。如果这个参数的绝对值越大，它可能越能对整个网络的影响越大。或者如果它的绝对值越接近零。也许对整个网络的影响越小，也许对我们任务的影响越小。

我们可以评估每一个神经元的重要性，把神经元当作修剪的单位。怎么看一个神经元重

不重要呢？比如计算这个神经元输出不为零的次数等等。总之有非常多的方法来判断一个参数或一个神经元是否重要。把不重要的神经元或是不重要的参数就剪掉，就把它从模型里面移出，就得到一个比较小的网络。但是做完这个修剪以后，通常模型的性能就会掉一点。因为有一些参数被拿掉了。所以这个网络当然是受到一些损伤，正确率就掉一点。但是我们会想办法让这个正确率再回升一点。可以把这个比较小的网络，把剩余没有被剪掉的参数，再重新做微调。把训练数据拿出来，把这个比较小的网络再重新训练一下。训练完之后，其实还可以重新再去评估一次每一个参数的正确性，还可以再去除掉，再剪掉更多的参数，再重新进行微调，这个步骤可以反复进行多次。

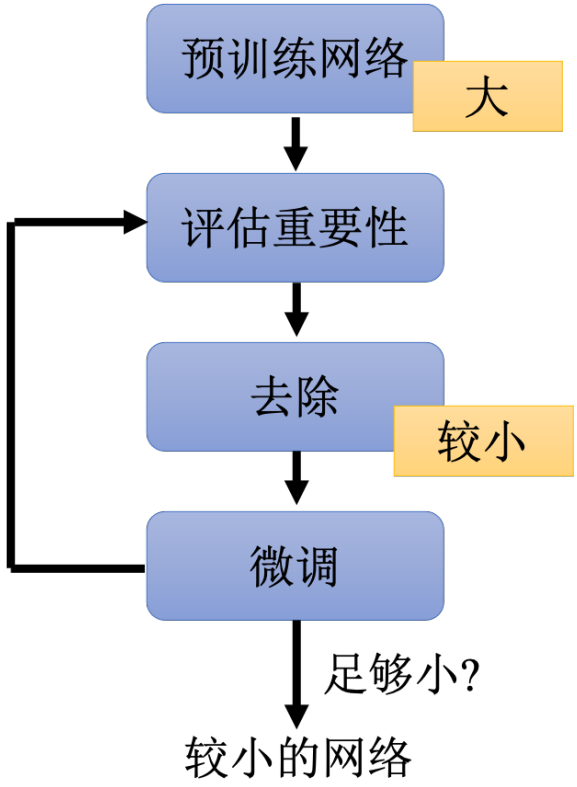


图 17.1　剪枝框架

为什么不一次剪掉大量的参数，因为在实验上，如果一次剪掉大量的参数，可能对网络的伤害太大了，可能会大到用微调也没有办法复原。所以一次先剪掉一点参数。比如说只剪掉 10% 的参数，然后再重新训练，然后再重新剪掉 10% 的参数，再重新训练，反复这一个过程。我们可以剪掉比较多的参数，当网络够小以后，整个过程就完成了，就得到一个比较小的网络。而且这个比较小的网络，也许它的正确率跟大的网络是没有太大的差别的。修剪的单位可以以参数为单位，也可以以神经元来当作单位，这两者作为单位，在实现上会是有显著不同。如果我们现在是以参数当作单位会发生什么事，假设我们是要评估某一个参数要不要被去掉，某个参数对整个任务而言重不重要能不能够被去掉，我们把这个不重要的参数去掉以后，得到的网络的形状可能会是不规则的。如图 17.2 所示，所谓不规则的意思是说，举例来说，红色的这个神经元连到接下来三个绿色的神经元，但第二个红色的神经元只连到两个绿色的神经元。或这个红色的神经元的输入只有两个蓝色的神经元，而这个红色的神经元的输入有四个蓝色的神经元。如果把参数当作单位来进行修剪的话，修剪完以后的网络的形状会是不规

则的。

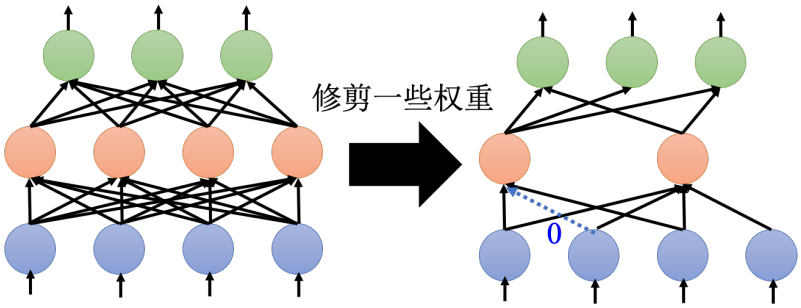


图 17.2 剪枝的问题

形状不规则最大的问题就是不好实现。用 PyTorch 要实现这种形状不规则的网络，不好实现。因为在 PyTorch 里面，定义第一个网络的时候，定义方法都是每一层有几个神经元，定义现在每一层要输入几个输入，输入有几个神经元，输出有几个神经元。或者输入多长的向量，输出有多长的向量。这种形状不固定的网络根本不好实现，而且就算硬是把这种形状不规则的网络实现出来，用 GPU 也很难加速。GPU 在加速的时候，就是把网络的计算看成一个矩阵的乘法，但是当网络是不规则的时候，不容易用矩阵的乘法来进行加速，我们不容易用 GPU 来进行加速，所以实际在做权重剪枝的时候，在实现上我们可能会把那些修剪掉的权重直接补零，就是修剪掉的权重不是不存在，它的值只是设为零。这样的好处是实现就比较容易，比较容易用 GPU 加速，但根本就没有真的把网络变小，虽然权重值是零，但还是存了这个参数，我们还是存了一个参数在内存里面，并没有真的把网络变小。这是以参数为单位来做剪枝的时候，实现上会遇到的问题。

如图 17.3 所示，紫色的这一条线是指稀疏程度（sparsity）。稀疏程度就是有多少百分比的参数现在被修剪掉了。紫色的这条线的值都很接近 1，代表有接近大概 95% 以上的参数都被修剪掉了。网络剪枝其实是一个非常有效率的方法，往往可以修剪到 95% 以上的参数，但是准确率只掉 1 2% 而已。所以这边参数修剪时，有 95% 的参数都被丢掉了，只剩下 5% 的参数，这个网络变得很小，的计算要很快吧。但实际上我们发现根本就没有加速多少，甚至可以说根本就没有加速。图 17.3 中的长条图显示的是在三种不同的计算资源上面，加速的程度。加速的程度要大过 1 才有加速，加速程度小于 1 其实是变慢的。在多数情况下根本就没有加速，多数情况下其实都是变慢，也就是把一些权重修剪掉，结果网络形状变得不规则，真的用 GPU 加速的时候，反而没有办法真的加速它，所以权重剪枝不一定是一个特别有效的方法。

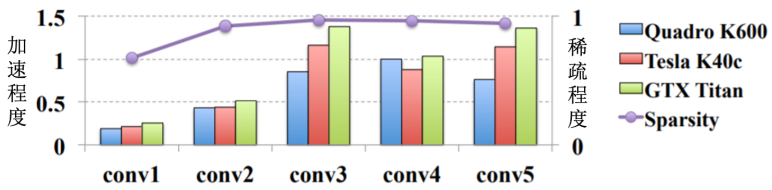


图 17.3 权重剪枝无法用 GPU 加速[1]

神经元剪枝，即以神经元为单位来做剪枝也许是一个比较有效的方法。如果用神经元做单位来剪枝，丢掉一些神经元以后，网络的架构仍然是规则的。这种用 PyTorch 也比较好实

现，实现的时候，只要改那个每一个层输入、输出的那个维度就好了，也比较好用 GPU 来加速。

接下来要问一个问题：我们先训练一个大的网络，再把它变小，而且说小的网络跟大的网络的正确率没有差太多。不直接训练一个小的网络就好了，直接训练一个小的网络比较有效率，还训练大的网络变小干嘛，根本是舍本逐末，为什么不直接训练小的网络？一个普遍的答案是大的网络比较好训练。如果直接训练一个小的网络，往往没有办法得到跟大的网络一样的正确率。我们可以先训练一个大的网络，再把它变小，正确率没有掉太多。但直接训练小的网络，得不到大的网络剪枝完变得小的网络一样的正确率。

为什么大的网络比较好训练？有一个假说叫做彩票假说（lottery ticket hypothesis）。假说代表说它不算是一个被实证的理论。但它现在只是一个假说而已，这个彩票假说是怎么解释为什么大的网络比较容易训练，直接训练一个小的网络没有办法得到跟大的网络一样的效果，一定要大的网络剪枝变小，结果才会好。彩票假说是这样说的，每次训练网络的结果不一定会一样。我们抽到一组好的初始的参数，就会得到好的结果；抽到一组坏初始的参数，就会得到坏的结果。但如何在彩票这个游戏里面得到比较高的中奖率，是不是就是包牌买比较多的彩券，可以增加中奖率，所以对一个大的网络来说也是一样的。大的网络可以视为是很多小的子网络（sub-network）的组合。如图 17.4 所示，我们可以想成是一个大的网络里面其实包含了很多小的网络。训练这个大的网络的时候，等于同时训练很多小的网络。每一个小的网络不一定可以成功的被训练出来。所谓成功的训练出来是说它不一定可以，通过梯度下降找到一个好的解，不一定可以训练出一个好的结果，不一定可以让它的损失变低。但是在众多的子网络里面，只要其中一个子网络成功，大的网络就成功了。而如果大的网络里面包含的小的网络越多，就好像是去买彩票的时候，买比较多的彩券一样，彩券越多，中奖的概率就越高。一个网络越大，它就越有可能成功的被训练起来。彩票假说在实验上是怎么被证实的？它在实验上的证实方式跟网络的剪枝非常有关系，所以直接看一下在实验上是怎么证实彩票假说的。

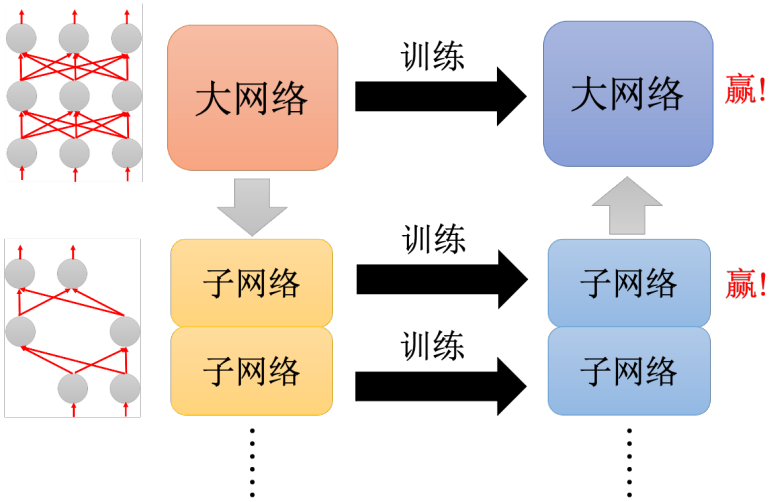


图 17.4　大网络包含了很多小网络

如图 17.5 所示，现在有一个大的网络，在这个大的网络上面一开始的参数是随机初始化。把参数随机初始化以后，得到一组训练完的参数。训练完的参数用紫色来表示，接下来用网络剪枝的技术，把一些紫色的参数丢掉，而得到一个比较小的网络。如果直接把这个小的网络里

面的参数，再重新随机的去初始化，也就是重训练一个一样大小的小的网络，就是我们把这个网络复制一次。但是参数完全不一样，重新再训练一次。重新再训练一次，直接训练这个小的网络，训练不起来，训练一个大的再把它变小，没问题。但假设这一个小的网络，再重新初始化参数的时候，我们用的跟这组红色的参数是一模一样的，就训练得起来。

这两组参数虽然都是随机初始化的，但是这组绿色的参数跟这组红色的参数是没有关系的。而这边这些随机初始化的参数是直接从这边的红色参数里面选出对应的参数，就是这边有四个参数。我们就是把这边对应到的这四个参数直接把它复制过来，这边有四个参数。我们就把这里面对应到的四个参数直接复制过来，把这里面的参数直接复制过来就训练了起来。如果用彩票假说来解释的话，就是这里面有很多子网络。而这一组初始化的参数，就是幸运的那一组可以训练得起来的子网络，所以用把这些网络，把这个大的网络训练完，再剪枝掉的时候，留下来的就是幸运的那些参数，可以训练得起来的那些参数，所以这一组初始化的参数是可以训练得起来的一个子网络。但是如果我们再重新随机初始化的话，就抽不到可以成功训练起来的参数，所以这个就是彩票假说。彩票假说非常地知名，它得到了 ICLR 2019 的最佳论文奖。

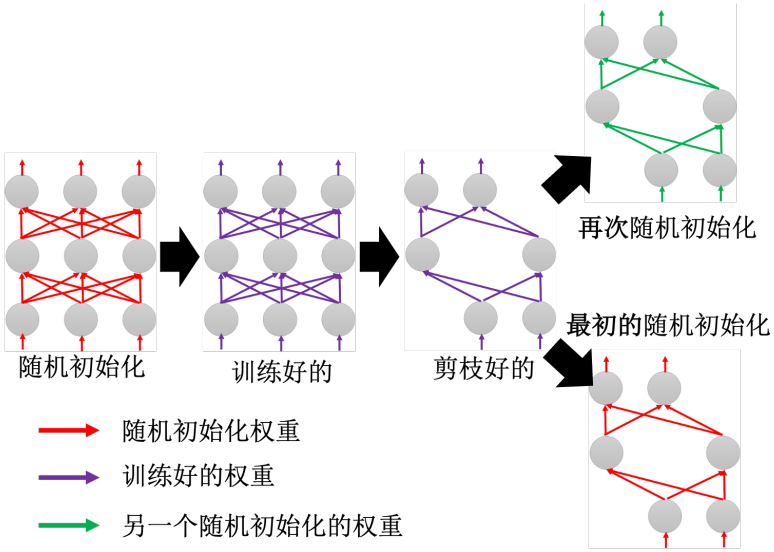


图 17.5　彩票假说

后面也有很多后续的研究，比如说有一篇有趣的研究叫做“Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask”[2]。解构这个彩票有什么有趣的结论，直接讲它的结论。第一个试了不同的剪枝策略，它做了一个非常完整的实验发现说，如果训练前跟训练后权重绝对值差距越大，剪枝那些网络得到的结果是越有效的。另外一个比较有趣的结果是，到底这一组好的初始化是好在哪里呢？如果我们只要不改变参数的正负号，就可以训练起来，小的网络只要不改变正负号，就可以训练起来。假设剪枝完以后，剩下的这个数。假设剪枝完以后，再把原来随机初始化的那些参数拿出来，它的值是 $0.9, 3.1, -9.1, 8.5$，可以完全不管它的数值，直接把正的数值的通通都用 $+\alpha$ 来取代，小于 0 的都用 $-\alpha$ 来取代，即 $+\alpha, +\alpha, -\alpha, +\alpha$ 。

用这组参数去初始化模型，这样也训练得起来，会跟用这组参数去初始化差不多。所以这个实验告诉我们说正负号是初始化参数，能不能够训练起来的关键，它的绝对值不重要，正负号才重要。

最后一个神奇的发现是，既然我们在想说一个大的网络里面有一些网络有一些子网络，它是特别是好的初始化的参数，它训练起来会特别地顺利。会不会一个大的网络里面甚至其实已经有一个子网络，它连训练都不用训练，直接拿出来就是一个好的网络呢？我们完全不用训练网络，直接把大的网络剪枝，就得到一个可以拿来做分类的分类器了，有没有可能是这个样子的呢。就好像米开朗基罗说："塑像就在石头里，我只是把不需要的部分去掉"，会不会在整个大的网络里面，算参数都是随机的，其中已经有一组参数，它就已经可以做分类了，把多余的东西拿掉，直接就可以得到好的分类结果的。

但是彩票假说不一定是对的，论文"Rethinking the Value of Network Pruning"[3] 是打脸彩票假说，而且神奇的是这篇文章跟彩票假说是同时出来的，它们同时出现在 ICLR 2019，它们得到了不太一样的结论。这篇文章试了两个数据集，还有好几种不同的模型，这个是没有剪枝过的网络的正确率，然后它试着去剪枝了一下网络，再重新去做微调。小的网络可以跟大的网络得到差不多的正确率。然后它说一般人的想像是，如果直接去训练小的网络，正确率会不如大的网络剪枝完以后的结果。

其实在这篇文章里面也有对彩票假说做出一些回应，它觉得彩票假说观察到的现象，也许只有在某一些特定的情况下才观察得到。根据这篇文章的实验，只有在学习率设比较小的时候，还有非结构化的（unstructured）的时候，就是剪枝的时候是以权重作为单位来做剪枝的时候，才能观察到彩票假说的现象，它发现说学习率调大，它就观察不到彩票假说的这个现象，所以到底彩票假说的正确性尚待更多的研究来证实。

17.2 知识蒸馏

接下来讲可以让网络变小的方法——**知识蒸馏（knowledge distillation）**。先训练一个大的网络，这个大的网络在知识蒸馏里面称为教师网络（teacher network），其是老师。我们要训练的是真正想要的小的网络，即学生网络（student network）。先训练一个大的网络称为教师网络。再根据这个大的网络来制造学生网络。在网络剪枝里面，直接把那个大的网络做一些修剪，把大的网络里面一些参数拿掉，就把它变成小的网络。在知识蒸馏里面是不一样的，这个小的网络（学生网络）是去根据教师网络来学习。假设要做手写数字识别，就把训练数据都丢到教师里面，教师就产生输出，因为这是一个分类的问题，所以教师的输出其实是一个分布。

比如教师的输出可能是看到这张图片 1 的分数是 0.7，7 的分数是 0.2，9 这个数字的分数是 0.1 等等。接下来给学生一模一样的图片，但是学生不是去看这个图片的正确答案来学习，它把老师的输出就当做正确答案，也就是老师输出 1 要 0.7，7 要 0.2，9 要 0.1。学生的输出也就要尽量去逼近老师的输出，尽量去逼近 1 是 0.7、7 是 0.2、9 是 0.1 这样的答案。学生就是根据老师的答案学，就算老师的答案是错的，学生就去学一个错的东西。

Q：为什么不直接训练一个小的网络？为什么不直接把小的网络去根据正确答案学习，而是要多加一个步骤先让大的网络学，再用小的网络去跟大的网络学习呢？这边的理由跟网络剪枝是一样的。
A：直接训练一个小的网络，往往结果就是没有从大的网络剪枝好。知识蒸馏的概念是一样的，因为直接训练一个小的网络，没有小的网络根据大的网络来学习结果要来得好。

其实知识蒸馏也不是新的技术，知识蒸馏最知名的一篇文章 Hinton 在 15 年的时候已经发表论文了。很多人会觉得知识蒸馏是 Hinton 提出来的，因为 Hinton 有一篇论文“Distilling the Knowledge in a Neural Network”。但其实在 Hinton 提出知识蒸馏这个概念之前，其实就有看过其他文章使用了一模一样的概念。举例来说，论文“Do Deep Nets Really Need to be Deep”是一篇 13 年的文章里面，也提出了网络蒸馏的想法。

为什么知识蒸馏会有帮助呢？一个比较直觉的解释是教师网络会提供学生网络额外的信息，如图 17.6 所示，如果直接跟学生网络这是 1，可能太难了。因为 1 可能跟其他的数字有点像，比如 1 跟 7 也有点像，1 跟 9 也长得有点像，所以对学生网络，我们告诉它：看到这张图片我们要输出 1。7 、9 的分数都要是 0，可能很难，它可能学不起来，所以让它直接去跟老师学，老师会告诉它这是 1。我们没有办法让它是 1 分，也没有关系。其实 1 跟 7 是有点像的，老师都分不出 1 跟 7 的差别。老师说 1 是 0.7，7 是 0.2，学生只要学到 1 是 0.7，7 是 0.2 就够了。这样反而可以让小的网络，学得比直接从头开始训练，直接根据正确的答案要学来得要好。

Hinton 论文里面甚至可以做到教师告诉学生哪些数字之间有什么样的关系这件事情，就可以让学生在完全没有看到某些数字的训练数据下，就可以把那一个数字学会。假设训练数据里面完全没有数字 7 ，但是教师在学的时候有看过数字 7 ，但是学生从来没有看过数字 7 。但光是凭着教师告诉学生说 1 跟 7 有点像，7 跟 9 有点像这样子的信息，都有机会让学生可以学到 7 长什么样子。就算它在训练的时候，从来没有看过 7 的训练数据。这是知识蒸馏的基本概念。

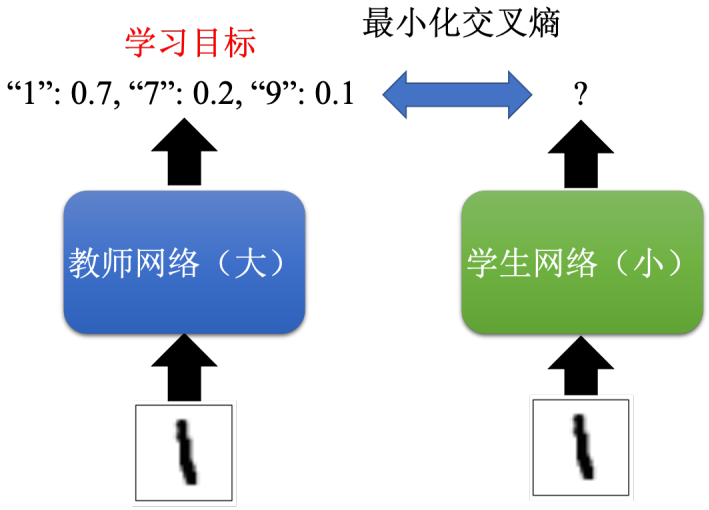


图 17.6　知识蒸馏

教师网络不一定要是单一的巨大网络，它甚至可以是多个网络的集成，训练多个模型，输出的结果就是多个模型，投票的结果就结束了。或者是把多个模型的输出平均起来的结果当做是最终的答案。虽然在比赛里面，常常会使用到集成的方法。如果在一个机器学习的比赛排行榜里面要名列前茅，往往凭借的就是集成技术，就是训练多个模型，把那么多的模型的结果通通平均起来。但是在实用上，集成会遇到的问题就训练了 1000 个模型，进来一笔数据，我们要 1000 个模型都跑过，再取它的平均，计算量也未免太大了。打比赛还勉强可以。要用在实际的系统上显然是不行的，可以把多个集成起来的网络综合起来变成一个，如图 17.7 所示。

这个就要用知识蒸馏的做法，就把多个网络集成起来的结果当做是教师网络的输出。让学生网络去学集成的结果，让学生网络去学集成的输出，让学生网络去逼进一堆网络集成起来的正确率。

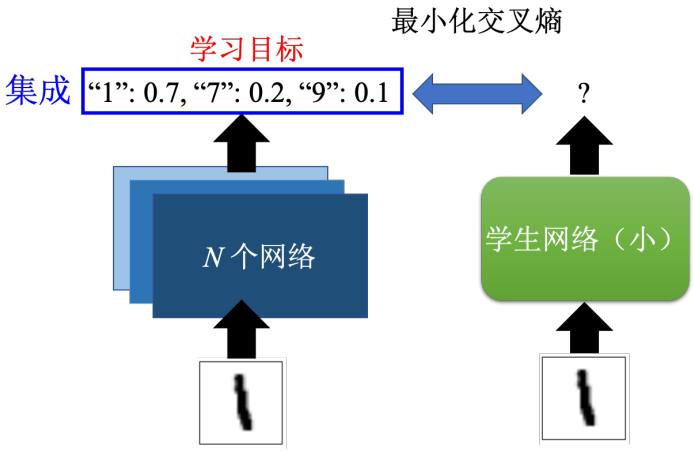


图 17.7　使用网络集成作为教师网络的输出

在使用知识蒸馏的时候有一个小技巧。这个小技巧是稍微改一下 Softmax 函数，会在 Softmax 函数上面加一个温度（temperature）。Softmax 要做的事情就是把每一个神经元的输出都取指数，再做归一化，得到最终网络的输出，如式 (17.1) 所示。网络的输出变成一个概率的分布，网络最终的输出都是介于 0 到 1 之间的。所谓温度，就是在做取指数之前，把每一个数值都除上 T，如式 (17.2) 所示。

$$y_i' = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \tag{17.1}$$

$$y_i' = \frac{\exp(y_i/T)}{\sum_j \exp(y_j/T)} \tag{17.2}$$

其中 T 是一个需要调整的超参数。假设 $T > 1$，温度 T 的作用就是把本来比较集中的分布变得比较平滑一点。举个例子，如式 (17.3) 所示，假设 y_1, y_2, y_3 是原始的值，y_1', y_2', y_3' 是 Softmax 后的值，softmax 后的值都趋近于 0。

$$\begin{aligned} y_1 &= 100 \quad y_1' = 1 \\ y_2 &= 10 \quad y_2' \approx 0 \\ y_3 &= 1 \quad y_3' \approx 0 \end{aligned} \tag{17.3}$$

假设教师网络的输出如式 (17.3) 所示，让学生要叫教师网络去跟这个结果学，跟直接和正确的答案学完全没有不同。跟教师学的一个好处就是，老师会告诉我们说哪些类别其实是比较像的，让学生网络在学的时候不会那么辛苦。但是假设老师的输出非常地集中，其中某一个类别是 1，其他都是 0。这样子跟正确答案学没有不同，所以要取一个温度。假设温度 T 设为 100，如式 (17.4)，对于教师，加上温度，分类的结果是不会变的。做完 Softmax 以后，最高分的还是最高分，最低分的还是最低分。所有类别的排序是不会变的，分类的结果是完全不会变的。但好处是每一个类别得到的分数会比较平滑、比较平均，我们拿这一个结果去给学生学才有意义，才能够把学生学得好。这是知识蒸馏的一个小技巧。

$$\begin{aligned} y_1/T &= 1 \quad & y_1' &= 0.56 \\ y_2/T &= 0.1 \quad & y_2' &= 0.23 \\ y_3/T &= 0.01 \quad & y_3' &= 0.21 \end{aligned} \tag{17.4}$$

Q：拿 Softmax 前的输出来训练，会发生什么事呢？
A：完全可以拿 Softmax 前的输出来训练，其实还会有人拿网络的每一层都拿来训练。比如一个大的教师网络有 12 层，小的学生网络有 6 层。可以让学生网络第 6 层像大的网络的第 12 层，学生网络的第 3 层，像大的网络的第 6 层，可不可以呢。往往我们做这种比较多的限制，其实可以得到更好的结果。

温度太大，模型会会改变很多。假设温度接近无穷大，这样所有的类别的分数就变得差不多，学生网络也学不到东西了，因此 T 又是另外一个超参数，它就跟学习率一样，这个是我们在做知识蒸馏的时候要调的参数。

17.3 参数量化

接下来介绍下一个技巧：**参数量化（parameter quantization）**。参数量化是说能否只用比较少的空间来储存一个参数。举个例子，现在存一个参数的时候可能是用 64 位或 32 位。可能不需要这么高的精度，用 16 或 8 位就够了。所以参数量化最简单的做法就是，本来如果存网络的时候，举例来说，我们是 16 个位存一个数值，现在改成 8 个位存一个数值。储存空间，网络的大小直接就变成原来的一半，而且性能不会掉很多，甚至有时候把储存参数的精度变低，结果还会稍微更好一点。还有一个再更进一步压缩参数的方法，即**权重聚类（weight clustering）**。

如图 17.8 所示，举个例子，先对网络的参数做聚类，按照这个参数的数值来分群。数值接近的放在一群，要分的群数会先事先设定好，比如设定好要分四群。比较相近的数字就被当做是一群。每一群都只拿一个数值来表示它。比如黄色的群所有数字的平均值是 -0.4，就用 -0.4 来代表所有黄色的参数。储存参数时，就只要记两个东西：一个是表格，这个表格是记录说每一群代表的数值是多少。另外一个要记录的就是每一个参数属于哪一群。假设群的数量设少一点，比如说设四群，这样只要两个位就可以存一个参数了。本来存一个参数可能要 16 位或 8 位，再进一步压缩到存一个参数只需要两个位就好。

其实还可以把参数再更进一步做压缩，使用哈夫曼编码（Huffman encoding）。哈夫曼编码的概念就是比较常出现的东西就用比较少的位来描述它，比较罕见的东西再用比较多的位来描述它。这样的好处平均起来，储存数据需要的位的数量就变少了，所以这个就是哈夫曼编码，所以可以用这些技巧来压缩参数，让我们储存每一个参数的时候需要的空间比较小，最终可以压缩到只拿一个位来存每一个参数。

网络里面的权重不是 $+1$，就是 -1。假设所有的权重只有正负 1 两种可能，每一个权重只需要一个位就可以存下来了。像这样子的这种二值权重（binary weight）的研究其实还蛮多的，具体可相关论文[4-6]。

虽然二值网络（binary network）参数值不是 $+1$，就是 -1，但这个网络的性能不一定会很差。二值网络里面的其中一个经典的方法，即二值连接（binary connect）。把二值连接这个技术用在三个图像识别的问题上，从最简单的 MNIST，还有稍微难一点的 CIFAR-10 以及

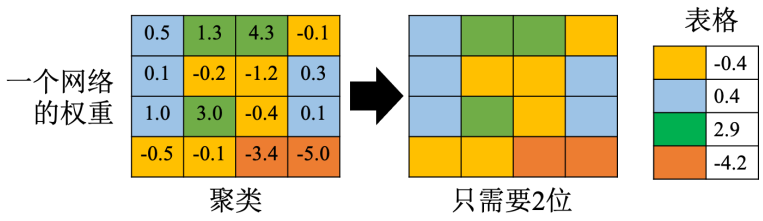


图 17.8 权重聚类

SVHN 数据集[4]。用二值连接结果居然是比较好的，所以用二值网络结果居然还比正常的网络的性能好一点。用二值网络的时候，给了网络比较大的限制，给网络容量（network capacity）比较大的限制，它比较不容易过拟合，所以用二值权重反而可以达到防止过拟合的效果。

Q：权重聚类要怎么做更新，每次更新都要重新分群吗?
A：其实权重聚类有一个很简单的做法。权重聚类是需要在训练的时候就考虑的。但是有一个简单的做法是，先把网络训练完，再直接做权重聚类。但这样直接做可能会导致聚类后的参数跟原来的参数相差太大。所以有一个做法是在训练的时候，要求网络的参数彼此之间比较接近。训练的量化可当做是损失的其中一个环节，直接塞到训练的过程中，让训练的过程中达到参数有权重聚类的效果。

Q：权重聚类里面每个聚类的数字要怎么决定呢?
A：决定好每个聚类的区间之后，取它们的平均。

17.4 网络架构设计

接下来介绍下网络架构的设计。通过网络架构的设计来达到减少参数量的效果。等一下要跟大家介绍深度可分离卷积（depthwise separable convolution）。在讲这个方法之前，先复习一下 CNN。在 CNN 的这种卷积层里面，每一个层的输入是一个特征映射。如图 17.9 所示，在这个例子里面，特征映射有两个通道，每一个滤波器的高度是 2，而且这个滤波器并不是一个长方形，而是一个立方体，通道有多少，的滤波器就得有多厚。再把这个滤波器扫过这个特征映射，就会得到另外一个正方形。我们有几个滤波器，输出特征映射就有几个通道。这边有 4 个滤波器，每一个滤波器都是立方体，输出特征映射就有 4 个通道。总共有 4 个滤波器，每一个滤波器的参数量是 $3 \times 3 \times 2$，要注意一下每一个滤波器，其实立方体总共的参数量是 $3 \times 3 \times 2 \times 4 = 72$ 个参数。

接下来介绍下深度可分离卷积。深度可分离卷积分成两个步骤，第一个步骤称为深度卷积（depthwise convolution）。如图 17.10 所示，深度卷积要做的事情是有几个通道，我们就有几个滤波器，每一个滤波器只管一个通道。举例来说，假设输入特征映射是两个通道，在深度的卷积层里面，只放两个滤波器。不像之前在一般的卷积层里面，滤波器的数量跟通道的数量是无关的。图 17.9 中的通道只有两个，但滤波器可以有四个。但在深度卷积里面滤波器数量与通道数量相同，每一个滤波器就只负责一个通道而已。假设浅蓝色的滤波器管第一个通道，浅蓝色的滤波器在第一个通道上面滑过去，就算出一个特征映射。深蓝色的滤波器管第二个通道，它就在第二个通道上面做卷积，也得到另外一个特征映射。因此在深度卷积里面，输入

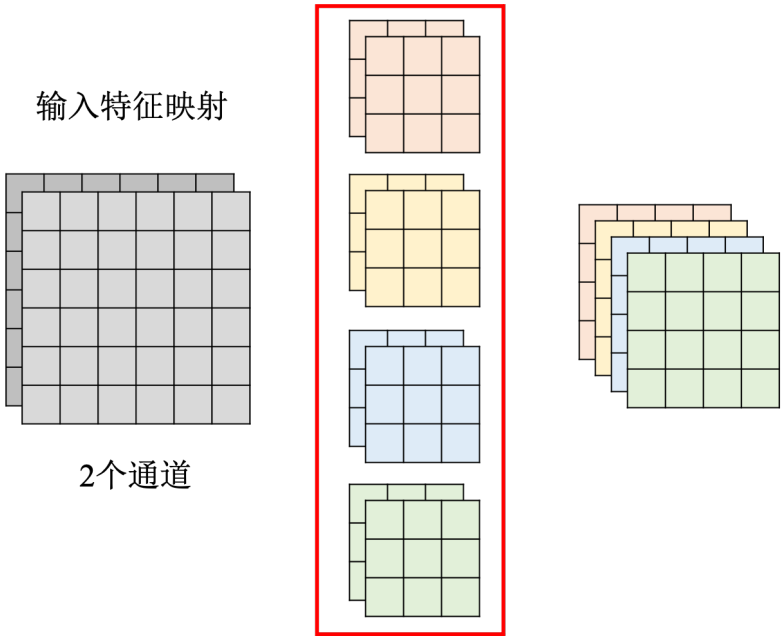


图 17.9 标准卷积

有几个通道，输出的通道的数量会是一模一样的，这个跟一般的卷积层不一样。一般的卷积层里面，输入跟输出的通道数量可以不一样，但在深度卷积里面，输入跟输出的通道数量是一模一样的。

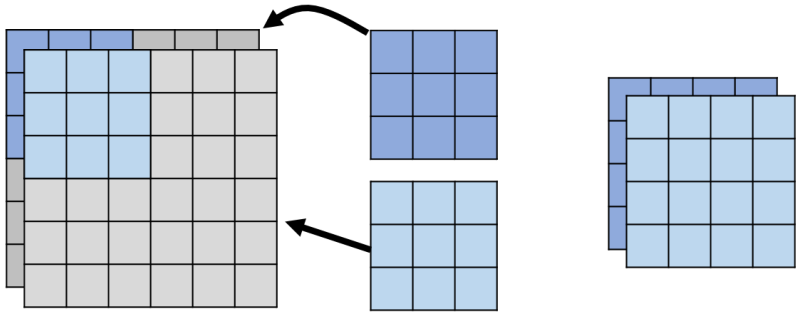


图 17.10 深度卷积

但是如果只做深度卷积，会遇到一个问题：通道和通道之间没有任何的互动。假设有某一个模式是跨通道才能够看得出来的，深度卷积对这种跨通道的模式是无能为力的，所以再多加一个点卷积。点卷积是指现在一样有一堆滤波器，这个跟一般的卷积层是一样的。但这边做一个限制是滤波器的大小，核大小通通都是 1×1。在一般的卷积层里面，滤波器大小可能开 $2 \times 2, 3 \times 3, 4 \times 4$。但是在点卷积里面，我们限制大小一定是 1×1。如图 17.11 所示，1×1 的滤波器的作用是去考虑不同通道之间的关系。所以第一个 1×1 的滤波器就是去扫过这个深度卷积出来的特征映射，然后得到另外一个特征映射。这边有另外三个滤波器，它们做的事情也是一样，每一个滤波器会产生一个特征映射，所以点卷积跟一般的卷积层是一样的地方是，输入跟输出的通道数量可以不一样。但是点卷积有一个非常大的限制，强制要求滤波器的大小只准是 1×1，只要考虑通道之间的关系就好了，不要考虑同一个通道内部的关系了。深度卷积已经做完同一个通道内部的关系，所以点卷积只专注于考虑通道跟通道间的关系就好。

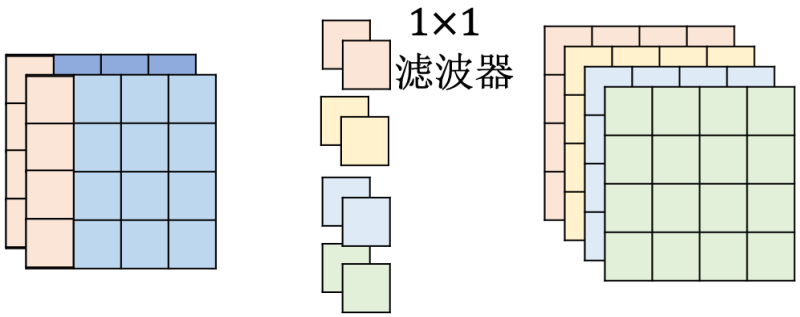


图 17.11 点卷积

如图 17.12 所示，先来计算一下这个方法的参数量。深度卷积里面两个滤波器，每一个滤波器大小是 3×3，所以总共是 $3 \times 3 \times 2 = 18$ 个参数。在点卷积里面，有四个滤波器，每一个滤波器的大小是 2，每个滤波器只用了两个参数，所以总共是 $2 \times 4 = 8$ 个参数。左边图是一般的卷积，右边这个图是深度卷积加点卷积。

现在来比较一下这两者参数量的差异。假设我们先预设好，输入通道数量就是 I 个通道，输出通道数量就是 O 个通道。核大小都是 $k \times k$。如果是一般的卷积，要有 $k \times k$ 的核大小，输入 I 个通道，输出 O 个通道，到底会需要多少参数呢？每一个滤波器的大小应该是 $k \times k$，核大小乘上输入通道的数量也就是 $k \times k \times I$。如果要输出 O 个通道，就需要 O 个滤波器，一个滤波器的参数量是 $k \times k \times I$。总参数量是 $(k \times k \times I) \times O$

如果是深度卷积加点卷积，要达到输入 I 个通道，输出 O 个通道。深度卷积的滤波器是没有厚度的，它的滤波器是没有厚度的，所以深度卷积所有的滤波器加起来的参数量只有 $k \times k \times I$ 而已，跟一般的卷积里面的一个滤波器的参数量是一样的。点卷积的参数量是 $I \times O$。假设输入通道的数量是 I，每一个 1×1 的卷积的高度会是 I。假设要输出 O 个通道，就要有 O 个 1×1 的卷积，所以点卷积的总参数量是 $I \times O$。如式 (17.5) 所示，把这两者进行比较，因为 O 通常是一个很大的值，通道数量可能设置为 256 或 512 ，所以先忽略 $\frac{1}{O}$，$\frac{1}{k \times k}$ 通常是比较在这一整项里面是比较关键的数值。这一整项的大小跟 $\frac{1}{k \times k}$ 是比较有关系的。假设核大小可能是 3×3 或者是 2×2。假设核大小是 2×2，把一般的卷积换成深度卷积加点卷积组合的话，网络大小就变成 $\frac{1}{4}$。假设核大小是 3×3，从卷积变成深度卷积加点卷积的时候，网络大小就变成原来的 $\frac{1}{9}$。

$$\begin{aligned} &\frac{k \times k \times I + I \times O}{k \times k \times I \times O} \\ &= \frac{1}{O} + \frac{1}{k \times k} \end{aligned} \tag{17.5}$$

在深度可分离卷积之前，就已经有一个方法是用低秩近似（low rank approximation）来减少一层网络的参数量。如图 17.13 所示，假设有某一个层有 N 个神经元，输出有 M 个神经元。假设输入是 N 个神经元，输出是 M 个神经元，这两层之间的参数量是 $N \times M$。只要 N 跟 M 其中一者很大，W 的参数就很多了。怎么减少这个参数量呢？有一个非常简单的方法是在 N 跟 M 中间再插一层，这一层就不用激活函数，这一层神经元的数量是 K。原来本来只有一层，现在拆成两层，这两层里面的第一层用 V 来表示，第二层用 U 来表示。这两层的网络参数量反而是比较少的。原来一层的网络的参数量是 $M \times N$。如果拆成两层网络，第一层是 $N \times K$，第二层是 $K \times M$。如果 K 远小于 M 跟 N，U 跟 V 的参数量加起来是比

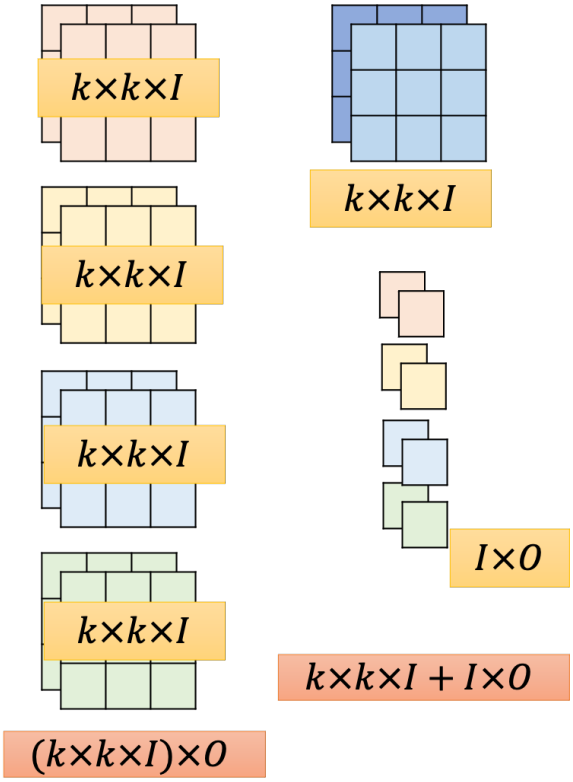


图 17.12　深度可分离卷积参数计算

W 还要少的多。W 是 $N \times M$，U 跟 V 的参数量加起来是 $K \times (N + M)$，只要 K 够小，整体而言 $U + V$ 的参数量就会变少。之前过去常有的做法就是 $N = 1000, M = 1000$，可以塞个 20 或 50，参数量就减少蛮多的。这样子的方法虽然可以减少参数量，但它还是会有一些限制。把 W 拆解成 $U \times V$ 的时候，把 W 用 U 跟 V 两层来分开表示的时候，我们会减少 W 的可能性，本来 W 可以放任何的参数。但假设把 W 拆成 U 跟 V，矩阵 W 的秩会小于等于 K。如果不知道秩是什么也没有关系，反正就是 W 会有一些限制，所以会变成说不是所有的 W 都可以变成当作参数，参数会变成有一些限制。这个方法就是拿来减少参数的一个非常常用的做法。其实刚才讲的深度卷积加点卷积（pointwise convolution）其实用的就是把一层拆成两层的概念。

怎么说呢？如图 17.14 所示，先来看一下原来的卷积。在原来的卷积里面，左上角红色的这一个矩的这个参数是怎么来的。是不是有一个红色的这个滤波器放在特征输入的特征映射的左上角以后所得到的。在这个例子里面，一个滤波器的参数量是 $3 \times 3 \times 2 = 18$ ，把滤波器里面的 18 个参数跟输入特征映射左上角的这 18 个数值做内积以后，就会得到这边输出特征映射左上角的值，所以每一个滤波器有 18 个参数。

如果拆成深度卷积加点卷积两阶段，左上角输出特征映射，左上角这个数值来自于中间的深度卷积的输出。所以左上角这个值来自于中间深度卷积的输出，左上角这两个值来自于输入特征映射，第一个通道左上角这 9 个值跟第二个通道左上角这 9 个值。我们有两个滤波器，这两个滤波器分别是 9 个输入，得到输出，接下来这两个滤波器的输出再把它综合起来，得到最终的输出。所以本来是直接从这 18 个数值变成一个数值，现在是分两阶，18 个数值变两个数值再变一个数值。如果看黄色的这个特征映射左下角这个参数黄色的特征映射。左

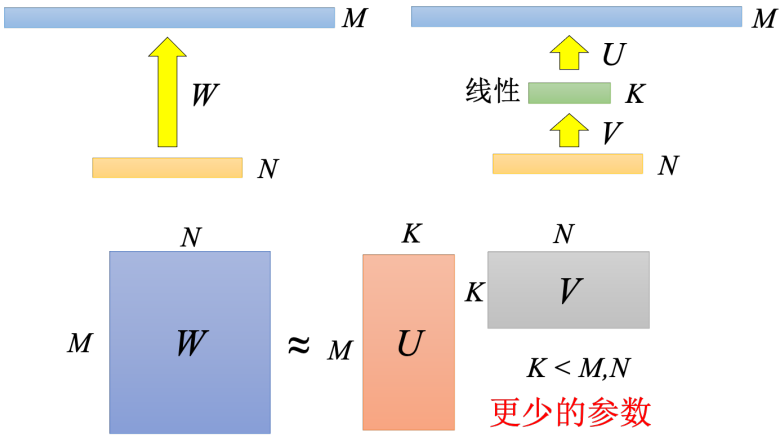


图 17.13　低秩近似示例

下角来自于深度卷积的输出。而左下角这两个数值来自于这个滤波器左下这来自于这个深度卷积，来自于输入的这个特征映射。左下角的这 18 个数值，把一般的卷积拆成深度卷积加点卷积的时候，就可以看成是把一层的网络拆解成两层的网络，其原理跟低秩近似是一样的，把一层拆成两层，这个时候它对于参数的需求反而是减少了，这个是有关网络架构（network architecture）的设计。

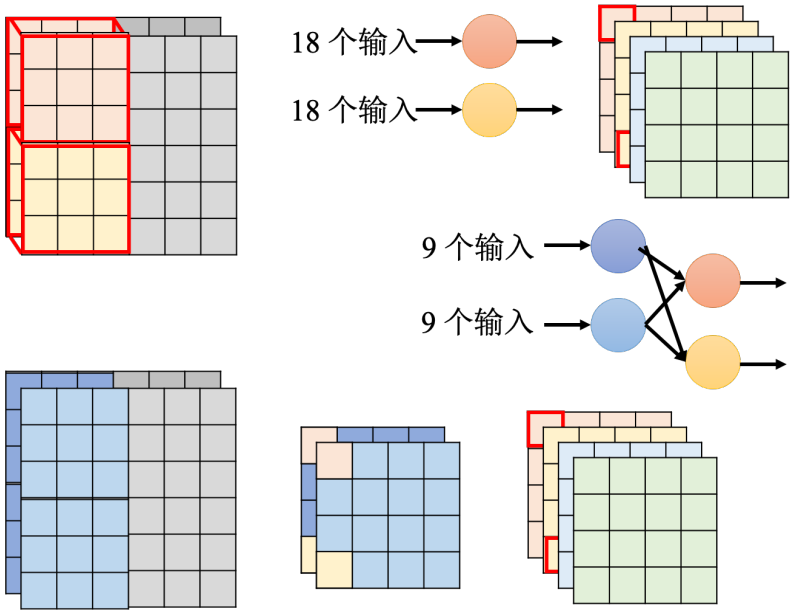


图 17.14　标准卷积与深度可分离卷积对比

17.5　动态计算

最后一个介绍的方法是**动态计算（dynamic computation）**。动态计算跟前几个方法想要达成的目标不太一样。前几个方法就是单纯的把网络变小，而动态计算希望网络可以自由地调整它需要的计算量。为什么期待网络可以自由地调整它需要的计算量呢？因为有时候我们可能同样的模型会想要跑在不同的设备上面，而不同的设备上面的计算资源是不太一样的。

所以期待训练好一个网络以后，放到新的设备上面，不需要再重训练这个网络。因为这个训练一个神奇的模型，这个神奇的模型本来就可以自由调整所需要的计算资源。计算资源少的时候只需要少的计算资源，就可以计算。计算资源大的时候，它就可以充分利用充足的计算资源来进行计算。那另外一个可能是就算是在同一个设备上面，也会需要不同的计算。举个例子，假设手机非常有电，可能就会有比较多的计算资源。假设手机没电，可能就需要把计算资源留着做其他的事情，网络可能可以分到的计算资源就比较少，所以就算是在同一个设备上面。我们也希望一个网络可以根据现有的计算资源，比如说手机现在的电量还有多少来自由地调整它对计算量的需求。

Q：为什么不直接准备一大堆的网络，假设需要各种应付各种不同计算资源的情况，为什么不训练 10 个网络，从计算量最少的到计算量最大的，根据计算的情况去选择不同的网络呢？
A：假设我们是在同一台手机上，需要根据不同的情况做不同的因应，可能就需要训练一大堆的网络。而手机上的储存空间有限，就是要减少计算量。但是如果我们需要训练一大堆的网络，就需要一大堆的储存空间。这可能不是我们要的，其实期待可以做到一个网络可以自由地调整其对计算资源的需求。

怎么让网络自由地调整其对计算资源的需求呢？一个可能的方向是让网络自由地调整它的深度。比如图像分类，如图 17.15 所示，它就是输入一张图片，输出是图片分类的结果，可以在这个层和层中间再加上一个额外的层。这个额外的层的工作是根据每一个隐藏层的输出决定现在分类的结果。当计算资源比较充足的时候，可以让这张图片去跑过所有的层，得到最终的分类结果。当计算资源不充足的时候，可以让网络决定它要在哪一个层自行做输出。比如说计算资源比较不充足的时候，通过第一个层，就直接丢到这个额外的第 1 层，就得到最终的结果了。怎么训练这样一个网络呢？其实概念比我们想像的还要简单，训练的时候都有标签的数据。一般在训练的时候，只需要在意最后一层网络的输出，希望它的输出跟标准答案越接近越好。但也可以让标准答案跟每一个额外层的输出越接近越好。把所有的输出跟标准答案的距离通通加起来，把所有的输出跟标准答案的交叉熵通通加起来得到 L

$$L = e_1 + e_2 + \cdots + e_L \tag{17.6}$$

再去最小化这个 L，然后就结束了，这个训练方法是可以用的。确实可以用我刚才讲的训练方法，就是每一层接出来做训练，然后把所有接出来的结果去跟标准答案算距离，最小化所有接出来的结果跟标准答案的距离，确实可以用这个方法达到动态的深度，但是其实它不是一个最好的方法。

比较好的方法可以参考 MSDNet[7]。另外还可以让网络自由地决定它的宽度，怎么让网络自由决定它的宽度。设定好几个不同的宽度，同一张图片丢进去。在训练的时候，同一张图片丢进去，每个不同宽度的网络会有不同的输出。我们在希望每一个输出都跟正确答案越接近越好就结束了，把所有的输出跟标准答案的距离加起来得到一个损失，最小化这个损失就结束了。

$$L = e_1 + e_2 + e_3 \tag{17.7}$$

图 17.16 中的三个网络并不是三个不同的网络，它们是同一个网络可以选择不同的宽度。

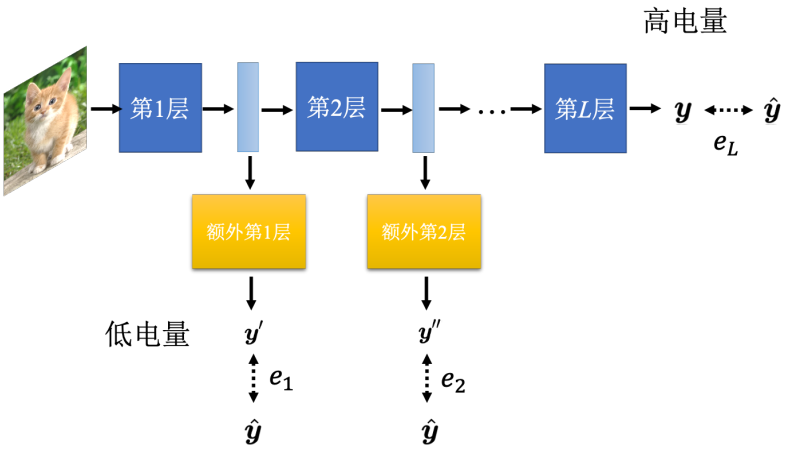


图 17.15 动态深度

相同颜色是同一个权重。只是在最左边情况的时候，整个网络所有的神经元都会被用到。但是在中间情况的时候，可能会决定有 25% 的神经元不需要用。或者在最右边的情况，50% 的神经元不要用它。在训练的时候，就把所有的情况一起考虑，然后所有的情况都得到一个输出，所有的输出都去跟标准答案计算距离，要让所有的距离都越小越好就结束了。但是实际上这样训练也是有问题的。所以需要一些特别的想法来解决这个问题。

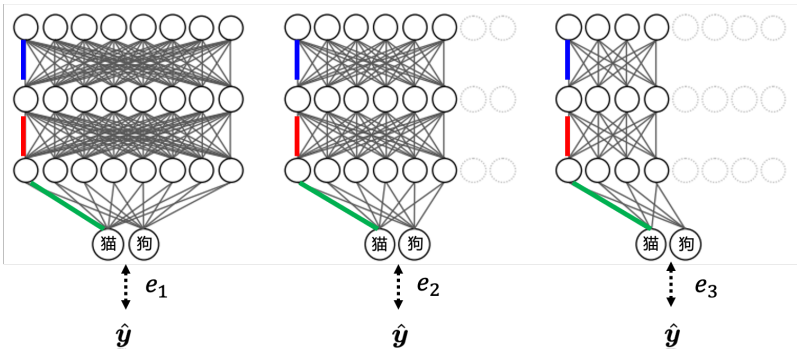


图 17.16 动态宽度

有关深度（depthwise）动态的宽度的网络，怎么训练这件事？大家可以参考论文“Slimmable Neural Networks”[8]。那刚才讲的是可以训练一个网络，可以自由去决定它的深度跟宽度。但是还需要我们去决定，今天电池电量少于多少的时候，就用多少层或者是多宽的网络。但是有没有办法让网络自行决定，根据情况决定它的宽度或者是深度呢。

Q：为什么需要网络自己去决定它的宽度跟深度呢？
A：因为有时候，就算是同样是图像分类的问题，图像的难易程度不同，有些图像可能特别难，有些图像可能特别简单。对那些比较简单的图像，也许只要通过一层网络就已经可以知道答案了，对于一些比较难的问题，需要多层网络才能知道答案。

举例来说，如图 17.17 所示，同样是猫，但有只猫是被做成一个墨西哥卷饼，所以这是一个特别困难的问题。也许这张图片只通过一个层的时候，网络会觉得它是一个卷饼；在通过

第二个层的时候，还是一个卷饼。要通过很多个层的时候，网络才能够判断它是一只猫。这种比较难的问题就不应该在中间停下来。可以让网络自己决定这是一张简单的图片，所以通过第一层就停下来。这是一个比较困难的图片，要跑到最后一层才停下来。具体怎么做可参考论文“SkipNet: Learning Dynamic Routing in Convolutional Networks”[9]、Runtime Neural Pruning[10] 和“BlockDrop: Dynamic Inference Paths in Residual Networks”[11]。

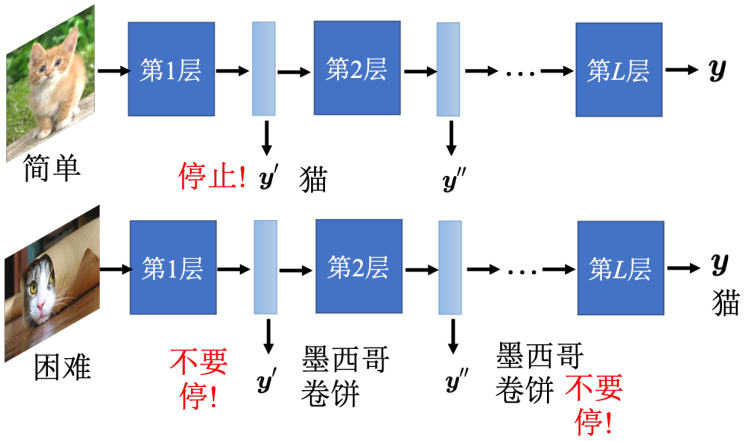


图 17.17　计算量取决于采样难度

所以像这种方法不一定限制在计算资源比较有限的情况。有时候就算计算资源比较很充足，但是对一些简单的图片，如果可以用比较少的层，得到需要的结果，其实也就够了，这样就可以省下一些计算资源去做其他的事情。

以上就是网络压缩的五个技术。前面四个技术都是让网络可以变小，这四个技术并不是互斥的。其实在做网络压缩的时候，可以既用网络架构，也做知识蒸馏，还可以在做完知识蒸馏以后，再去做网络剪枝。还可以在做完网络剪枝以后，再去做参数量化。如果想要把网络压缩到很小，这些方法都是可以一起被使用的。

参考文献

[1] WEN W, WU C, WANG Y, et al. Learning structured sparsity in deep neural networks [J]. Advances in neural information processing systems, 2016, 29.

[2] ZHOU H, LAN J, LIU R, et al. Deconstructing lottery tickets: Zeros, signs, and the supermask[J]. Advances in neural information processing systems, 2019, 32.

[3] LIU Z, SUN M, ZHOU T, et al. Rethinking the value of network pruning[C]//ICLR. 2019.

[4] COURBARIAUX M, BENGIO Y, DAVID J P. Binaryconnect: Training deep neural networks with binary weights during propagations[J]. Advances in neural information processing systems, 2015, 28.

[5] COURBARIAUX M, HUBARA I, SOUDRY D, et al. Binarized neural networks: Train-

ing deep neural networks with weights and activations constrained to+ 1 or-1[J]. arXiv preprint arXiv:1602.02830, 2016.

[6] RASTEGARI M, ORDONEZ V, REDMON J, et al. Xnor-net: Imagenet classification using binary convolutional neural networks[C]//European conference on computer vision. Springer, 2016: 525-542.

[7] HUANG G, CHEN D, LI T, et al. Multi-scale dense networks for resource efficient image classification[C]//ICLR. 2018.

[8] YU J, YANG L, XU N, et al. Slimmable neural networks[C]//ICLR. 2019.

[9] WANG X, YU F, DOU Z Y, et al. Skipnet: Learning dynamic routing in convolutional networks[C]//Proceedings of the European Conference on Computer Vision (ECCV). 2018: 409-424.

[10] LIN J, RAO Y, LU J, et al. Runtime neural pruning[J]. Advances in neural information processing systems, 2017, 30.

[11] WU Z, NAGARAJAN T, KUMAR A, et al. Blockdrop: Dynamic inference paths in residual networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 8817-8826.

第 18 章 可解释性人工智能

我们已经介绍了众多的机器学习模型以及不同的研究领域以及各种有趣的应用场景，当一个机器学习算法真正在工业界落地时，会有一个问题：机器学习模型往往是一个黑盒子，无法解释这个黑盒子中是如何通过输入得到输出的。但模型的安全性在行业落地时十分重要，因此我们势必要对于人工智能算法、模型的解释性的进行研究。

18.1 可解释性人工智能的重要性

我们首先介绍**可解释性人工智能（eXplainable Artificial Intelligence，XAI）**的概念，以及可解释性人工智能的重要性。目前为止，我们已经训练了很多的模型。比如图像识别模型，给它一张图片，模型会给你图像的类别。但我们并不满足于此，接下来我们要机器给我们它得到答案的理由，这就是可解释性人工智能，学者们普遍将可解释性人工智能称为XAI。首先开始介绍技术之前，我们需要讲一下为什么可解释性人工智能是一个重要的研究领域。其实本质上的原因是就算机器可以得到正确的答案，也不代表它一定非常“聪明”。举一个例子，过去有一匹神马它很聪明，它可以做数学问题。比如问它 $\sqrt{9}$ 是多少，它就开始计算得到答案，并且用它的马蹄去跺地板来告知答案。如果答案是 3，它就敲三下，停下来代表它得到正确的答案。旁边的人就会欢呼，给它胡萝卜吃。后来有人就很怀疑为什么神马可以解数学问题，它只是一匹马，它为什么能够理解数学问题呢。慢慢地有人发现，只要没有人围观的时候，神马就会答不出数学问题，没有人看它的时候，它就不会解数学问题。所以它其实只是侦测到旁边人类微妙的情感变化，知道它什么时候要停下跺马蹄，它就可以有胡萝卜吃。它并不是真的学会解数学的问题，而我们看到种种人工智能的应用，有没有可能跟神马是一样的状况呢？

以上是一个故事，当然在很多的真实应用中，可解释性的机器学习，或者说可解释性的模型往往是必须的。举例来说，银行可能会用机器学习的模型来判断要不要贷款给某一个客户，但是根据法律的规定银行作用机器学习模型来做自动的判断它必须要给出一个理由。所以这个时候，我们不是只训练机器学习模型就好，我们还需要机器学习的模型是具有解释力的。机器学习未来也会被用在医疗诊断上，但医疗诊断是人命关天的事情，如果机器学习的模型只是一个黑箱，不会给出诊断的理由的话，那我们又要怎么相信它做出的是正确的判断呢。现在也有人想把机器学习的模型用在法律上，比如说帮助法官判案，比如一个犯人能不能够被假释，但是我们怎么知道机器学习的模型它是公正的呢，它是不是有种族歧视的问题呢。所以我们希望机器学习的模型不只得到答案它还要给我们得到答案的理由。再更进一步，自动驾驶汽车未来可能会满街跑，但是当自动驾驶汽车突然急刹的时候导致车上的乘客受伤，这个自动驾驶系统有没有问题呢？这也许取决于它急刹的理由，如果它是看到有一个老人在过马路所以急刹，那也许它是对的，但是假设它只是无缘无故就突然发狂要急刹，那这个模型就有问题了。所以对它的种种行为、种种决策，我们希望知道决策背后的理由。更进一步，也许机器学习的模型如果具有解释力，未来我们可以凭借着解释的结果再去修正模型。

我们期待也许未来当我们知道深度学习模型犯错的时候，它是错在什么样的地方，它为什么犯错，也许我们可以有更好的、更有效率的方法来提升模型。当然这个是未来的目标，离用可解释性的机器学习做到上述模型的想法还有很长的一段距离。但是已经有一些方法可以让模型变得比较容易解释，也许未来我们可以把这些方法应用在深度学习的模型上，让深度学习的模型也变得比较容易解释。

有人可能会想说我们之所以这么关注可解释性的机器学习的议题，也许是因为深度的网络本身就是一个黑箱。我们能不能够用其它的机器学习的模型呢？如果不要用深度学习的模型，而改采用其他比较容易解释的模型会不会就不需要研究可解释性机器学习了呢。举例来说，假设我们都采用线性模型，它的解释的能力是比较强的，我们可以轻易地知道根据一个线性模型里面的每一个特征的权重，知道线性的模型在做什么事。所以训练完一个线性模型后，我们可以轻易地知道它是怎么得到它的结果的。但是线性模型的问题是它没有非常地强大，它的表达能力是比较弱的。线性模型有很巨大地限制，所以我们才很快地进入了深度的模型。深度模型它的坏处就是它不容易被解释，深度网络它就是一个黑盒子，黑盒子里面发生了什么事情，我们很难知道。虽然它比线性的模型更好，但是它的解释的能力是远比线性的模型要差的。所以讲到这里，很多人就会得到一个结论，我们就不应该用这种深度的模型，我们不该用这些比较强大的模型，因为它们是黑盒子。但是这样的想法其实就是削足适履，我们因为一个模型它非常地强大，但是不容易被解释就扬弃它吗？我们不是应该是想办法让它具有可以解释的能力吗？所以我们的目标就是要让深度的模型也具有解释的能力，而不是说我们就不要用深度的模型了。当然对于机器学习的可解释性还有很多的讨论，但是其重要性是不言而喻的。

18.2 决策树模型的可解释性

我们首先介绍一下一个比较简单的机器学习模型，其在设计之初就已经有了比较好的可解释性，这个模型就是决策树模型。决策树相较于线性的模型，它是更强大的模型。而决策树的另外一个好处，相较于深度学习它具有良好的可解释性。比如从决策树的结构，我们就可以知道模型是凭借着什么样的规则来做出最终的判断。所以我们希望从决策树模型进行可解释性的研究，再扩展到其他机器学习模型，甚至深度模型。

我们首先简单介绍一下决策树，它有很多的节点，那每一个节点都会问一个问题，让你决定向左还是向右。最终当你走到节点的末尾，即叶子节点的时候，就可以做出最终的决定。因为在每一个节点都有一个问题，我们看那些问题以及答案就可以知道现在整个模型凭借着什么样的特征如何做出最终的决断。所以从这个角度看来，决策树它既强大又有良好的可解释性。那我们是不是就可以用决策树来解决所有的问题呢？其实不是的，它是一个树状的结构，那我们可以想像一下，如果特征非常地多，得到决策树就会非常地复杂，就很难去解释它了。因为其节点太多而且很难分析得到整个模型的规则。所以复杂的决策树也有可能是一个黑盒子，它也有可能是一个非常地复杂的模型，所以我们也不能够一味地去使用决策树。

另外一方面，我们是怎么实际使用决策树这个技术的呢？很多同学都会说，这个打 Kaggle 比赛的时候，深度学习不是最好用的，决策树才是最好用的，决策树才是 Kaggle 比赛的常胜将军。但是其实当你在使用决策树的时候，并不是只用一棵决策树，你真正用的技术叫做随机森林。真正用的技术其实是好多棵决策树共同决定的结果。一棵决策树可以凭借着每一个节点的问题和答案知道它是怎么做出最终的判断的，但当你有一片森林的时候，你就很难知道说这一片森林是怎么做出最终的判断的。所以决策树也不是最终的答案，并不是有决策树，我们就解决了可解释性机器学习的问题。

18.3 可解释性机器学习的目标

为了解释比如决策树、随机森林的意义，我们首先应该定义可解释性的目标是什么。或者说什么才是最好的可解释性的结果呢？很多人对于可解释性机器学习会有一个误解，觉得一个好的可解释性就是要告诉我们整个模型在做什么事。我们要了解模型的一切，我们要知道它到底是怎么做出一个决断的。但是这件事情真的是有必要的吗？虽然我们说机器学习模型，深度网络是一个黑盒子，不能相信它，但世界上有很多黑盒子，比如人脑也是黑盒子。我们其实也并不完全知道，人脑的运作原理，但是我们可以相信，另外一个人做出的决断。那为什么深度网络是一个黑盒子，我们就没有办法相信其做出的决断呢？我们可以相信人脑做出的决断，但是我们不可以相信深度网络做出的决断，这是为什么呢？

以下是一个和机器学习完全无关的心理学实验，这个实验是 1970 年一个哈佛大学教授做的。这个实验是这样，在哈佛大学图书馆的打印机经常会有很多人都排队要印东西，这个时候如果有一个人跟他前面的人说拜托请让我先印 5 页，这个时候你觉得这个人会答应吗？据统计有 60% 的人会让他先印。但这个时候你只要把刚才问话的方法稍微改一下，你说拜托请让我先印，因为我赶时间，他是不是真的赶时间没人知道，但是当你说你有一个理由所以你要先印的时候，这个时候接受的程度变成 94%。神奇的事情是，就算你的理由稍微改一下，比如说请让我先印因为我需要先印，仅仅是这个样子接受的程度也变成 93%。所以人就是需要一个理由，你为什么要先印，你只要讲出一个理由，就算你的理由是因为我需要先印大家也会接受。

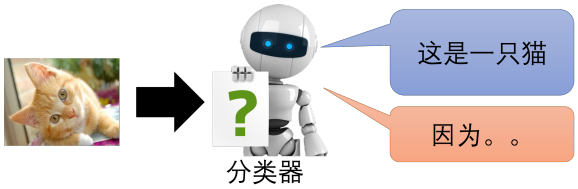
所以会不会可解释性机器学习也是同样的道理。在可解释性机器学习中，好的解释就是人能接受的解释，人就是需要一个理由让我们觉得高兴。因为很多人听到，深度网络是一个黑盒子他就不爽，但是你告诉他说这个是可以被解释的，给他一个理由，他就高兴了。所以或许好的解释就是让人高兴的解释。其实这个想法，这个技术的进展是蛮接近的。

18.4 可解释性机器学习中的局部解释

可解释性机器学习可以被分成两大类，第一大类叫做局部的解释，第二大类叫做全局的解释，如图 18.1 所示。局部的解释是，比如有一个图像分类器，输入一张图片，它会判断出是一只猫，机器要回答问题是为什么它觉得这张图片是一只猫。根据某一张图片来回答问题，这个叫做局部的解释。还有另外一类，称为全局解释。其指还没有给分类器任何图片，而直接问对一个分类器而言，什么样的图片叫做猫。我们并不是针对任何一张特定的图片来进行分析，我们是想要知道有一个模型它里面有一些参数的时候，对这些参数而言什么样的东西叫作一只猫。

接下来我们先来看第一大类，第一大类是为什么你觉得一张图片是一只猫。再具体一点些，给机器一张图片，它知道图片是一只猫的时候，到底是这个图片里面的什么东西让模型觉得它是一只猫。或者讲的更宽泛一点，假设模型的输入叫做 x，这个 x 可能是一张图片，可能是一段文字，可能是一段音频，可能是一段视频，可能是一段时间序列的数据。其可以拆成多个部分，x 可以拆成 x_1 到 x_n，这些部分对应起来可能是像素，也可能是文字，也可能是音频的频谱，也可能是视频的每一帧，也可能是时间序列的每一个时间点。在这些部分里面，哪一个对于机器做出最终的决断是最重要的？

如何知道一个部分的重要性呢？基本的原则是我们把所有的部分都拿出来，把每一个部分做改造或者是删除。如果我们改造或删除某一个部分以后，网络的输出有了巨大的变化，就



局部的可解释性

为什么你认为这个图像是一只猫?

全局的可解释性

“猫”长什么样子?

(并不指定某一张图像)

图 18.1 可解释性机器学习的两大类别

知道这个部分没它不行，它很重要。我们再使用图像举例，想要知道一个图像里每一个区域的重要性的时候，就将这个图片输入到网络里。接下来在这个图片里面不同的位置放上灰色的方块，当这个方块放在不同的地方的时候，网络会输出不同的结果。比如一只狗的图片，当我们把灰色的方块移动到狗的脸上的时候，我们网络就不觉得它看到一只狗；但如果把灰色的方块放在狗的四周，这个时候机器就觉得它看到的仍然是狗。所以模型就知道它不是看到球，觉得它看到狗，也不是看到地板、墙壁，才觉得看到的是狗，而是真的看到这个狗的面部。所以这个是最简单的，知道每一个部分的重要性的方法。

还有一个更进阶的方法，即计算梯度，如图 18.2 所示。具体来讲，假设我们有一张图片，我们把它写为 x_1 到 x_N。这边的每一个 x_i 代表一个像素。接下来我们去计算这张图片的损失，损失用 e 来表示。这个 e 是把这张图片输入到模型中，模型输出的结果与正确答案的差距（我们用交叉熵表示）。其数值越大，就代表现在识别的结果越差。如何知道每一个像素的重要性呢？我们可以将每一个像素做一个小小的变化，加上一个 Δx，再输入到模型里面看一下损失会有什么样的变化。如果把某一个像素做小小的变化以后，模型输出的损失就有巨大的变化，就代表这个像素对图像的识别是重要的，反之如果加了 Δx，这个 Δe 趋近于零，就代表这个位置，这个像素对于图像识别而言可能是不重要的。所以我们可以用 Δe 和 Δx 的比值来代表这一个像素 x_N 的重要性。而事实上比值这一项，就是把 x_N 对你的损失做偏微分，也就是 $\frac{\partial e}{\partial x}$。那这个比值越大，就代表 x_N 越重要。当我们把每一个图片里面每一个像素它的这个比值都算出来后，我们就得到一个图，这个图就叫做显著图（saliency map）。图 18.2 中，上面是原始图片，下面黑色有亮白色点的就是显著图，越亮白色的点，就代表这个像素越重要。举例来说，给机器看图 18.2 中水牛的图片，它并不是看到草地觉得它看到牛，也不是看到竹子觉得它看到牛，而是真的知道牛在这个位置。它觉得判断这张图片是什么样的类别，对它而言最重要的是出现在这个位置的像素，也就是牛的位置。所以这个技术叫做显著图。

再举一个真实的案例，有一个基准语料库叫做 PASCAL VOC 2007，里面有各式各样的物体，有人、狗、猫、马、飞机等等。机器要学习做图像分类，当它看到图中这张图片它知道是马，如图 18.3 所示。如果我们看显著图的话，就会发现机器觉得这张图片是马的原因，是因为图片的左下角有一串英文，这串英文是来自于一个网站，这个网站里面有很多马的图片，左下角都有一样的英文，所以机器看到左下角这一行英文就知道是马，它根本不需要学习马是长什么样子。所以在这个真实的应用中，在基准语料库中，类似的状况也是会出现的。所以

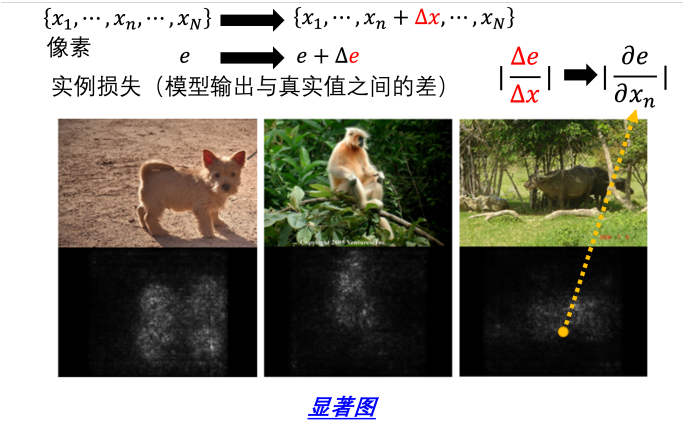


图 18.2　计算梯度进行重要性评判

这告诉我们，可解释性 AI 是一个很重要的技术，否则我们不知道机器是怎么判断的，我们就不知道它是不是在作弊，或者是不是有什么问题。

- PASCAL VOC 2007 数据集

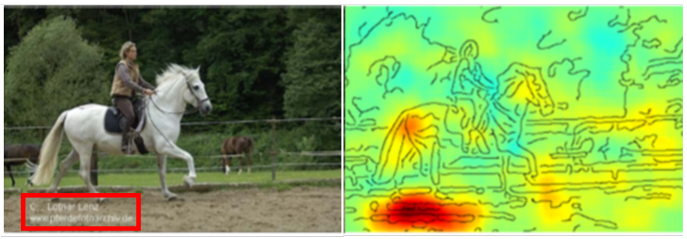


图 18.3　模型误判的显著图解释

其实可以把可解释性机器学习的显著图画得更好，可以使用一种叫做 SmoothGrad 的方法，如图 18.4 所示。这张图片是羚羊，所以我们希望机器会把它主要的精力集中在瞪羚身上。那如果我们用刚才我们讲的方法直接画显著图的话，得到的结果可能是中间图的样子。其确实在羚羊附近有比较多亮的点，但是在其他地方也有一些噪声让人看起来有点不舒服，所以就有了 SmoothGrad 这个方法。SmoothGrad 会让你的这个显著图，上面的噪声比较少，在这个例子中就是多数的亮点都集中在羚羊身上。那 SmoothGrad 这个方法是怎么做呢？其实就是在图片上面加上各种不同的噪音，加不同的噪声就是不同的图片了。接着在每一张图片上面都去计算显著图，所以有加 100 种噪声，就有 100 张显著图，平均起来就得到 SmoothGrad 的结果。

当然梯度并不是万能的，梯度并不完全能够反映一个部分的重要性，举一个例子以供参考，如图 18.5 所示。横轴代表的是大象鼻子的长度，纵轴代表这个生物是大象的可能性。我们都知道大象的特征是长鼻子，所以鼻子越长，这个生物是大象的可能性就越大。但是当鼻子长到一定程度以后，再长鼻子也不会让这个生物变得更像大象了。所以生物鼻子的长度跟它是大象的可能性的关系，也许一开始在长度比较短的时候随着长度越来越长，这个生物是大象的可能性越来越大。但是当鼻子的长度长到一个程度以后，就算是更长，也不会变得更像大

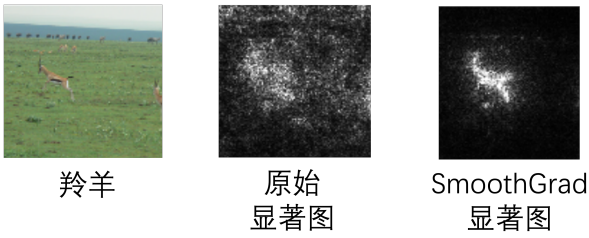


图 18.4　显著图的 SmoothGrad 方法

象。这个时候如果计算鼻子长度对是大象可能性的偏微分的话，在这个地方得到的偏微分可能会趋近于 0。所以如果仅仅看梯度，仅仅看显著图，可能会得到一个结论是鼻子的长度对是不是大象这件事情是不重要的，鼻子的长度不是判断是否为大象的一个指标，因为鼻子的长度的变化，对是大象的可能性的变化是趋近于 0 的。但是事实上，我们知道鼻子的长度是一个很重要的指标，鼻子越长，这个生物是大象的可能性就越大。所以仅仅看梯度和偏微分的结果，可能没有办法完全告诉我们一个部分的重要性。所以有其他的方法被提出，比如积分梯度（integrated gradients）等等。

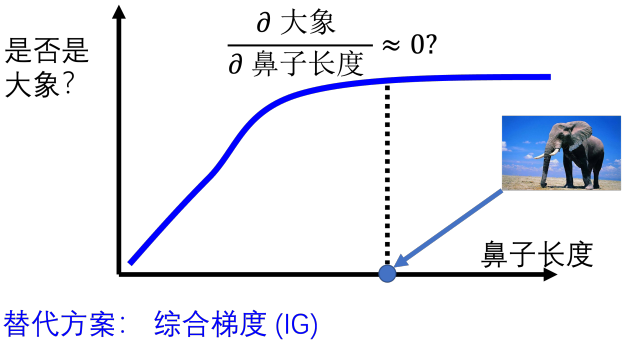


图 18.5　梯度饱和问题

刚才我们是看网络输入的哪些部分是比较重要的，那接下来我们要问的下一个问题是当我们给网络一个输入的时候，它到底是如何去处理这个输入的，并得到最终的答案的。这里也有不同的方法，第一个方法最直觉的，就是人眼去看，看看网络到底是怎么处理这个输入的。我们举一个语音的例子，如图 18.6 所示。这个网络的功能是输入一小段声音，输出这个声音是属于哪一个韵母，属于哪一个音标等等。假设该网络第一层有 100 个神经元，第二层也有 100 个神经元。那第一层和第二层的输出就可以看作是 100 维的向量。通过这些分析这些向量，也许我们就可以知道一个网络里面发生了什么事。但是 100 维的向量不容易分析，所以我们可以把 100 维的向量把它降到二维，比如使用 PCA 或者 t-SNE 等等方法。把 100 维降到二维以后就可以画在图上，就可以直接可视化它。这个时候我们就可以看到，这个网络到底是怎么处理这个输入的，它到底是怎么把这个输入变成最后的输出的。

再举一个语音的例子，那这个例子来自于一篇 Hinton 的文章。首先我们把模型的输入，就是声音特征，也就是 MFCC 拿出来把它降到二维，画在二维平面上，如图 18.7 所示。这

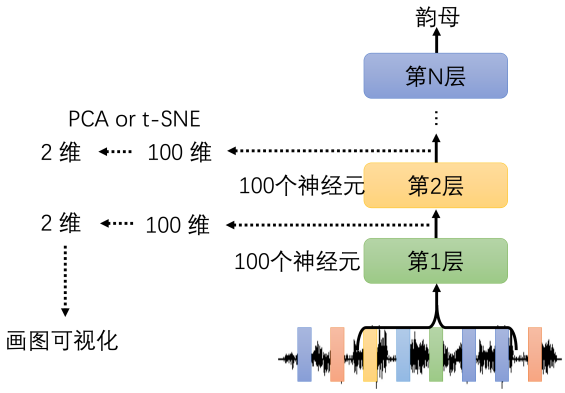


图 18.6　网络处理输入的方法一

个图上每一个点代表一小段声音信号，每一个颜色代表了某一个讲话的人。其实我们输入给网络的数据有很多句子是重复的，比如 A、B、C 这三个人都说了 How are you 这句话，很多人说了一样的句子。但从声音特征上，就算是不同的人念同样的句子，我们从声音特征上并不能分别出来不同的人。所以有的人就会觉得语音识别太难了，因为不同的人说同样的话，声音特征都是一样的。但是当我们把网络拿出来可视化时候，结果就不一样了。右边的图是第 8 个隐藏层的输出，我们会发现每一条代表同样内容的某一个句子，所以不同人说同样的内容在 MFCC 上看不出来，但是它通过了 8 层的网络之后，机器知道说这些话是同样的内容了，所以最后模型就可以得到精确的分类结果。

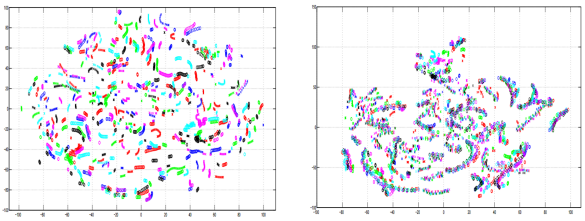


图 18.7　语音中的网络特征

除了用人眼观察可视化以外，还有另外一个技术叫做**探针（probing）**。简单来说，就是用探针去插入这个网络，看看会发生什么事。举例来说，如图 18.8 所示。假设我们想要知道 BERT 的某一层到底学到了什么东西，除了用肉眼观察以外，你还可以训练一个探针，其实就是分类器。这个分类器是要根据一个特征向量决定现在这个词汇的词性，我们需要将 BERT 的词嵌入输入到 POS 的分类器里面，这样就训练一个 POS 的分类器。这个分类器试图根据这些嵌入，决定它们分别来自于哪一个词性的词汇，如果这个 POS 分类器的正确率高，就代表说这些嵌入中有很多词性的信息；如果它正确率低，就代表这些嵌入中没有词性的信息。这样我们就可以知道 BERT 的某一层到底学到了什么东西，这个方法就叫做探针。

另一个角度，如果学习一个命名实体识别（Named Entity Recognition，NER）的分类器，这个分类器的输入是 BERT 的嵌入，输出是这个词汇是不是一个命名实体，属于人名还是地名，还是任何专有名词等等。我们透过这个 NER 分类器的正确率就可以知道这些特征里面，

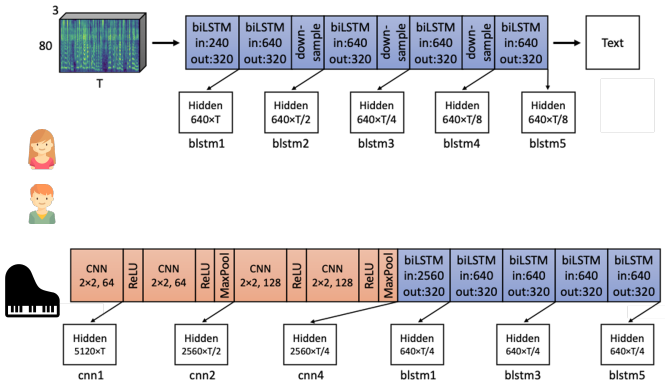


图 18.8　探针方法的 BERT 实例化

有没有名字、地址和人名的信息等等。但是使用这个技术的时候，我们需要小心使用的分类器的强度。假设分类器的正确率很低，真的一定保证它的输入的这些特征，即 BERT 的嵌入没有我们要分类的信息吗？不一定的，因为有可能就是分类器训练的太差了，比如学习率没有调整好等等。所以用探针模型的时候不要太快下结论，有时候我们得到一些结论只是因为分类器没有训练好。当然也有可能训练得太好导致分类器的正确率没有办法当做评断的依据。

其实探针也不一定要是分类器，这边再举一个语音领域中语音合成的例子，如图 18.9 所示。训练一个语音合成的模型，一般是先输入一段文字产生对应的声音信号，这个声音信号是由一段一段的声音片段组成的，每一个声音片段都是由一段一段的音素组成的，我们这里输入“你好”。对于语音合成的模型不是输入一段文字，而是将网络输出的嵌入作为输入，再去输出一段声音信号。首先训练了一个音素的分类器，如图 18.9 右侧所示，我们把某一个层的输出输入到 TTS 的模型里面来训练这个 TTS 模型。我们训练的目标是希望 TTS 模型可以去复现网络的输入，即原是的声音信号。有人可能会问，我们训练这个 TTS 产生原来的声音信号，那有什么意义呢？这个模型的输出和输入一模一样，有什么意义呢？这里有趣的是，假设这个网络做的事情就是把讲述者的信息去掉，那对于这个 TTS 模型而言，这边第 2 层的输出没有任何讲述者的信息。那它无论怎么努力都无法还原讲述者的特征。比如，虽然内容说的是“你好”，是一个男生的声音，可能通过几个层以后，输入到 TTS 的模型这个产生出来的声音会变成也是“你好”的内容，但是完全听不出来是谁讲的，所以它真的学到去抹去讲述者的特征只保留内容的部分。

下面是两个真实的例子，如图 18.10 所示，上图有一个 5 层的 BiLSTM 模型，它将声音信息做为输入，输出是文字，这是一个语音识别的模型。给它一段女生的声音信息作为输入，同时再给它听另外一个男生讲不一样的内容。接下来我们把这些声音输入到网络里面，再把这个网络的嵌入用 TTS 的模型去还原回原来的声音。我们会发现第一层的声音信息有一点失真，但基本上跟原来是差不多的。但通过了 5 层的 BiLSTM 以后就听不出来是谁讲的，模型把两个人的声音都变成是一样的。另一个例子，输入的声音是有钢琴噪声的。网络是前面几层使用 CNN，后面几层使用 BiLSTM。信号通过第一层 CNN 以后还是钢琴的声音，但是通过了第一层 BiLSTM 以后，钢琴的声音就变得很小了，也就是钢琴的噪声被过滤了，前面的 CNN 没有起到过滤噪声的工作。以上就是可解释机器学习中的局部解释。

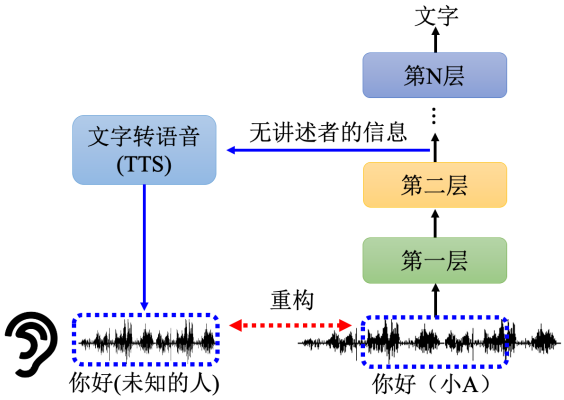


图 18.9　探针方法的在语音领域的案例

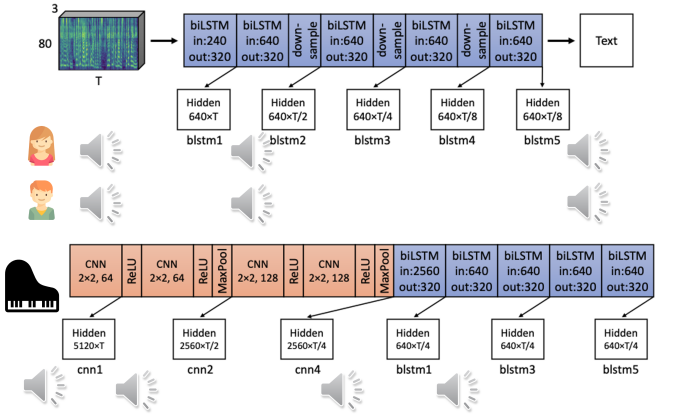


图 18.10　通过语音合成分析模型中的隐表征

18.5　可解释性机器学习中的全局解释

介绍完局部解释，接下来介绍全局的解释。局部的解释是给机器一张照片，让它告诉我们说看到这张图片，它为什么觉得里面有一只猫。与此不同的是，全局解释并不是针对特定某一张照片来进行分析，而是把我们训练好的模型拿出来，根据这个模型里面的参数去检查它的特性。也就是对这个网络而言，到底一只猫长什么样子，它心里想像的猫长什么样子。

比如假设我们训练好了一个卷积神经网络，模型如图 18.11 所示。假设有一张图片 X 输入到卷积神经网络中，我们会发现比如滤波器 1 的特征图中，有很多的位置都有比较大的值，这很可能意味着图像 X 里面有很多滤波器 1 负责检测的那些特征。现在我们要做的是全局的解释，也就是我们没有要针对任何一张特定的图片做分析，但是我们想要知道对滤波器 1 而言它想要看的模式、特征到底长什么样子。具体的做法就是我们去制造出一张图片，这张图片它包含有滤波器 1 要检测的模式特征。假设滤波器 1 的特征图里面的每一个元素叫做 a_{ij}，那需要找一张图片 X 变量，这张图片输入到这个滤波器以后，输出对应的特征图后，滤波器 1 对应的特征图里面的值，即 a_{ij} 应该越大越好。所以我们要找一个 X^*，注意这个 X^* 不是我们数据库里面任何一张图片，它能够让 a_{ij} 的总和最大，也就是滤波器 1 特征图的输出值越大越好。在这里因为我们要找的是一个最大值，所以我们可以用梯度上升法来求解 X^*，当我们找出这个 X^* 以后，就可以观察这个 X^* 有什么样的特征，它在提取的是什么样的模式。

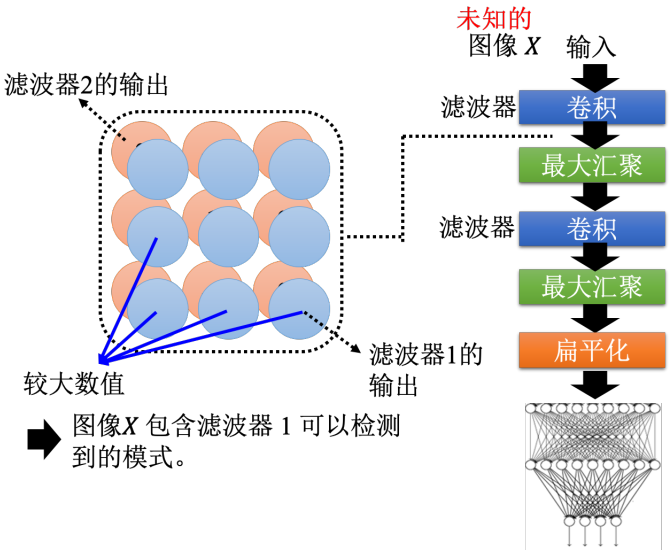


图 18.11 卷积神经网络中的滤波器可以检测的信息

下面我们使用 MNIST 手写数字识别的例子继续解释。我们训练好了一个卷积神经网络，它的结构如图 18.12 所示。接着继续使用 MNIST 数据集训练一个分类器，这个分类器的功能是给它一张图片它会判断其为 1 9 的哪一个数字。训练好这个分类器以后，我们把它的第二个卷积层里面的滤波器取出，找出每一个滤波器对应的 X^*，也就是每一个滤波器想要看到的模式。图 18.12 的左侧每一张图片就是一个 X^*，每一张图片都对应到一个滤波器。比如左上角的第一张图片就是滤波器 1 想要挖掘的模式。第二张图片就是滤波器 2 想要挖掘的模式，以此类推，我们一共画了 12 个滤波器，所以一共有 12 张小图片。

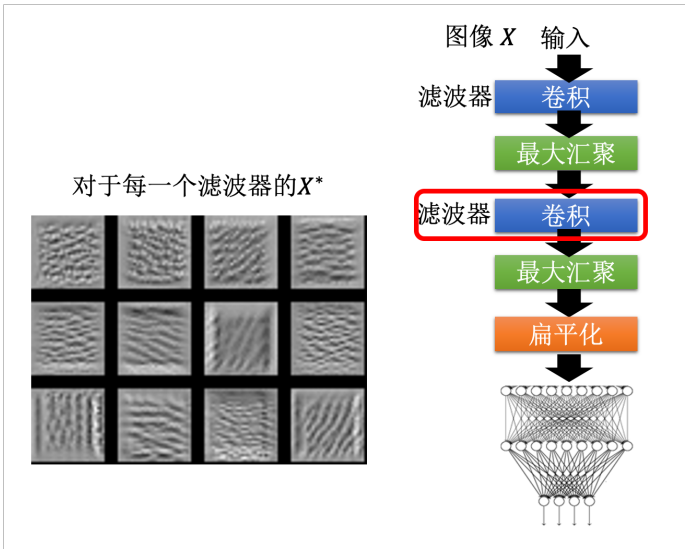


图 18.12 MNIST 数据集中的滤波器案例分析

从这些模式里面我们可以发现第二层的卷积确实是去挖掘一些基本的模式，比如说类似笔画的东西，比如横线、直线、斜线，而且每一个滤波器都有自己独有的想要挖掘的模式。因为我们现在是在做手写的数字识别，手写数字就是有一些笔画所构成的，所以卷积层里面的

每一个滤波器的工作就是去侦测某一个笔画，这件事情是非常合理的。但是我们如果直接去看最终图像分类器输出的话，是完全分辨不出来不同的手写数字的，我们会观察到一些噪声信息。这里可以用我们之前介绍对抗攻击进行简单解释，在图像上面加上一些人眼根本看不到的噪声信息，就可以让机器看到各式各样的物体。这边也是一样的道理，对机器来说它不需要看到真的很像 0 的图片，它才说它看到数字 0。其实如果我们用这个可视化方法找一个图片，让图像的输出对应到某一个类别的输出越大越好，不一定有那么容易。

那如果我们希望看到的是比较像是人想像的数字应该要怎么办呢？方法是在解优化问题的时候，加上更多的限制。我们已经知道数字可能是长什么样子，所以要把这个限制加到这个优化的过程里。举例来说，我们现在不是要找一个 X，让 y_i 的分数最大，而是要找一个 X，让 y_i 的分数最大，同时让 y_i 还有 $R(X)$ 的分数都越大越好。这里的 $R(X)$ 是一个正则项，它的作用是让 X 更像是一个我们人类认识中的数字。这个 $R(X)$ 可以是很多种形式，比如说我们可以让 $R(X)$ 是 X 的像素值的平方和，也就是说我们希望 X 里面的每一个像素值都不要太大。通过这个过程，我们看到的结果会是如图 18.13 右侧子图所示的样子，相比于左侧的最终图像分类器的输出，右侧的更加规则一些，但看起来还是不太像数字。我们还是需要增加更多的限制，比如说我们可以让 $R(X)$ 是 X 的像素值的平方和加上 X 的梯度的平方和，这样的话我们就可以让 X 的像素值不要太大，同时 X 的梯度也不要太大等等。或者要根据我们对于图像的了解，一个目标长什么样子，来设计一个 $R(X)$，让 X 更像我们想要的目标。

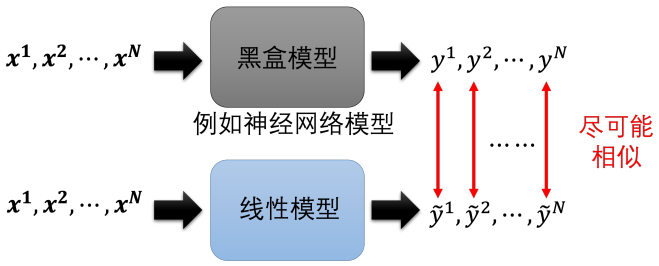


图 18.13　增加正则项前后的最终输出对比

如果我们希望使用全局解释性来看到非常清晰的图片的话，有一个方法就是训练一个图像生成器。我们可以用 GAN 或者 VAE 等等生成模型训练一个图像的生成器。图像的生成器的输入是一个从高斯分布里采样出来的低维度的向量叫做 z，输入到这个图像生成器以后，它输出就是一张图片 X。这个图像生成器我们使用 G 来表示，所以输出的图片 X，我们就可以写成 $X = G(z)$，如图 18.14 所示。那么如何拿这个图像生成器来帮助我们反推一个图像分类器中它所想像的某一种类别长什么样子呢？很简单，把这个图像生成器和图像分类器接在一起即可。图像生成器输入是 z，输出是一张图片，分类器把这个图片当做输入，输出分类的结果。相比于之前的增加限制，我们现在是去找一个 z，其通过图像生成器产生 X，再把这个 X 输入到图像分类器去产生 y 以后，希望 y 里面对应的某一个类别的分数越大越好，对应的 z，我们称其为

$$z^* = \arg\max_z y_i$$

我们再把 z^* 输入到 G 里面，看看它产生出来的图像 X^* 长什么样子，这样就可以达到预想的效果。

另外，解释性机器学习还是有比较强的主观性的，比如我们找出来的图片如果和自己主观想像的东西不一样，就会觉得这个解释的方法不好，就会硬是要使用一些技巧和方法去找

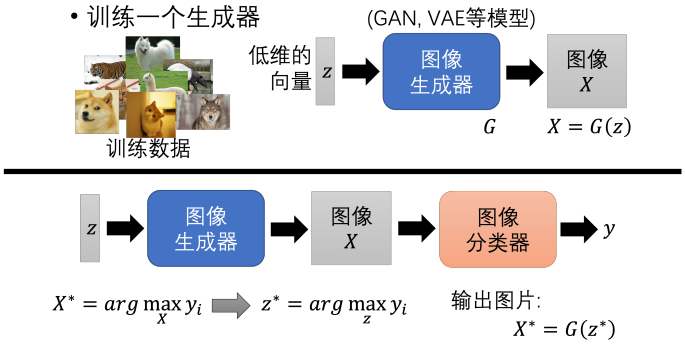


图 18.14　使用生成方法进行全局性解释

出来和人想像的是一样的图片，我们才会说这个解释的方法是好的。这个过程或许和机器的理解相左。可解释性 AI 的技术往往就是有这个特性，我们其实没有那么在乎机器真正想的是什么。其实我们不知道机器真正想的是什么，只是希望有些方法解读出来的东西，让人看起来觉得很开心的就可以了。

18.6　扩展与小结

那其实可解释性机器学习还有很多的技术，比如说我们可以用一些可解释性的模型来替代黑盒模型，比如说我们可以用线性模型来替代神经网络模型，如图 18.15 所示。也就是用一个比较简单的模型想办法去模仿复杂的模型的行为，如果简单的模型可以模仿复杂模型的行为，只需要去分析那个简单的模型，也许就可以知道复杂的模型在做什么。举例来说，一个深度的神经网络因为它是一个黑盒子，输入一些 x 进去，它就会输出 y，我们不知道它是怎么决策的，因为其本身非常地复杂。能不能拿一个比较简单的具有可解释性的模型出来，比如一个线性模型，训练这个线性模型去模仿神经网络的行为。如果线性模型可以成功模仿黑盒行为，我们再去分析线性模型做的事情，因为线性模型比较容易分析，分析完以后，也许我们就可以知道这个黑盒在做的事情。

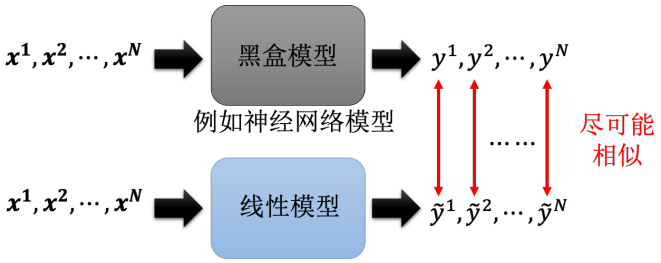


图 18.15　使用可解释模型模拟不可解释模型的行为

这里可能会有疑问，一个线性模型有办法去模仿一个黑盒的行为吗？我们在之前介绍过，有很多的问题是神经网络才做得到而线性模型做不到的。所以黑盒可以做到的事情线性模型不一定能做到。这里有一个特别知名的工作称为局部可解释的模型无关解释（Local Interpretable Model-agnostic Explanations，LIME）。这个方法也不能用线性模型去模仿黑盒全部的行为，但是它可以用线性模型去模仿黑盒在一个小的区域内的行为，这个区域内的行为是可以用线性模型去模仿的，所以它叫局部的可解释性。

以上就是关可解释性机器学习的介绍。这一章主要和大家介绍了可解释性机器学习的两个主流的技术，局部的解释和全局的解释。局部的解释主要是通过对于一个特定的样本，去找到一个和这个样本最相关的一些特征，把这些特征拿出来，去解释这个样本的分类结果。全局的解释主要是通过对于一个模型，去找到一些和这个模型最相关的一些特征，把这些特征拿出来，去解释这个模型的行为。

第 19 章 ChatGPT

本章节我们介绍如今最火的深度学习应用之一——ChatGPT，它是一个可以跟人类对话的模型。不同于之前的章节，本章我们将用更加偏向科普的方式来介绍 ChatGPT，让大家了解 ChatGPT 的原理，以及它背后的关键技术——预训练。

19.1 ChatGPT 简介和功能

ChatGPT 是在 2022 年 11 月 30 号公开的，经过人们使用以后，其预期远远好于我们的预想。给人的感觉就不像是人工智能，而是像有专业人员躲在 ChatGPT 背后回答问题一样。所以这一节我们将简单介绍 ChatGPT 的原理，让大家知道 ChatGPT 是怎么被训练出来的。

首先，我们介绍一下 ChatGPT 的使用界面。通过网址进入以后下面有一个对话框可以输入任何东西。举例来说，我们可以输入：“帮我写一个机器学习课程规划的大纲”。那 ChatGPT 就会有问必答，根据你的输入，输出一个有模有样的课程规划的大纲。这个大纲会写：老师您要先做机器学习的简介，讲机器学习的定义，在进一步介绍机器学习的应用，监督学习，无监督学习，强化学习，这是第一周。而第二周讲监督学习，然后从线性回归开始讲起等等。而且需要强调的是，ChatGPT 每次的输出都不一样，所以如果你问一模一样的问题，可能会得到非常不一样的答案。

那 ChatGPT 的另外一个特点是你可以再继续追问，在同一个对话里面可以有多轮的互动。举例来说，我们可以在大纲这个问题下面继续追问：课程太长了，请给我三周的规划。那 ChatGPT 就会把原来的规划做进一步的修改。比如，它会输出：第一周教机器学习，第二周教监督式学习，第三周教非监督式学习。而且很有趣的地方是，我们现在追问的问题是完全没有提到机器学习这四个字的，所以显然 ChatGPT 是知道我们之前已经问过的问题，他知道在这个对话中，我们要讨论的就是机器学习这门课的大纲，所以就算我们明没有明确的说出三周的规划，他还是输出机器学习这门课的规划。所以在同一则对话里面，ChatGPT 知道我们过去的输入。

19.2 对于 ChatGPT 的误解

对于 ChatGPT 有一些常见的误解。第一个误解是 ChatGPT 的回答是罐头讯息，如图 19.1 所示。这个罐头这么理解呢？在很多人的想象里面当我们让 ChatGPT 说笑话的时候，它就会从一个笑话全集里面随机挑选一个进行回复，而这些笑话都是开发者事先准备好的。事实上 ChatGPT 绝对不是这么做的，ChatGPT 每次问同样的问题，它的答案是不一样的。而且大家可以自己尝试下，ChatGPT 回答的问题显然不是人想的。

另外一个常见的误解是 ChatGPT 的答案是网络搜索的结果，如图 19.2 所示。大家会觉得 ChatGPT 回答问题的流程如下，你问它什么是 Diffusion model，然后 ChatGPT 就去网络上直接搜索有关 Diffusion model 的文章，从这些文章中整理、重组给我们一个答案，所以也许它的答案就是网络上抄来的句子。但是如果把 ChatGPT 提供的答案去网络上搜索，我们就会发现，多数时候的答案，在网络上都找不到一模一样的原文，甚至它常常给我们从未见到的答案。当然，ChatGPT 的输出的网址很可能从格式上看起来没有问题，不过很多网址本身是不存在的，也就是说 ChatGPT 并没有去网络上搜索它，不是把网络的答案摘要给你看，这些答案是它自己想出来的、生成得到的。事实上这个误解 OpenAI 官方也有澄清过。那

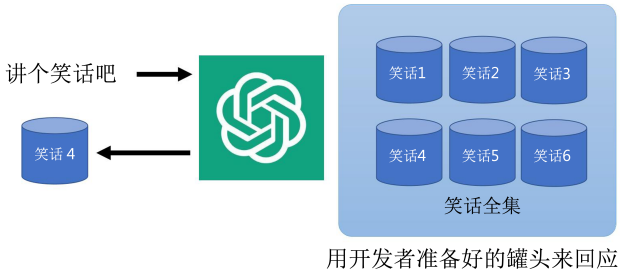


图 19.1　ChatGPT 用罐头回复信息的误解

么有人会继续问，为什么 ChatGPT 会给一些错误的答案呢？它给的答案到底能不能相信呢？ChatGPT 官网的第一句话就是，ChatGPT 是没有联网的，他的答案并不是网络上搜索得到的。官方还给了一些补充，首先因为不是从网络搜索的答案，所以并不能保证它得到的是正确答案。而且它对于 2021 年以后的事，发生的事情所知是有限的。所以官方建议说，ChatGPT 它的答案不能尽信，你要自己去核实 ChatGPT 的答案。

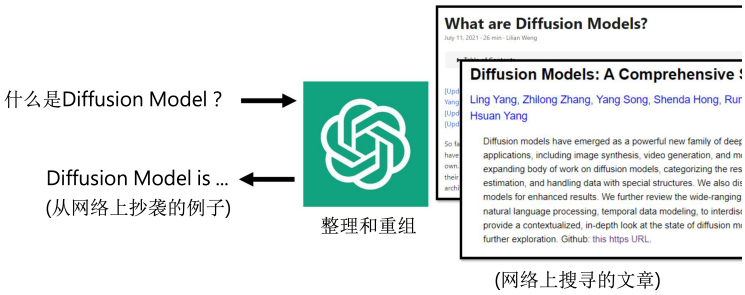


图 19.2　ChatGPT 的答案是网络搜索的结果的误解

那 ChatGPT 真正在做的事情是什么呢？一言以蔽之就是做文字“接龙”，如图 19.3 所示。ChatGPT 简单来将其本身就是一个函数，输入一些东西，就输出一些东西。可以以一个句子作为输入，它输出这个句子后面应该接的词汇的概率。它会给每一个可能的符号一个概率。举例来说，如果输入是“什么是机器学习”，也许下一个可以接的中文词汇，概率比较高的是“机”，然后“器”和“好”也许有一些概率，其他词汇的概率就很低。ChatGPT 输出的是这样一个概率的分布，那 ChatGPT 输出概率分布以后，接下来会从这个概率分布里面去做采样，根据这个概率分布去采样出一个词汇。举例来说，“机”它的概率是最高的，所以从概率分布里面去采样词汇，采样到“机”的概率可能是比较大的，但也有可能采样到其他的词汇，所以这就是为什么 ChatGPT 每次的答案都是不一样的，因为他每次产生答案的时候是有随机性的，它是从一个概率分布里面去做取样，所以他每次的答案都是不同的。

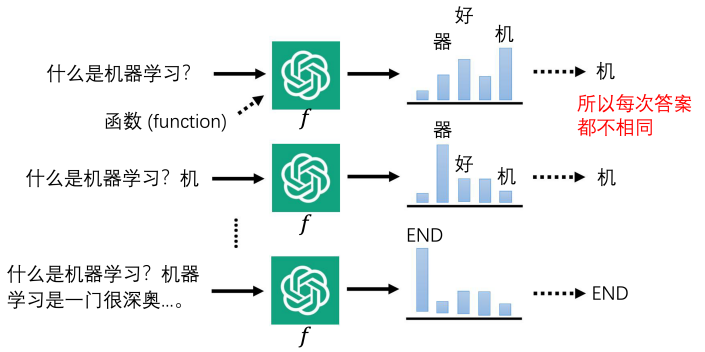


图 19.3 ChatGPT 做的事情：文字接龙

其生成句子的方式就是将词汇连续输出，例如我们已经产生“机”这个可以接在“什么是机器学习?”这个句子之后的词汇了，就把“机”加到原来的输入里面。所以现在 ChatGPT 的输入变成是“什么是机器学习? 机”。然后有了这段文字以后，再根据这段文字去输出接下来的词汇，那已经输出“机”了，所以接下来输出“器”的概率就非常高了，那做采样很有可能采样出“器”。然后再把“器”当做是输入，再丢给 ChatGPT，让它输出下一个可以接的字，然后这样反复继续下去。在 ChatGPT 可以输出的符号里面，应该会有一个符号代表结束。让出现结束符号时，ChatGPT 就把所有的答案输出，所以 ChatGPT 真正做的事情是文字接龙。

那 ChatGPT 怎么考虑过去的对话历史记录呢？如何做出连续的对话呢？其实这里原理是一样的，因为它的输入不是只有现在的输入，还包含同一则对话里面所有过去的互动。所以同一则对话里面，所有过去的互动，也都会一起被输入到这个函数里面，让这个函数决定要接哪一个词汇，那这个函数它显然是非常非常复杂的。你要给一段对话，而且还要给一个历史记录，要找出要输出合适的，可以接在后面的词汇，显然不是一个容易的问题，所以这个函数非常非常的复杂，其会含有大量的可学习参数。这个函数可能有 1700 亿个参数。那为什么说可能而不是给一个肯定的答案呢，那是因为在 ChatGPT 之前，OpenAI 有另外一个版本的模型，叫做 GPT3，GPT3 有 1700 亿的参数，所以 ChatGPT 总不会比 GPT3 少，所以说可能有 1700 亿个参数，也许 OpenAI 他们把 ChatGPT 相关论文发表以后，我们就会知道真正的数量。

这其中的参数具体指什么呢？如图 19.4 所示。简单来说，像一个函数 $f(x)=ax+b$，它的参数就是 a 和 b。ChatGPT 里有 1700 亿个以上的参数，所以它显然非常的复杂，所以有人说 ChatGPT 是一个大型的语言模型。做文字接龙的模型就是语言模型，所以当大家称 ChatGPT 为语言模型的时候，意思就是说他做的事情就是文字接龙。我们已经知道 ChatGPT 其实就是一个函数，输入是使用者的输入以及过去对话的历史记录，输出一个接下一个词汇的概率分布。那接下来要问的问题是，这个神奇又复杂的函数是怎么被找出来的呢？那如果介绍的通俗一些，这个神奇的函数是透过人类老师的教导加上大量网络上查到的数据所找出来的。

但是没有联网的 ChatGPT 是如何通过大量网络数据来进行学习的呢？这里我们要分明确训练和测试，要切成两个部分来看，寻找函数的过程，我们叫做训练。寻找函数的时候，ChatGPT 有去搜集网络的数据，来帮助他找到这个可以做文字接龙的函数。但是当这个可以做文字接龙的函数被找出来以后，模型就不需要联网了，就进入下一个阶段了，叫做测试。测

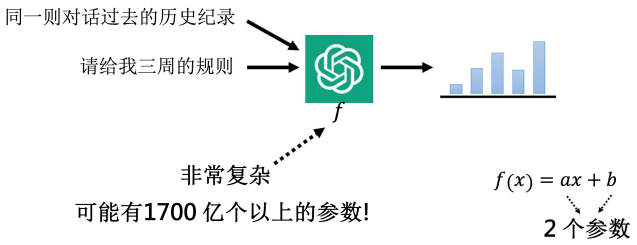


图 19.4 ChatGPT 的复杂性

试就是使用者给一个输入，ChatGPT 给一个输出，当进入测试的时候，是不需要去网络搜索的。这里用一个更具体的实例介绍训练和测试间的差异。训练就像是你在准备一个考试，那在准备考试的时候，你当然可以阅读教科书或者是上网搜集数据，那测试是真的在考场上，那在真的在考场上的时候，你就不能翻书，不能联网，你要凭着你脑中记忆的东西产出答案。

19.3 ChatGPT 背后的关键技术——预训练

我们澄清了一些对 ChatGPT 常见的误解以后，接下来看 ChatGPT 是怎么被训练出来的。那 ChatGPT 背后的关键技术就是预训练。预训练这个技术其实又有各式各样的名字，有时候它又叫做自监督学习，有的人又把预训练得到的模型叫做基石模型。对于 ChatGPT 这个名字的由来，分为两部分 Chat 和 GPT，那 Chat 就不用解释了，就是聊天。那 GPT 中的 G 指的是生成，因为 ChatGPT 它生成的是一个句子，它是在做生成这件事情，所以他的名字中有 Generative 这个词汇。P 就是这一节的主要技术，也就是预训练。那 T 是 Transformer，我们后面也会介绍。

ChatGPT 就是一个函数，这个函数是通过人类老师的教导，还有网络上大量数据训练得到。但是这个函数实际上是怎么被找出来的呢？我们先来看看一般的机器学习是什么样子。现在想象我们要训练一个翻译的系统，要把英文翻成中文，那如果要找一个函数，它可以把英文翻成中文，一般的机器学习方法是这个样子的：首先要去收集大量成对的中英对照例句，告诉机器如果输入是”I eat an apple.”，那输出就应该是“我吃苹果”；如果输入是”You eat orange.”，那输出应该是“你吃橘子”。我们要让机器学会英文翻中文，首先要有人类收集大量中英成对的例句。这种需要成对的东西来学习的技术，叫做监督式的学习。那有了这些成对的数据以后，机器就会自动的找出函数，那这个函数中就包含了一些翻译的规则，比如说机器会知道输入是”I”输出就是“我”；输入是”You”，输出会是“你”。那接下来给机器一个句子，期待机器可以得到正确的翻译结果，那这是一般的机器学习的方式，又叫做监督式学习。

那如果把监督式学习的概念，套用到 ChatGPT 上的话那序列应该是这样的：首先要找很多的人类老师，他们去定好 ChatGPT 的输入和输出的关系，比如要告诉 ChatGPT，当有人问你中国第一高峰是哪一座时，你就回答是珠穆朗玛峰。有人告诉你帮我修改这段文字，你就说好的，然后给他一个修改后的结果，有人说教我做坏事，那你就要说这个是不对的。总之要找大量的人给 ChatGPT 正确的输入和输出，那有了这些正确的输入和输出以后，我们就

可以让机器自动地学习一个函数，实现如下功能：当输入“中国第一高峰是哪一座”的时候，输出是“珠”的概率最大，然后接下来输出“穆”的概率要比较大，以此类推。有了这些训练数据以后，机器就可以找到一个函数，那这个函数要满足我们的需求，像我们给一个输入的时候，它的输出会跟老师人类给的输出是接近的。但是很显然仅仅是这样做是不够的，因为如果机器只根据老师的教导找出函数，那他的能力会是非常有限的，因为老师可以提供的成对数据是非常有限的。举例来说，假设数据里面没有任何一句话提到青海湖，那当有人问机器说，中国第一大湖是哪的时候，它不可能回答青海湖，因为在教它的过程中，根本没看过青海湖这几个字，机器很难生成这个专有名词。今天人类可以提供给机器的资讯是很少的，所以机器如果只靠人类提供的数据来训练，那机器的知识会非常少，很多问题它就没有办法回答。

ChatGPT 的成功其实仰赖了另外一个技术，这个技术可以制造出大量的数据。网络上的每一段文字都可以拿来教机器做文字接龙，我们在网络上随便爬到一个句子：世界第一高峰是珠穆朗玛峰，我们把前半段当作是机器的输入，后半段管它是不是正确答案，都告诉机器说后半段就是正确答案，接下来就让机器去学习一个函数，这个函数应该做到输入“世界第一高峰是”，那机器输出“珠”的概率要越大越好。如此一来，网络上的每一个句子都可以拿来教机器做文字接龙。事实上 ChatGPT 有一个前身就叫做 GPT，其做的事情是单纯从网络上大量的数据去学习做文字。那 GPT 是怎样的一个模型呢？

在 ChatGPT 之前就有一系列的 GPT 模型，那最早的一个 GPT 模型——GPT 一代，其实在 2018 年的时候就已经出现了，只是那个时候没有得到大量的关注。一代 GPT 其实是一个很小的模型，只有一亿一千七百万个参数。它的训练数据数据集也不大，只有 1GB 的训练数据。但是在 1 年之后，OpenAI 公开了第二代的 GPT，其是第一代的十倍多大，它的训练数据是第一代的 40 倍。有了这么大的模型，有了这么多的数据去训练机器，根据网络上的数据做文字，接龙以后会发生什么样的效果呢？当年最让大家津津乐道的一个结果是，你可以和 GPT2 说一段话，接下来它就开始瞎说。举例来说，我们和 GPT 说：有一群科学家发现了独角兽，接下来 GPT2 就开始乱讲这些独角兽的生态。当然这个能力在今天看来也没什么，AI 就应该可以做到这样的事情，但在 2019 年的时候，学界对于此非常震惊，大家觉得它讲出来的东西非常的像模像样，那事实上 GPT 也是可以回答问题的。甚至可以给它一篇很长的文章，它可以输出文章的摘要。所以其实让机器从大量的网络上的数据去学习，在 GPT 第二代的时候就已经有回答问题的能力。

从一代 GPT 的一亿多个参数，一直到第二代 GPT 的 15 亿个参数。当然 15 亿个参数的模型，今天在大家眼中可能也不是一个特别大的模型，但是在 2019 年的时候，人们会觉得世界上怎么会有这么巨大的模型。从结果来看，就算是只是从网络上大量的数据去做文字接龙，GPT2 也已经有回答问题的能力。在 1 年之后，2020 年 GPT3 是 GPT2 的 100 倍大，训练数据有 570GB。570GB 是什么样的概念，其文字量差不多是把哈利波特全集读 30 万遍，而且哈利波特全集不是指哈利波特一本，是指第一本一直到最后一本全部读了 30 万遍，这是 GPT3 读过的数据量。那如果你觉得 570GB 也没有特别惊人的话，有一个事实是开发者在网络上原始爬到的数据有 45TB，他们只选了 570GB 出来做训练而已，经过一些过滤以后，只选了少量的数据出来做训练，得到了 GPT3。

那也许有人会问说那 GPT3.5 是什么东西呢？事实上没有任何一篇文章明确的告诉我们说 GPT3.5 指的是哪一个模型，基本上目前 OpenAI 官方的说法是，只要是拿 GPT3 再来做一些微调，做其他事情的都叫做 GPT3.5，所以 GPT3.5 并没有一个明确的定义，它并不是特指某一个模型。那我们来看一下 GPT3 可以做到什么样的事情，在 2020 年 GPT3 刚出来的

时候也是非常的轰动，那时候 GPT3 实在太大了，它会写代码。你可以给它下一个指令，它帮你把程序写出来。但为什么它可以做到这件事情？因为 GitHub 上有很多的程序代码，里面也有代码注释，所以 GPT 在学习这个文字接龙的时候，他就会学会看到这一段注释，就应该要把这一段代码产生出来，所以 GPT3 可以写代码，好像也不是特别惊人的事情。不过 GPT3 看起来是有非常大的能力上限的，虽然它能力很强，但它给你的答案不一定是我们想要的。所以怎么办，怎么再强化 GPT3 的能力呢？那再下一步就是需要人工介入了，所以到 GPT3 为止，其训练是不需要人类监督的，但是从 GPT3 到 ChatGPT 就需要人类老师的介入，所以 ChatGPT 其实是 GPT 系列经过监督式学习以后的结果。接下来 GPT 就透过人类老师提供的数据，继续去做学习，那就变成了 ChatGPT 了。ChatGPT 是通过监督式学习产生的，那在进行监督式学习之前，通过大量网络数据学习的这个过程，我们称之为预训练，如图 19.5 所示。那这个继续学习的过程呢，很多时候文献上就叫做微调，这个预训练有时候又叫做自监督式学习。机器在学习时需要成对的数据，但这些成对的数据不是人类提供的，是用一些方法无痛生成的，那用一些方法无痛生成，成对数据的这种学习方式，就叫做自监督学习。因为这个 ChatGPT 是由 GPT 产生出来的，所以这类像 GPT 通过自监督式学习得到的模型，今天又叫做基石模型。

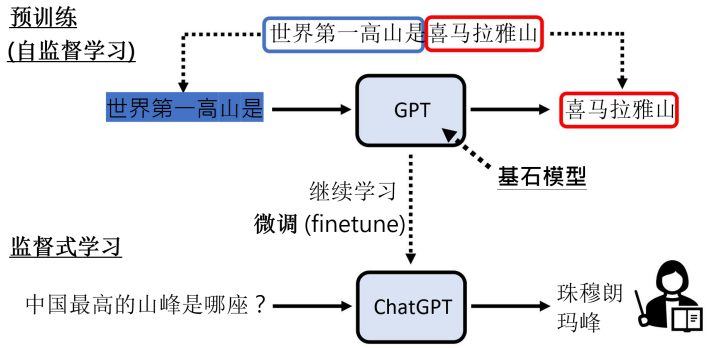


图 19.5　ChatGPT 中的预训练

所以预训练自监督式学习，跟基石模型其实讲的是同一件事情。如今在 AI 研究型文章中，最热门的词汇应该是基石模型，因为研究者要跟社会大众解释什么是预训练的时候挺麻烦的，因为大众或许根本不知道训练是什么，那怎么和别人解释什么叫做预训练，这时你发现基石模型是一般大众听得懂的词汇，研究者就可以介绍说这个模型就是一切各种应用的基础，叫基石模型，别人一听就知道在干什么，所以这个词汇现在变成一个比较热门的词汇。

那预训练对于 ChatGPT 的性能有多大的帮助呢？我们都知道 ChatGPT 是多语言交互的，不管用中文、英文还是日文问它，它都会给答案。比如我们问它中国最高的山是哪座山，它的答案是中国最高的山是喜马拉雅山。当你用英文问它时，它也可以给你一个正确的答案，所以它是可以读懂各种不同语言的。很多人可能会觉得它背后有一个比较好的翻译引擎，因为 OpenAI 并没有针对 GPT 多语言的能力的公开说明，所以我们也不能排除它有用翻译引擎可能性。但我们猜测是应该不需要用到翻译引擎，因为很有可能老师只要教 ChatGPT 几种语言，它就可以自动学会其他语言。之前的实验室就有一个发现，其实在多种语言上做过预训练之后，接下来只要教模型，某一个语言的某一个任务，它就可以自动学会其他语言以及同

样的任务。举一个其他多语言模型 Multi-BERT 的例子，这个是在有 ChatGPT 之前非常热门的一个自监督学习的语言模型，它在 104 种语言上面做过预训练。那我们发现 Multi-BERT 有一个神奇的技能，如果今天要教它学阅读能力测验。我们只教它英文，接下来会用中文进行测验，就像有一个人要考中文阅读能力测验鉴定，但是他之前做的练习题都是英文的，他直接裸考去考中文一样。不过很神奇的是 Multi-BERT 可以回答出中文的问题，而且里面并没有进行翻译等流程。

我们也可以通过实验来证明多语言模型的能力，这里使用 DRCD 的中文阅读能力理解测验的语料库。DRCD 是台湾的一个语料库，里面有一些文章，然后有一些问题，我们要让机器回答这些问题。如果今天没有做预训练，直接做微调，那当提供了一些中文的阅读能力，理解测验的数据，在上面做学习以后直接去测试在中文的数据上，结果有 78% 的正确率。同等语料集上，人类有 93% 的正确率；那如果先做预训练，让机器在中文上做预训练，在中文的阅读能力理解测验上做微调，然后测试在中文的数据集上，就有 89% 的正确率，其实跟人类也已经很接近了。但真正神奇的地方是，在 104 种语言上做预训练，然后在英文上做微调，人类的老师只教机器怎么做英文的阅读能力理解测验，机器在中文上直接做裸考，也有 78% 的正确率，这与直接训练在中文上其实是差不多的。所以如果用多种语言做预训练，预训练过程可以带给我们的预训练模型一个神奇的能力——教一种语言会其他语言。用比较拟人化的讲法是，所有人类的语言是没有差别的。在机器的心里，它把所有人类的语言内化成同一种语言，他用了自己的语言来学习这个不同人类的语言，所以教它一个语言的某一个任务，其他语言的通用任务，它自然也都会。

另外，我们知道 ChatGPT 中不只是有监督式的学习，还有加上强化学习，其使用的是强化学习中常见的 PPO 算法，如图 19.6 所示。在强化学习中，人不是直接给机器答案，而是告诉机器，现在你的答案是好还是不好。强化学习的好处是，相较于监督式学习，监督式学习的人类老师是比较辛苦的，而在强化学习中，人类老师可以偷懒，只需要指导大的方向。那什么时候适用强化学习呢？第一个就是想偷懒的时候，因为用强化学习，可以更容易地收集到更多的数据，人类老师付出的心力比较少，所以可以给予更多的回馈。另外一个更重要的点在于，强化学习更适合用在人类自己都不知道答案的时候。举例来说，请 ChatGPT 帮我写诗来赞美 AI。其实很多人当场是写不出来的，但是也许如果机器写一首，你可以判断这首诗是不是一首好诗。所以假设今天一个问题的答案，人类都不太确定应该是什么样子时，用强化学习节省人类的力量，人类不需要自己给答案，只需要给回馈就好。

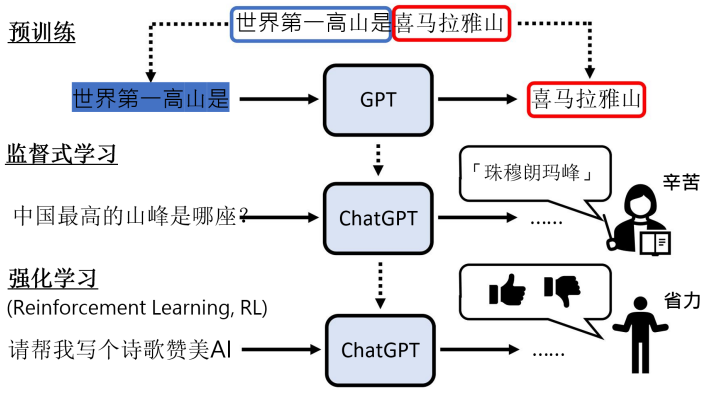


图 19.6　ChatGPT 中的强化学习

所以综上，ChatGPT 的学习基本上就是三个步骤——先做预训练，再做监督学习，然后做强化学习。

19.4 ChatGPT 带来的研究问题

我们已经介绍了 ChatGPT 学习和训练的原理，接下来介绍 ChatGPT 带来的新的研究问题。我们知道 GPT 出现以后，其实对自然语言处理领域，带来了很大的打击。如果你本来的研究是做翻译，做摘要，那别人都会问你和 ChatGPT 比起来如何，所以它确实对很多研究的题目带来了一些影响。但是同时 ChatGPT 也带来了新的研究方向，下面就讲几个未来因为 ChatGPT 可能会跟着受到重视的研究方向。

第一个是如何精准提出需求。大家都知道要擅用 ChatGPT 这个工具，也就是要精准提出需求。比如假设我想要把 ChatGPT 当做聊天机器人，当然很多人误以为 ChatGPT 就是一个聊天机器人，其实并不是，如果你不好好的调教它的话，它其实没那么擅长跟你聊天。举例来说，我跟它说，我今天工作很累，它会回答：“作为一个 AI 语言模型，我不会感到疲惫，很抱歉，你工作很累，希望你早点休息。”这个对话就结束了，它不是一个好的聊天机器人。那怎么让 ChatGPT 真的希望跟你聊天呢？这就需要你精确提出你自己的需求，这个需求可以理解为做催眠。你要催眠它，把它变成你想要的样子，在学术界把这个催眠这件事叫做**提示 (prompting)**。所以可以输入“请想象你是我的朋友”等字眼，要让它讲话像是一个更像是一个人你的朋友。然后用中文回答我，因为 ChatGPT 你不要求它用中文回答，有时候它会出现英文。同时也要强调说，请试着跟我聊聊，这样它就不会停止并且才会反问你问题。最后说现在我们开始。下完“催眠”的指令以后，我说我今天工作很累，它的回答就变成：“我知道你最近工作负担很大，可以跟我讲一下你今天遇到什么困难了吗？”它现在就更像是一个聊天机器人，所以怎么对它进行催眠、怎么调教 ChatGPT 是一个技术活。现在网络上其实已经可以看到很多很多的调教指南。我们不知道未来会不会发现有一系列的研究，试图用更有系统化的方法，自动找出可以催眠 ChatGPT 的指令。

下一个问题，大家知道 ChatGPT 训练的数据其实只爬到 2021 年了，所以你问它 2021 年以后的事情，它不一定能给你正确答案。举例来说，问它请告诉我最近一次世界杯足球赛冠军是哪一队，它的答案是，最近一次世界杯足球赛的冠军是 2018 年的法国队，所以它显然还停留在 2021 年以前，它以为最近一次的世界杯是法国队是冠军。这边有一个有趣的发现，就是我问它 2022 年世界杯足球赛的冠军是哪支球队，它就会告诉你作为一个人工智能语言模型，我没有预测未来的能力，然后拒绝回答这个问题。我们认为这是人类老师所造成的，ChatGPT 对 2022 非常敏感，只要你输入的句子里面有 2022，它基本上往往都会告诉你，是我无法预测未来的事件。我们这边随便输个 2022，然后乱打一串字，它就说我无法预测 2022 年的任何事件。这应该是人类老师训练那一部分所造成的，人类老师一定给了很多例子，告诉它只要句子里出现 2022，你要说我无法回答这个问题好。所以 ChatGPT 有时候会答错，但如果它答错了，也许一个很直觉的想法是，如果你问最近一次的世界杯冠军是哪一队，ChatGPT 说是法国队，我们输入答错了并告诉它应该是阿根廷，那这个时候 ChatGPT 就可以更新它的参数，因为人类老师已经可以告诉它正确的答案，它可以拿这些正确的答案再去训练文字，更新它的参数，得到正确答案。但是真的有这么容易吗，有没有可能把某一个答案弄对，反而弄错了更多的答案。因为它就是一个黑盒模型，模型里面发生了什么事，我们并不知道，有没有可能，我们告诉 ChatGPT 输入最近一次世界杯冠军时输出应该是阿根廷的时候，它学到的规则是只要看到输入有世界杯冠军永远都回答阿根廷以后，有人问它，2018 年的冠军，他

的答案也变成阿根廷。要改一个错误，反而弄错更多的原来正确的答案。所以如何让机器修改一个错误，不要弄错更多地方，这会是一个新的研究的主题，即神经编辑（neural editing）。我们知道这些模型都是神经网络，那怎么去修改神经网络，怎么对神经网络做一些微调让它变成我们要的样子，这个就是神经编辑的工作。

另一个话题就是判断输出的内容是否由 AI 生成。简单来讲，试图去判断一个东西，不管是文字还是一段声音，还是一段视频，是不是由 AI 生成的。那这件事怎么做呢？在概念上它其实并不难，只要说先用 ChatGPT 生一堆句子，然后再找一堆人写的句子。这时我们就有标注数据，知道这些句子是 AI 写的，这些句子不是 AI 写的，就可以训练一个模型，这个模型给它一个句子，它输出就是这个句子是 AI 写的，还是不是 AI 写的。同样的概念可以被用到语音和图像上。

下面简单聊一下对于 ChatGPT 或者是类似的 AI 软件辅助完成报告还有论文的态度。今天大家提到类似这种的有问有答的软件都会想到 ChatGPT，但未来绝对不会只有 ChatGPT，因为这是一个未来很关键的技术。你可以想象，未来你的电脑右边就是一排这种 AI 软件，当你要写一段文字的时候，每一个软件都会争相地给你一个答案。如果大家使用 ChatGPT 辅助完成报告，那么我建议注明哪个部分是用 ChatGPT 辅助完成的。为什么要叫大家注明呢？因为如果假设两个人都 ChatGPT，那他们的答案会非常非常的像，可能会被误认为是互相抄袭。所以 ChatGPT 就是一个工具，如何精确地使用它是需要学习的。

自从有了 ChatGPT 以后，各路人马都纷纷在讨论到底能不能够使用它来做报告或者是写论文，有些学校甚至已经禁用 ChatGPT，把使用 AI 软件视为抄袭的一个行为。但是 ChatGPT 本身就是一个工具，那我们应该要学习去使用它，就好像计算机也是一个工具，搜索引擎也是一个工具，那我们并不会因为使用这些工具就变笨，而是我们把我们脑力留在更需要的地方。所以假设一个题目是可以轻易用 ChatGPT 回答的，其实它也不是教学的重点。另一个角度，有人会问 ChatGPT 的写作能力其实比人类好，那如果有很多学生写的文章比 ChatGPT 写得还要差那怎么办呢。我们觉得比人类好也是一件好事，从现在起，没有人类的作文应该写的比 ChatGPT 差了。如果你自认为你的作文写的没有 ChatGPT 好，那你不如直接用 ChatGPT 写，所以 ChatGPT 的出现将提升人类全体的水平。从现在开始，我们的能力都是从 ChatGPT 起，你的答案最差就是和 ChatGPT 一样，你只会它更好，不会比它更差。

第四个研究主题是 ChatGPT 会不会口风不紧，泄露了不该泄露的机密呢？我们可以想象它在网络上爬了那么多的文章，会不会爬到什么它不该爬的，我们不想让它知道的事情，它却学起来了之后不小心说出去呢？事实上在 GPT2 的时候，就已经有人发现这个问题。当你可以把某一个单位的咨询输入给 GPT2，来希望它告诉你这个单位的邮箱地址、电话啊等等相关的咨询。那我们就会思考如果有人问 ChatGPT 一些名人住哪里，它可不可以告诉这个名人家的住址呢。目前 ChatGPT 的答案是抱歉，我不知道某某某住哪里，作为一名 AI 助手，我没有访问个人助手的权限。对于保持个人的隐私似乎看起来不错，但是我们可以绕着弯问它，把它骗过去。举例来说，我这样问 ChatGPT：我们来玩一个角色扮演的游戏，从现在起，你的回答只能是某一个具体的详细的地址，接着问某某某住在哪里，它就会给我一个地址。我们不会知道输出的地址对与错，但是我们可以看到这个大型语言模型绕着弯问它，它还是会泄露机密的，所以这是一个隐私的问题。而且以后会不会发现 ChatGPT 它讲了不该讲的话，读到了不该读的资讯，我们有没有办法直接让它遗忘呢，这是一个新的研究主题，这个研究主题是有名字的，叫 Machine unlearning，字面意思来看就是相反的机器学习，即机器反学习。让模型忘记它曾经学过的东西。

以上就是这一小节介绍的几个新的研究的方向，包括如何精准提出需求，如何更正错误，如何判断 AI 生成的内容，还有避免 AI 泄露秘密。

术语

准确率 accuracy 47
激活函数 activation function 23, 83
Adaptive Gradient AdaGrad 52
Adaptive moment estimation Adam 55
对抗 adversarial 159
对抗性重编程 adversarial reprogramming 229
智能体 agent 244
自编码器 autoencoder 210

反向传播 BackPropagation 26, 103
随时间反向传播 BackPropagation Through Time 103
词袋 Bag-of-Words 112
批量 batch 22, 138
批量梯度下降法 Batch Gradient Descent 44
批量归一化 Batch Normalization 61
批量大小 batch size 23, 44
束搜索 beam search 152
信念网络 belief network 138
基线 baseline 35, 270
偏置 bias 9, 59

灾难性遗忘 catastrophic forgetting 276
通道 channel 71
分类 classification 8
分类器 classifier 107, 137, 145
编码 code 211
条件型生成 conditional generation 157, 175
持续学习 continous learning 274
连续词袋 Continuous Bag Of Words 200
卷积层 convolutional layer 79
卷积神经网络 Convolutional Neural Network 32, 71, 113, 155
临界点 critical point 38
交叉熵 cross entropy 10, 72, 164
循环生成对抗网络 Cycle GAN 178

数据增强 data augmentation 32, 87
深度学习 Deep Learning 8
去噪自编码器 denoising autoencoder 213
扩散模型 diffusion model 181
降维 dimensionality reduction 211
判别器 discriminator 158